POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Automation of Delta Sigma ADC Filter Design with High-Level Synthesis

Supervisors

Prof. Luciano LAVAGNO

Prof. Mihai LAZARESCU

Co-supervisors

Josef POLAK

Davide MARIZ

Candidate

Shehryar AKBAR

March 2024

Summary

This thesis presents an innovative exploration of the design process of various digital filter IPs used inside a Delta-Sigma ADC. This work has been carried out in collaboration with *Infineon Technologies, Austria* and it is based on the use of High-Level Synthesis (HLS) to create parameterized and reusable digital filters hardware architectures. In this thesis, we focus on Finite Impulse Response (FIR) filters, Polyphase decimation filters and Cascade Integrator Comb (CIC) decimation filters, which are amongst the most common filter structures used in this type of application.

The hardware design of these filters is achieved through Siemens EDA HLS tool 'Catapult'. HLS is a technology that assists with the transformation of a high-level description of hardware (written in C++ or SystemC) into a synthesized netlist and, as by-product, an RTL model. Although significantly faster than writing RTL code, using a higher abstraction level can still be time consuming when designing specific hardware blocks according to user/customer requirements. To this end, we have tried to take the power of HLS one step further by automating the complete process: from concept design in Matlab to synthesizable netlist, until functional verification of the hardware. Through our research, we aim to make this digital design process smoother, faster and adaptable to quick changes which will allow for shorter concept-synthesis loops.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Luciano Lavagno, for his guidance during my master's thesis. Then, I must express my wholehearted gratitude to Josef Polak for providing me with this opportunity and to Davide Mariz, whose everyday support, explanations, and tutoring made this thesis possible. Lastly, I would like to extend my gratitude to Bachhuber Werner from Siemens for his indispensable assistance. It has been a great honor and pleasure to meet and learn from all the staff of Infineon Techonologies Austria, which gave me the incredible opportunity to work on this project in a friendly, productive and enjoyable environment.

Table of Contents

List of Tables IX

List of Figures X

Acronyms XIV

1 Introduction 1

1.1 High-level Synthesis 1

1.1.1 Definition 2

1.1.2 Advantages of HLS 3

1.1.3 HLS flow description 4

1.2 State of the Art 6

1.3 Catapult 10

1.3.1 Catapult flow description 12

2 Methodology of thesis work 17

2.1 Matlab code and reference model 17

2.2 Algorithmic description in C++ 18

2.3 Optimization 19

2.4 Testbench and functional verification 19

2.5 Comparison with existing products 20

3 Design of FIR filter 22
3.1 Architecture and frequency response 22
3.2 Matlab code and reference model 23
3.3 Synthesizable C++ code (Algorithmic description) 25
3.4 Architectural optimizations 28
3.5 Class-based hierarchical design 40
3.5.1 Hierarchy 41
3.5.2 Template Features 43

4 Design of Polyphase decimation filter 47
4.1 Architecture 47
4.2 Matlab code and reference model 49
4.3 Mathematical modeling and manual hardware sharing 49
4.4 Synthesizable C++ code (Algorithmic description) 52
4.5 Architectural optimizations 54
4.6 Results 58
4.6.1 Order:5 , Decimation:2 & 3 58
4.6.2 Order:123 , Decimation:2 & 62 61
4.6.3 Order:28 , Decimation:8 63

5 Design of CIC decimation filter 66
5.1 Architecture and frequency response 66
5.2 Matlab code and reference model 68
5.3 Algorithmic description in C++ 69
5.4 Architectural optimizations 73
5.5 Results 76
5.5.1 Order:3 , Decimation:512 76
5.5.2 Order:4 , Decimation:8 79

6 Design of Filter Chain 82

6.1 Architecture 83

6.2 Matlab code and reference model 83

6.3 Algorithmic description in C++ 86

6.4 Architectural optimizations 92

6.5 Results 94

7 Conclusion 97

List of Tables

6.1 Filter chain specifications 95

List of Figures

1.1 HLS Flow[2] 4
1.2 Table of performances[4] 7
1.3 Quality of results and design effort of HLS compared to hand-written RTL in several case studies[12] 9
1.4 Verification effort comparison for a video decoder compatibility regression 11
1.5 Catapult design flow [16] 11
1.6 Hardware implementation of fully unrolled loop [17] 14
1.7 Pipelining with II=1 [17] 15

2.1 Catapult SCVerify flow[18] 20
2.2 Methodology 21

3.1 FIR filter of order N 23
3.2 Filter response using designed vs quantized coefficient values 25
3.3 Internal architecture of un-optimized MAC and SHIFT loops 29
3.4 Bill of materials after loop unrolling 31
3.5 Simulation result after loop unrolling 32
3.6 Bill of materials after unrolling and pipelining 33
3.7 Internal architecture after unrolling and pipelining 34

3.8 Internal architecture after storing coefficients inside registers 35
3.9 Bill of materials after storing coefficients inside registers 36
3.10 Design optimizations of FIR filter 37
3.11 Simulation result after optimizations 38
3.12 Graphical illustration of slice method[17] 39
3.13 Graphical illustration of half bit rounding 40

4.1 Illustration of decimation by 8 48
4.2 Polyphase implementation of decimation filter. (a) Two polyphase components. (b) Equivalent configuration. [20] 49
4.3 Output equations of order 5 FIR filter 50
4.4 Shifting of input samples inside shift register 51
4.5 Derivation of coefficient index 52
4.6 Derivation of register index 52
4.7 2 multipliers used for 6 taps 58
4.8 Simulation on NCSim showing output with decimation rate of 3 . . 59
4.9 Result of HLS generated design as compared to Matlab results . . . 59
4.10 Comparison of areas for two different decimation rates 60
4.11 Simulation on NCSim showing output with decimation rate of 2 . . 60
4.12 3 multipliers used for 6 taps 61
4.13 Area comparison: Polyphase filter of order 123 designed using HLS flow and RTL flow 62
4.14 Order 123 and Decimation 2 63
4.15 Order 123 and Decimation 62 63
4.16 Area comparison of order 28 and decimation 8 filter 64
4.17 Resource sharing inside polyphase filter 64
4.18 Area saving after optimization 65

5.1 CIC decimation filter of order 1 67
5.2 Design Analyzer view of CIC filter composed of separate sub-blocks 70
5.3 Comb part before optimization 74
5.4 Comb part after optimization 74
5.5 Block diagram view: With pipe 75
5.6 Block diagram view: No pipe 75
5.7 Comparison of areas for CIC filter architectures 75
5.8 NCSim simulation of given CIC filter 76
5.9 BOM of Order 3 CIC filter 77
5.10 Area comparison of order 3 and decimation 512 filter 78
5.11 Area comparison of order 4 and decimation 8 filter 80
5.12 Schedule of order 4 and decimation 8 filter 81

6.1 Filter chain in a hierarchical design 83
6.2 Block diagram view of the filter chain inside Catapult 92
6.3 Scheduling operation showing an un-shared adder 93
6.4 Scheduling operation of a shared adder over 4 ccs 94
6.5 Block diagram view of the filter chain 94
6.6 Area score of filter chain 95
6.7 Area comparison of filter chain with 7 filters 96

Acronyms

HLS

High-level Synthesis

DSP

Digital Signal Processing

DFG

Data Flow Graph

IP

Intellectual Property

ADC

Analog to Digital Converter

FIR

Finite Impulse Response

FF

Flip Flop

GUI

Graphical User Interface

CIC

Cascaded Integrator Comb

DUT

Device Under Test

ccs

Clock cycles

BOM

Bill of Materials

LSB

Least Significant Bit

VLSI

Very Large-Scale Integration

RTL

Register Transfer Level

FPGA

Field Programmable Gate Array

SoC

System on Chip

Chapter 1

Introduction

1.1 High-level Synthesis

The ever-increasing digitization of the physical world and the need for better hardware has resulted in a growing complexity of digital electronic systems. As the complexity of systems increases, it becomes more difficult to not only design but also market them. Even with the best efforts of skilled engineers, embedded systems can become so complex that their development becomes increasingly risky and prone to delays. Against this backdrop of increased technological advancements and faster time-to-market requirement of companies, the traditional method of describing hardware by using hand-written code no longer seems a viable option. These reasons have forced design methodologies and tools to raise the abstraction description levels. In particular, designers would use a High-Level-Language (HLL) to describe the algorithm, instead of describing an RTL architecture.
Moreover, in this era of highly intricate digital systems , there is a need to be flexible and quick whenever design changes need to be implemented either due to changes in user requirements or the need to explore different micro-architectures. Case in point are digital filter chains which are deployed after Delta Sigma ADCs

for the purpose of filtering and low-rate sampling using decimation. The number of filters in the chain, the type of filters and their architectures are parametrized constructs which change with each new user requirement.
This calls for the need to deploy HLS tools whose versatility, easy debugging capabilities and easy optimization options are a perfect match to meet this demand of complex hardware design. Moreover, there exists the possibility also to take the power of HLS tools even one step further and 'automate' the whole process of creating new IP designs which are flexible, reusable and conforming to the desired target requirements with the best possible results in terms of resource sharing and latency.

1.1.1 Definition

High-level synthesis (HLS) is a technology that assists with the transformation of a behavioral description of hardware into an RTL model.
Typically, design projects start with some kind of specification. HLS tools work by converting this abstract functional description (usually written in C/C++) into a language-independent control-data-flow-graph (CDFG) that represents the flow of data. Then the arithmetic operators and control logic are added, and scheduled statically over the clock cycles. In the next step, the functional operations are bound to a physical resource and some optimizations are also carried out. Lastly, the final RTL architecture is generated.
To sum up, the HLS tool performs a translation of the high-level code into RTL-code starting from the description in a high-level programming language (C++, C, SystemC and others). A set of constraints and goals must be specified together with the high-level description as they affect the behaviour of the final hardware and its implementation. The quality of these final results has been showing progress over the course of time as the related concepts and technology of HLS are becoming

more and more mature[1].

1.1.2 Advantages of HLS

1. **Shorter design cycle:** HLS greatly decreases the design cycle time of hardware / IPs by raising the abstraction level and freeing the programmer of low-level details. Functionality is achieved by writing fewer lines of code as compared to HDL. HLS tools automatically map abstract transactions to actual RTL interfaces, add necessary hardware details, handshake protocols and I/O interfaces etc. Moreover, the target technology can be changed instantly and caters to ASICs and FPGAs both.

2. **Micro-architectural exploration:** Due to fast synthesis process, it is possible to explore a large number of different architectures, early on in the design phase or even in the concept phase. This exploration of different implementations is done by applying constraints in the HLS tools (either through GUI or by using pragma directives) and not changing the source code. Comparative analysis can then be done of the different designs in terms of latency, throughput, area and scheduling etc. It is also possible to compare the utility of implementing memories versus register arrays.

3. **Documentation:** HLS tools keep a track of all the decisions carried out which makes the documentation process easier to make and maintain. After successful synthesis, many output reports are automatically generated e.g. bill of materials (BOM), scripts and timing / area reports etc.

4. **Faster verification:** HLS tools enable very fast simulation in comparison to RTL flow. Functional verification is instantly possible by using the same testbench to simulate the C++ / System C model and the synthesized RTL code. This eliminates the need to write extra testbench code for testing the

design at the hardware level and provides one-on-one matching between C++ and generated RTL.

5. **Design optimization:** It is possible to do rapid iterations from C++ to alternative RTL implementations for the purpose of optimizations. HLS tools provide excellent optimization techniques which include arithmetic optimizations, bit-width trimming, dead code removal, pipelining, fine / coarse grain resource sharing and low power design techniques (multiple clock domains, multi-level clock gating).

1.1.3 HLS flow description

The HLS flow description can generally be described in few steps which are summarized below:

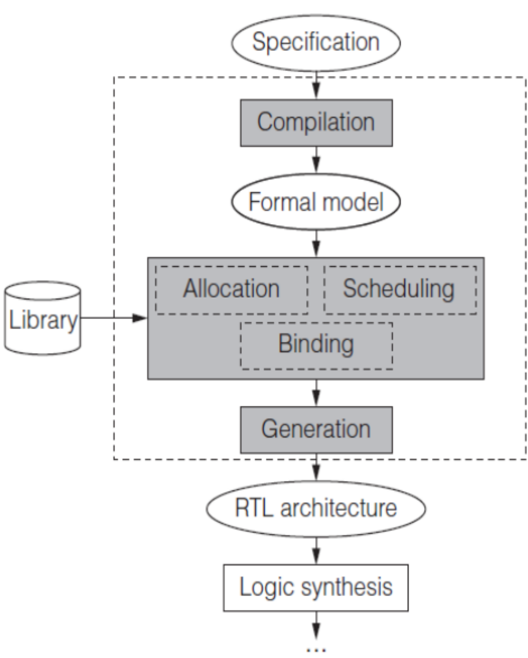


Figure 1.1: HLS Flow[2]

1. **Compilation:** The first step includes generation of a formal model through compilation which produces a data flow graph (DFG). Optimizations like false data dependency elimination and dead code removal are carried out and the representation of control flow is also made by extending DFG representation to control DFGs (CDFG). It shows the control flow between basic blocks and the data dependencies inside them.

2. **Allocation:** This step provides the number and the type of allocated resources but this is not the only step for allocation of resources. Some resources could be added during the scheduling and the binding step also. The allocation is done using the components of a specified library, where also timing, power and area properties of the components are contained.

3. **Scheduling:** The functional operations are scheduled into cycles. These functional operations can be chained, so that their outputs feed other successive operations, or scheduled to be executed in parallel, when no data dependencies occur. The final scheduling graph will depend on the usage of options like pipelining and loop unrolling.

4. **Binding:** In the binding phase, the tool checks if the allocated resources and scheduling allows resource sharing or not. As a result, every functional operation, variable and connection is bound to a physical resource. Since area and delay are estimated as early as possible, the binding step can perform some kind on optimization of the architecture.

5. **Generation:** This step takes all the decisions taken in the allocation, scheduling and binding steps and it generates the final RTL architecture. The RTL code is mostly available in VHDL and Verilog both.

1.2 State of the Art

As mentioned in section 1.1, HLS is an automated technique that significantly influences the design of digital circuits. Numerous scholarly works have examined and established the superiority of this novel technique over the conventional design flow over the years.
The works that ultimately reach a conclusion (positive or negative) about HLS as a method for designing DSP components and some other applications, will be highlighted in this section.
[3] is somewhat connected to the scope of this work in which one of the earliest versions of Synopsis HLS tool was used for the design and VLSI implementation of a 'fast' FIR filter. And the results are compared with the conventional methods also, which are shown to be quite encouraging. T.Ognunfunmi and S.Desai used Verilog hardware description language as the input language to the HLS tool, which was deemed as somewhat *"high level"* in 1994. At that time, C-like language had not been introduced for the HLS tools and this work can be classified as being in the midst of Generation 2. According to the report, the tool's results were quite remarkable for its time, but they fall short of what modern tools are capable of producing.
Hanbo, Shaojun, and Yigang conducted an analysis of FIR filter implementation in 2015[4], which can be categorized as one of the modern works. Following an explanation of how HLS differs from standard RTL design, this work shows the FIR filter implementation (20-order low-pass filter with 16-bit input data) using three separate tools: **Vivado HLS**, **LabVIEW FPGA**, and **DSP Builder**. Although the tool utilized in our study, Catapult HLS, is not included in the comparison, we can still draw meaningful deductions from this study.
The paper discusses the general advantages of using HLS tools and offers a brief introduction to them also. Furthermore, it goes into detail about the project design

flows and the corresponding platforms, as well as the optimization strategies they have to offer. It also furnishes comparison against the traditional method of IP core design as well, with focus on resources occupation, synthesis time, optimization options, the highest frequency of devices running and latency.

Development Tools	Resources Occupation		Synthesis Time (to RTL)	Optimization Options	The Highest Frequency	Latency
	LUTs	DSPs				
VIVADO HLS	236	44	25s	Exist, plenty	229.86MHz	15
VIVADO (IP core)	89	1	65s	Exist, plenty	222.22MHz	44
DSP Builder	111	2	14s	Null	180.54MHz	48
Quartus II (IP core)	12	19	26s	Exist, few	116.25MHz	78
LabVIEW	9623	5	572s	Null	68.68MHz	-

Figure 1.2: Table of performances[4]

From the table above, it can be seen that one of the most distinct features of HLS tools is the ability to shorten the synthesis time. In case of Vivado HLS and DSP Builder, the time is more than halved. Also, the performances for highest frequency and latency are better, as well as provision of more optimization options. In case of resource occupation, most of the DSP blocks are occupied by Vivado HLS. Its DSP block demands a lot of logical resources due to the numerous optimization options it requires, (e.g. splitting the arrays and unrolling the loops), but it also performs exceptionally well when the DSP unit is used extensively. Improving the performance of the design outcomes requires a significant investment of resources. However due to IP core's complete optimization, it performs better while requiring comparatively fewer hardware resources.

HLS has applications outside of the realm of digital signal processing also[5]. It is also extensively utilized in algorithm validation, design space exploration[6][7], migration, hardware acceleration[8], and other domains since it can be explained at the algorithmic level[9].

The work of C. Fibich and S. Tauner[10] evaluates the use of two HLS tools (Vivado HLS and PandA Bambu) for offloading computation-intensive algorithms to an FPGA implementation using four case studies (*matrix multiplication, FIR filter, Atmel barcode reader and QR decoding library*). PandA Bambu[11], is an open-source C-based HLS tool that is under development at the Politecnico di Milano. As with Vivado HLS, the tool supports a subset of standard C. The case studies had to be combined into appropriate designs with measurement facilities and appropriate interfaces for data and control information transmission in order to quantify runtime performance on hardware.

The software implementations were evaluated both on a 64-bit Intel x86 CPU and an ARM Cortex A9 SoC,whereas the hardware implementations were obtained both from Vivado HLS (for a Zynq 7010 SoC FPGA), and PandA Bambu (for Altera Cyclone V SoC FPGA).

The paper goes into further detail regarding setup of benchmark environments, estimation of timing results and compiler optimization options for software etc, but overall this conclusion is drawn that the execution times of the hardware implementations are slower when compared to their software counterparts, even after taking into account the influence of the interface to/from the accelerator.

This shows that the performance of HLS is also application-dependent and sometimes it may not have the desired effect of getting better results. This is further elaborated by the Masters thesis work of Tero Joentakanen of Tampere University of Technology[12] in which he investigates the compliance of HLS with existing ASIC implementation flow, for different applications. The summary of previous works on HLS design as compared to traditional manual design is depicted in Figure 1.3

Author	Ref	Tool	Application	Target platform	Area (ASIC) / Resources (FPGA)	f_{max}	Effort estimation
Ollikainen P.	[30]	N/A	DSP + control	ASIC	+15%	–	−17%
Järviluoma J.	[15]	HDL coder	IQ data scaling	ASIC	−29%	–	–
Zhu Q. & Tatsuoka M.	[53]	Stratus	DMA controller	ASIC	**−38%**	–	−66%
Sun Z. et al.	[39]	N/A	AES encryption	ASIC	+37%	+1.5%	**0%**
Torppa E.	[45]	Catapult	Adder-tree FIR	ASIC	−30%	+1.6%	
			Systolic FIR	ASIC	−11%	±0%	
			Basis functions	ASIC	−36%	−3.3%	
			Adder-tree FIR	FPGA	+36% LUT, −31% FF	−8.5%	
			Systolic FIR	FPGA	+23% LUT, +11% FF	**−29%**	
			Basis functions	FPGA	+35% LUT, +0.5% FF	**+10%**	**−80%**
Kivimäki I.	[18]	Vivado	Signal correction	FPGA	**+173% LUT, +34% FF**	+7.3%	−70%
Zwagerman M. D.	[54]	Vivado	Image processing	FPGA	+61% LUT, −3% FF	−10%	−55%
Karras K. et al.	[17]	Vivado	Memcached server	FPGA	−22% LUT, −35% FF	–	−50%

Figure 1.3: Quality of results and design effort of HLS compared to hand-written RTL in several case studies[12]

One thing becomes extremely evident from the analysis of all these works: performance varies greatly depending on the tool and the target technology. Upon looking at all the results of these works, it is shown that area ranges from -38% to +173% and maximum frequency from -29% to +10%. For the sake of clarity, it is imperative to note that these numbers are the variations between the HLS designs with respect to the manual one.
Additionally, the same application, but for two distinct target platforms, performs entirely differently. Case in point is the design of Systolic FIR created in Catapult for both FPGA and ASIC. Catapult shows better performance when the target is ASIC (-11% area), whereas it displays poor results (+23% LUT, +11% FF and -29% f_{max}) for FPGA target technology.
One feature, however, remains consistent in all these case studies and that is the effort estimation required to design all these applications with HLS. The speed of implementation in the front end design is apparent and results demonstrate a 2-5 times higher design productivity as compared to manually written RTL. This can be attributed to the fact that designers can concentrate on functionality rather than implementation specifics. Also, the fact that high-level description is at the same abstraction level as the algorithm model, makes it possible to reuse code/model

specifics in the HLS design.

1.3 Catapult

According to the fact sheet provided by Siemens Digital Industries Software[13], their EDA tool “Catapult” is one of the leading HLS solutions for ASIC and FPGA design. Supporting C++ and SystemC, designers work in their preferred language, moving up in productivity and quality. With 80% less coding and simulation speeds up to 1000x faster than Verilog[14], design and verification using Catapult is faster and easier. This process gives designers the possibility to move up to a higher abstraction level for both design and verification of ASICs and FPGAs. Both time-to-market and freedom to automatically explore different solutions are boosted and a fully optimized and error-free hardware implementation is quickly achieved. Catapult has integrated High-Level Verification (HLV) tools and methodologies that enable designers to complete their verification signoff at the C++ level with fast closure for RTL. Different architectural solutions can be explored by setting constraints such as folding/unfolding, pipelining, retiming etc. Shifting to a new target technology is also easier and faster by selecting a different library and no need of changing the source code.
[15] discusses a case study published by NVIDIA® where the company adopted Catapult HLS to design a JPEG encoder/decoder and obtained monumental time/-effort savings with respect to traditional RTL design methodology. By adopting this flow, NVIDIA® was able to simplify their code by 5X, reduce the number of CPUs required for regression testing by 1000X, and run 1000X more tests to achieve higher functional coverage of their designs. HLS decreased design time by 50% and overall development time, including verification, by 40%.

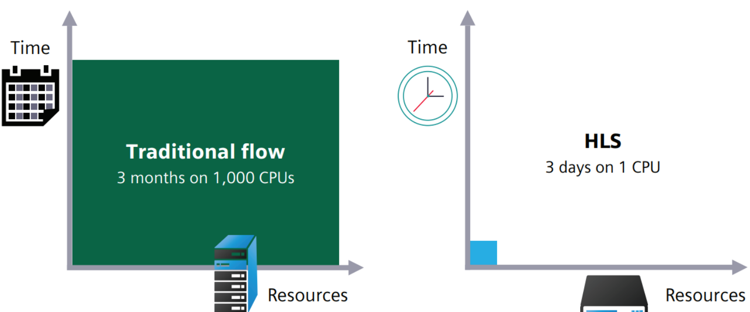


Figure 1.4: Verification effort comparison for a video decoder compatibility regression

The highly interactive Catapult workflow provides full visibility and control of the synthesis process, enabling designers to rapidly converge to the best implementation for performance, area, and power. After the RTL has been synthesized, Catapult automates a complete verification infrastructure reusing the original C++ or SystemC testbench to exercise the generated RTL.

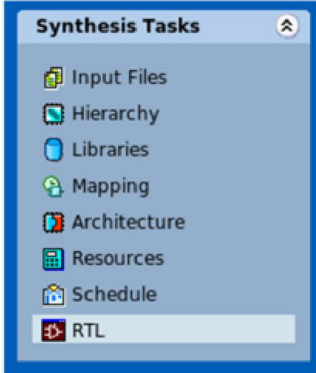


Figure 1.5: Catapult design flow [16]

1.3.1 Catapult flow description

A user friendly and accessible GUI is provided by Catapult in order to help attain the final synthesized netlist. There are some simple steps which are needed to be performed, details of which are mentioned below:

- **Input files:** The overall design flow starts with the inclusion of .cpp files which can be source code files also as well as testbench files. Header files need not be included, but it is important to put them in the same directory incase they are used. It is possible to exclude files from the synthesis process e.g. testbench.

- **Hierarchy:** Catapult requires establishing a hierarchy in the design in order to output the required structure. This hierarchical setting is set for functions and classes both, and the type of setting greatly influences design parameters such as timing, pipelining and resource sharing. There are three possible statuses: Top, Block and Inline.
  For simple, single block designs, the Top setting is by default. For hierarchical designs containing multiple blocks, it is necessary to choose the type of setting. In a design, only one function/class can be specified as 'Top' which will further call other functions/classes. They can be specified as 'inline' or 'block'. Inline functions are contained inside that particular block so there is no special communication established between them. In case of 'block' setting, it designates a separate sub-block inside the top-level block and communication is only possible through dedicated ac_channels. To define this setting in the source code, *hls_design pragma* directive is used, followed by one of the three types.

- **Libraries:** This step requires selecting the technology which we have to use for synthesis. To specify the technology, the target RTL synthesis tool and a

device are selected. Based on this selection, an associated list of IP libraries appears in the Compatible Libraries field on the right. The set of libraries consists of a base library and some additional IP libraries (i.e. for RAMs and ROMs). In this thesis work, a target library of *130 nm* is used when I have to compare my results with those of Infineon IPs. In other cases, (such as design of generic architectural flows) I have used inbuilt synthesizer provided with Catapult, with target library *Nangate 45nm*.

- **Mapping:** This step allows setting the clock frequency, period, duty cycle, offset and uncertainty. Advanced signals like reset (synchronous or asynchronous), enable signal for clock gating, transaction done signal (for synchronization of input/output signals) and triosy signal (to indicate completion of I/O transaction when there is no specified handshake). For this thesis work, we have used ac_channel interface which has in-built ready/valid handshake, also known as the *wait handshake*. Further details on this will be discussed later.

- **Architecture:** This step is one of the most crucial ones in the whole design flow as it allows the designer to set different constraints. Changing these constraints has a big impact on the overall architecture and therefore, particular attention should be paid while deciding on these settings. The constraints are set on loops, I/O interfaces, arrays and memories. The interface is of different types and can be set depending on multiple parameters, and it defines the process level handshakes between the design and the outside world. We have used the wait protocol in my designs (atleast for the top level) which is a two-way handshake with valid/ready signal, and it is also the default handshake for ac_channel ports (will be discussed later). It automatically determines whether the source is ready to send data, and the output is ready to receive it. If the source is not ready, or if there is no data to send, then the design stalls. The arrays inside the 'Core' are the register memories used for storing

data (coefficients, input data, output data, accumulator data etc). Then we can select the settings for the 'Loops' which is useful for optimization and forms an integral part of the whole design process. The two main techniques provided by Catapult for this purpose are "Pipelining" and "Unrolling", both of which are used for exploiting parallelism between loop iterations.

– **Unrolling:** Loop unrolling is the primary mechanism to add parallelism into a design. This is done by automatically scheduling multiple loop iterations in parallel, when possible. The amount of parallelism is controlled by how many loop iterations are run in parallel and it is only possible if there are no data dependencies in between the successive iterations. Loop unrolling can theoretically execute all loop iterations within a single clock cycle as long as there are no dependencies between successive iterations, also called 'fully unrolled loops'. It is also possible to do 'partial unrolling' which only duplicates the loop "x" number of times e.g. unrolling a loop by a factor of 2 has the equivalent effect of manually duplicating the loop body two times and running the loop for half as many iterations.

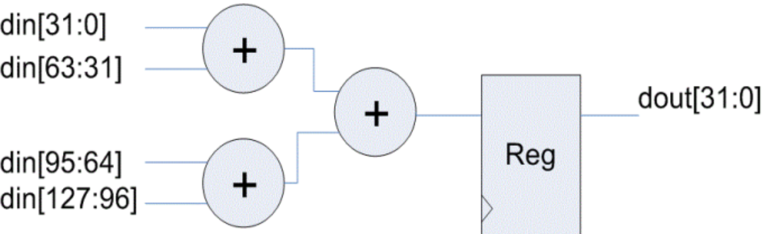


Figure 1.6: Hardware implementation of fully unrolled loop [17]

– **Pipelining:** It allows new instructions to be read before the current instruction has finished. Loop pipelining allows the execution of the loop iterations to be overlapped, increasing the design performance by running them in parallel. The amount of overlap is controlled by the 'Initiation

Interval (II)'. This also determines the number of pipeline stages. The Initiation Interval (II) is how many clock cycles are taken before starting the next loop iteration. Thus an II=1 means a new loop iteration is started every clock cycle. The initiation interval is set on a desired loop either as a design constraint in the HLS design environment, or alternatively can be set using a C++ compiler pragma.

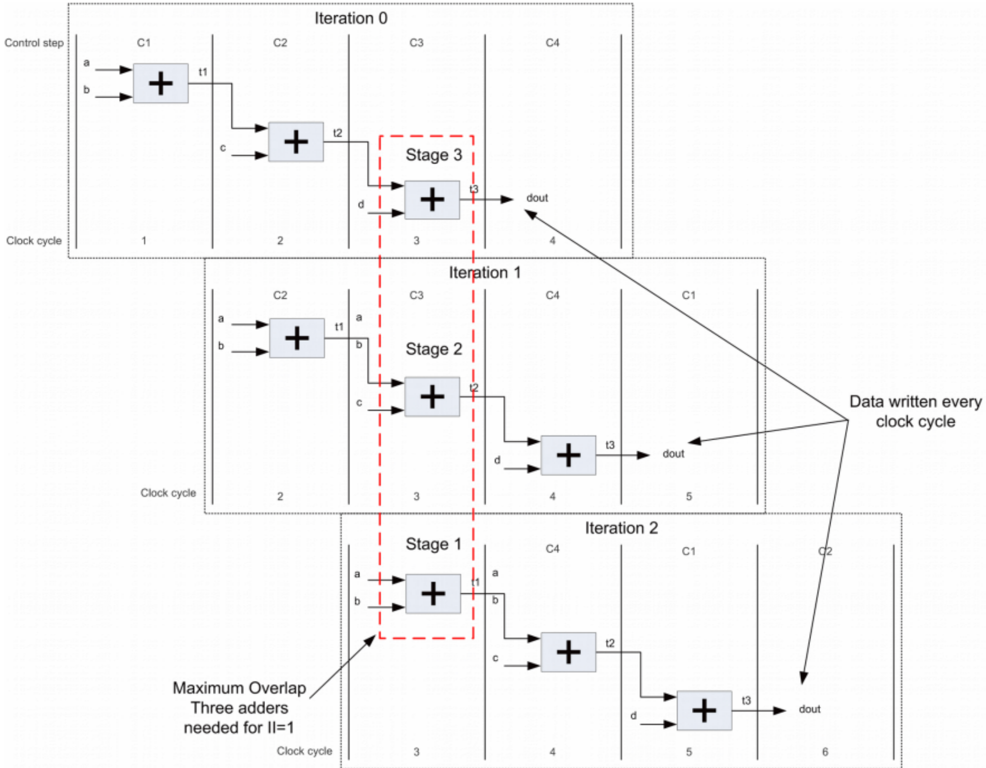


Figure 1.7: Pipelining with II=1 [17]

As shown in Figure 1.7, pipelining with II=1 results in a new iteration at each clock cycle. This results in an increased number of adders also.

- **Resources:** This option shows the resources will be utilized in the RTL composition, and the designer can choose which components to use (adder, multiplier etc). This step can be skipped in which case Catapult automatically

chooses the best components following the directives of the constraints at the previous steps.

- **Schedule:** In this step, Catapult generates the whole schedule of the design in the form of Gantt chart. The user can view the operations done in each clock cycle and compare their execution times. For each loop, the number of control steps (C-steps) are shown as well as the sequence of operations inside them. Selecting a data object in the schedule window highlights the object in all columns and displays arrows in the gantt chart to show dependencies between the selected data object and other operations.

- **RTL:** This last step is used to generate the HDL code in Verilog and VHDL both, and it is available in the 'output' folder. Extra information in the form of reports is also generated like *Bill of Materials (BOM)* which shows the pre and post-allocation resources along with area etc.
  At the end, Catapult displays some main characteristics of the design: latency, throughput, area and slack.

Chapter 2

Methodology of thesis work

The methodology adopted for this thesis work follows the following main 5 steps:

2.1 Matlab code and reference model

Concept design of the required filter is carried out in Matlab according to some given specifications. The filter can be anyone of the following four types:

- FIR filter
- Polyphase decimation filter
- CIC decimation filter
- Filter chain

For generalization purposes, the specifications are kept generic and can be changed later on. Based on these specifications, we get numerical output results from the Matlab flow.
These specifications are also useful for obtaining further data / information about the filter which can be used to devise all the input parameters while doing the

design in hardware later e.g. Matlab helps to obtain filter coefficients in case of a FIR filter when its inbuilt function is used and provided with the required parameters. The output frequency response can be shown as well as the list of output results is obtained. This result acts as the golden reference model against which result from HLS design is compared and the matching is expected to be 100%.

2.2 Algorithmic description in C++

The next step is writing the algorithmic depiction of the filter model which serves as the base of HLS coding in C++. This model defines the core functionality of the filter and focuses on the deployment of resources (multipliers, adders, internal shift registers etc) in the correct order.
The idea is to write the source code in fully generic way such that it can be used to design the specified type of filter for any input specifications as required by the user. Writing a specific source code for some target requirements always yields the best results, but this thesis work is aimed to generalize the implementation of the code in such a way that it is able to provide the best implementation of any particular filter during run time synthesis.
Subsequent to this, the framework of this working model is also developed which consists of the top-level block, multiple sub-blocks (if required), interconnections between the blocks and type of hierarchy (class or function based). This part of the code does not explain the intrinsic behaviour of the filter, but rather builds up the design interfaces with the outside world and comprises of port definitions, direction, bit-widths and data types. By default, Catapult builds synchronous designs which means that all outputs of the top-level design are registered to guarantee that timing is met when connecting to another design.

2.3 Optimization

This part of the process deals with further optimizing the model in order to obtain better results in terms of area, latency or throughput. Usually, some basic techniques are employed like pipelining with a specific initiation interval and loop unrolling. Area and throughput are mostly a trade-off, but the final decision depends on the filter design and customer requirement. In case of multi-rate filters, resource sharing can be done even without decrease in throughput, the details of which will be discussed later in this document. The correct utilization of these optimization techniques also depends on the hierarchical design as well as the method of application of constraints in the correct scenario.

2.4 Testbench and functional verification

The verification flow is divided into two steps:

- Pre-synthesis validation that the C program correctly implements the required functionality.
- Post-synthesis verification that the generated RTL code performs as expected.

This functional verification of the Device Under Test (DUT) is performed by writing a testbench in C++. The same testbench is used to check validity of the C model as well as the synthesized RTL which also ensures that the RTL functionality is identical to the C source code. C++/SystemC can execute upwards of multiple times faster than Verilog/VHDL RTL simulation, so the test coverage comes from this high level testbench. First of all, the functional correctness of the DUT is checked by comparing the results of the C code against the golden reference model obtained from Matlab. This can be easily done by capturing the output values of the DUT inside the testbench and checking its numerical correctness against the

corresponding golden value.
Secondly, the RTL code also needs to be checked. For this purpose, the 'SCVerify' flow is used which is an automated verification flow designed for use with Catapult. SCVerify is set up at the beginning of the synthesis project and then the files needed to simulate using SCVerify flow are generated at various stages of the synthesis process. It uses the same testbench to compare the C++ design to the RTL and check that the outputs of the RTL design match the outputs of the C++ design; demonstrating functional equivalence between the C++ and the RTL for the test vectors provided in the testbench.

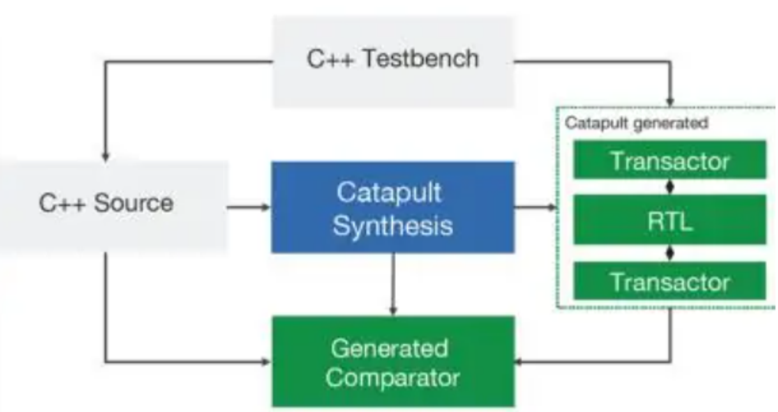


Figure 2.1: Catapult SCVerify flow[18]

2.5 Comparison with existing products

These specifications are provided by Infineon and correspond to an IP which had been developed by the company using RTL programming, and deployed in some integrated circuit.
Comparisons are made between the design generated by the automated HLS methodology and the design provided by Infineon (both of them having the same

specifications and synthesized by the same tool), and major differences are noted. Mostly, only the **area** is compared along with static power consumption.

This complete methodology of a fully automated flow for designing parametric filter structures is depicted in Figure 2.2.

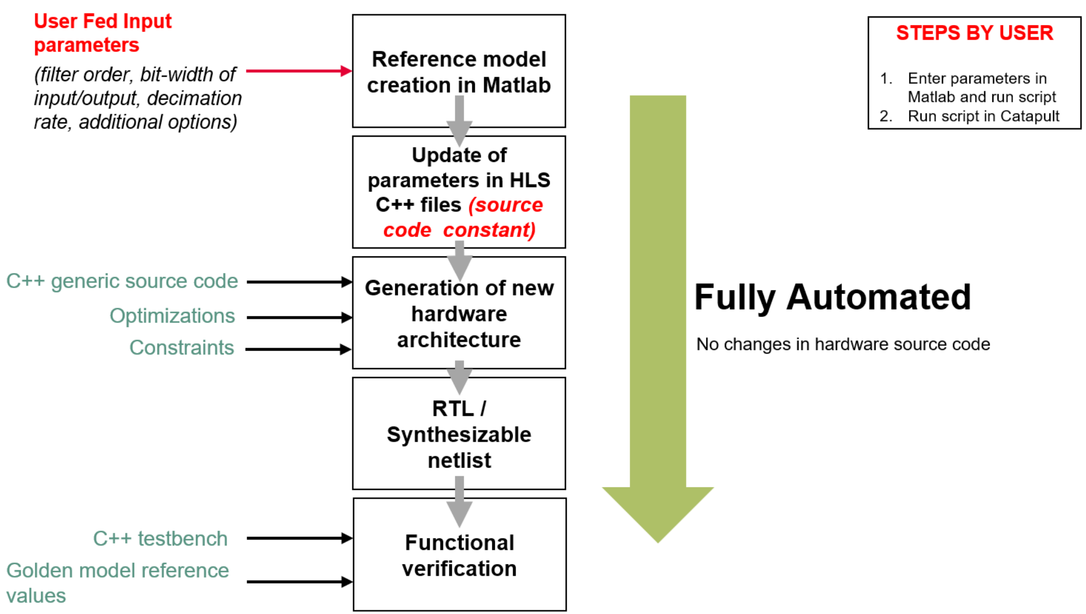


Figure 2.2: Methodology

Chapter 3

Design of FIR filter

3.1 Architecture and frequency response

FIR filters are feedback-free digital filters with a finite impulse response, whose output values y(n) are calculated from a stream of input signals with value x(n). There are several key components that make up an FIR filter. One of the most important is the impulse response, which is the filter's response to a unit impulse. This response can be used to characterize the filter's behavior and to analyze its performance. The impulse response is typically a finite sequence of samples, which is why FIR filters are also known as finite impulse response filters. This impulse response is defined by filter coefficients, which are the values that determine how the filter processes the signal. The filter coefficients are typically determined by a design algorithm that takes into account the desired frequency response and other design constraints, such as filter order and passband ripple.
The number of filter coefficients, or taps, determines the filter's length and complexity. Longer filters can achieve sharper frequency response characteristics, but require more processing power and memory to implement. FIR filters work by convolving the input signal with the filter coefficients. This process essentially

multiplies each sample of the input signal by a corresponding filter coefficient, and then adds up the results to produce the output signal. The choice of filter coefficients determines the filter's frequency response, or how it affects different frequencies in the signal. A FIR filter has the simple structure of a tapped delay line, where the value in each register / delay element is multiplied by a coefficient and the sum of them forms the output. The two variables are the number of taps and the value of the coefficients. Such digital filters are usually determined by their "order", which results from the number of delay elements. For example, a 2nd order filter has two delay elements and three taps.

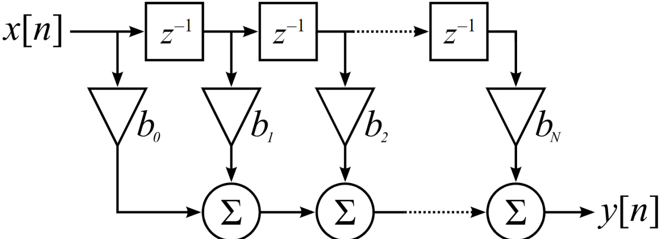


Figure 3.1: FIR filter of order N

FIR filters can be designed to have various frequency responses, such as low-pass, high-pass, bandpass, or band-reject. In our case, we have chosen to implement a low-pass filter. FIR filters can be of various types and each have their own architecture, unique characteristics and applications. They can be further divided into symmetric and non-symmetric impulse response filters, and the details of this will be discussed later during the optimization phase.

3.2 Matlab code and reference model

To design a *generic and parameterized* filter in HLS, which could then later be compared with the one given by Infineon, first there was a need to specify its

functionality and optimize it for the best implementation. For this purpose, a general Matlab code for a FIR filter is written with the following specifications:

- Input sine wave which is obtained by the summation of two separate pure sine waves; one with a frequency of $f_1 = 500\ Hz$ and the second with a frequency of $f_2 = 3.5\ kHz$.
- The bit-widths of the input and output ports/channels at the hardware level are $nb = 8$ so the Matlab values also have to be quantized on the same number of bits, so that the correct frequency response can be obtained and matched.
- The order of the filter is kept at $N = 32$.
- The sampling frequency is $f_s = 10\ kHz$, according to the Nyquist rate.
- The cut-off frequency is kept at $f_c = 3\ kHz$ which will filter the higher frequency f_2 and allow only f_1 to pass. The normalized cut-off frequency f_0 is obtained by dividing f_c by the Nyquist frequency (half of sampling frequency f_s). So $f_0 = 0.6$.

The coefficients for this type of filter are obtained by passing the parameters through the inbuilt ***fir1()*** function which returns the coefficient values. These values are also quantized to the number of bits which are available at the hardware. With these specifications, the frequency response of the filter (with real and quantized values both) is obtained. The response is shown in Figure 3.2.

The output values are calculated by using the inbuilt filter function which takes the quantized inputs and quantized coefficients as its parameters. The output is again quantized to the required number of bits and saved to a separate file which will be used as the golden reference model.

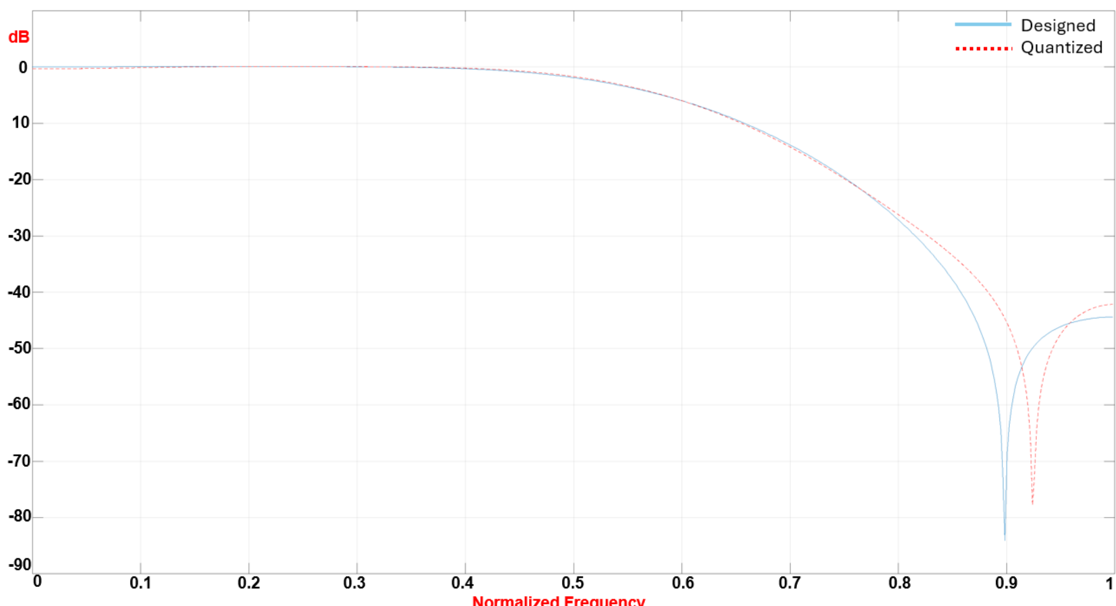


Figure 3.2: Filter response using designed vs quantized coefficient values

At the end, the Matlab script is used to automatically edit the **top.h** header file which contains all the parameters and information used by the FIR filter source files for the design process.

3.3 Synthesizable C++ code (Algorithmic description)

Catapult asks as input an algorithm that is synthesizable. It means that the inputs are not generated internally but are taken from the outside one at a time (from a testbench), because they are the results of a sampling. In the first implementation, the coefficients are passed to the filter from outside as direct inputs. To be synthesizable, the project must include three different files; the header file (.h), the description of the algorithm (.cpp) and the testbench (.cpp). The header file contains declarations of the variables, constants and functions. The core .cpp file

contains the synthesizable description of the FIR filter along with all the loops. The testbench is used to read the inputs from an external file, pass the values in the core file, receive the results, compare them and write them to an external file. It is pertinent to mention here that the bit-accurate datatypes have been used in the HLS design using the ac_fixed library. Instead of using 'double' datatype which is 64 bits wide and requires a fair amount of hardware to implement, the use of AC fixed-point (ac_fixed) datatype allows the designer to use less number of bits. This allows to save resources and improve speed in the hardware implementation. We have already quantized all the signals on 15 bits in the fractional part as per the model; so an example declaration of input signal will be ac_fixed <16,1,true> which translates to a total bit width of 16 bits, out of which 1 bit is used for the integer part and rest 15 bits for the fractional part. And the value is signed. To obtain the full accuracy inside the filter, there is a need to determine the intermediate bit-widths of the accumulators. For this purpose, instead of manually calculating the output widths of the multiplier and adder, we use a feature of ac library known as derived data types. To define the smallest full-accuracy data type which is the product of two ac_fixed operand data types, we use:

```
typedef OperandAType::rt_T<OperandBType>::mult ProductType;
```

And to define the fixed-point type for the summation of these products, we use:

```
typedef ProductType::rT_unary::set<TAP_COUNT>::sum SumType;
```

Another useful way to determine the intermediate bitwidth of the signals inside the FIR filter is to calculate it mathematically inside the source files, as a pre-processor directive. This part of the code is not synthesized but it can be used to calculate a number of useful and important parameters on the fly.

For the above mentioned purpose, the following mathematical formula has been utilized:

$$Accumulatorbitwidth = x_i(n) + x_c(n) + floor[log2(tap_count)] \qquad (3.1)$$

Where $x_i(n)$ are number of bits of input samples, $x_c(n)$ are number of bits of coefficient samples and tap_count are number of taps of the filter (order + 1).

In the C code, this is used in the following way:

```
1  template <class inType, class coeffType, int filt_order>
2  struct find_inter_type_fir {
3    enum {
4      W_IN     = inType::width,
5      I_IN     = inType::i_width,
6      W_CF     = coeffType::width,
7      I_CF     = coeffType::i_width,
8      S        = inType::sign,
9      outF = W_IN - I_IN,
10     // Formula to calculate intermediate bitwidth
11     // outW = floor(log2(tap_count)) + W_IN + W_CF + (!S)
12
13     outW = ac::log2_floor<filt_order + 1>::val + W_IN + W_CF + int(!S
       ),
14     outI = outW - outF
15   };
16   typedef ac_fixed<outW, outI, true> int_type_fir;
17 };
```

The algorithmic C library has been utilized at line 13 to statically compute the required bitwidth. This implementation results in a fully parameterized structure which can be utilized inside the generic source FIR filter. For every instance of the

FIR filter instantiated, there exists the option to calculate different accumulator bitwidths, depending upon the individual filter characteristics.
This generalization feature relieves the programmer to manually calculate the intermediate accumulator widths or even calculate it from Matlab etc and pass it as a separate parameter to the source code. This is especially useful when generating filter chains, which are discussed in detail in Chapter 6.

The following specifications have been used to synthesize the filter and these will be kept the same (mostly) for the subsequent filter designs also:

- RTL Synthesis tool: Oasys RTL
- Library: Infineon technologies library
- Frequency: 100 MHz

The source code of the filter contains two loops which make up the entirety of the whole implementation:

1. **SHIFT LOOP**: The SHIFT_LOOP is merely a shift register which is used to shift all the values in the array to the next location in the array, while the first location takes the input sample.
2. **MAC LOOP**: The MAC_LOOP is used to calculate the multiplication of all the corresponding shift register values with their coefficients and add them together.

3.4 Architectural optimizations

The result of the above synthesized filter shows and Area = **29955.33** um^2 and throughput = 68 ccs.
Figure 3.3 shows the internal architecture of MAC and SHIFT loops of such a filter architecture.

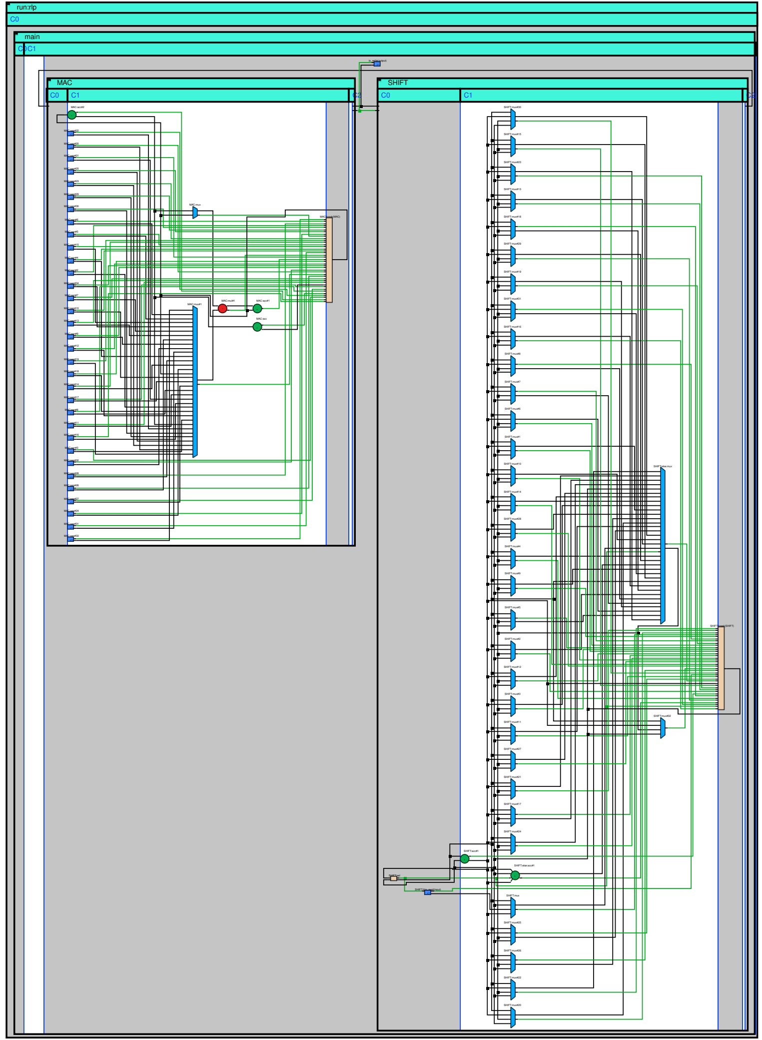


Figure 3.3: Internal architecture of un-optimized MAC and SHIFT loops

Because we have not set any constraints on the throughput, only 1 multiplier is being used to calculate all the input-coefficient products, which takes 33 ccs (equal to number of taps in the filter), colored red in the figure. Moreover, the input samples are being shifted in the shift register in a sequential way, which further takes 33 ccs. Further 2 ccs are being utilized extra at the input and output which

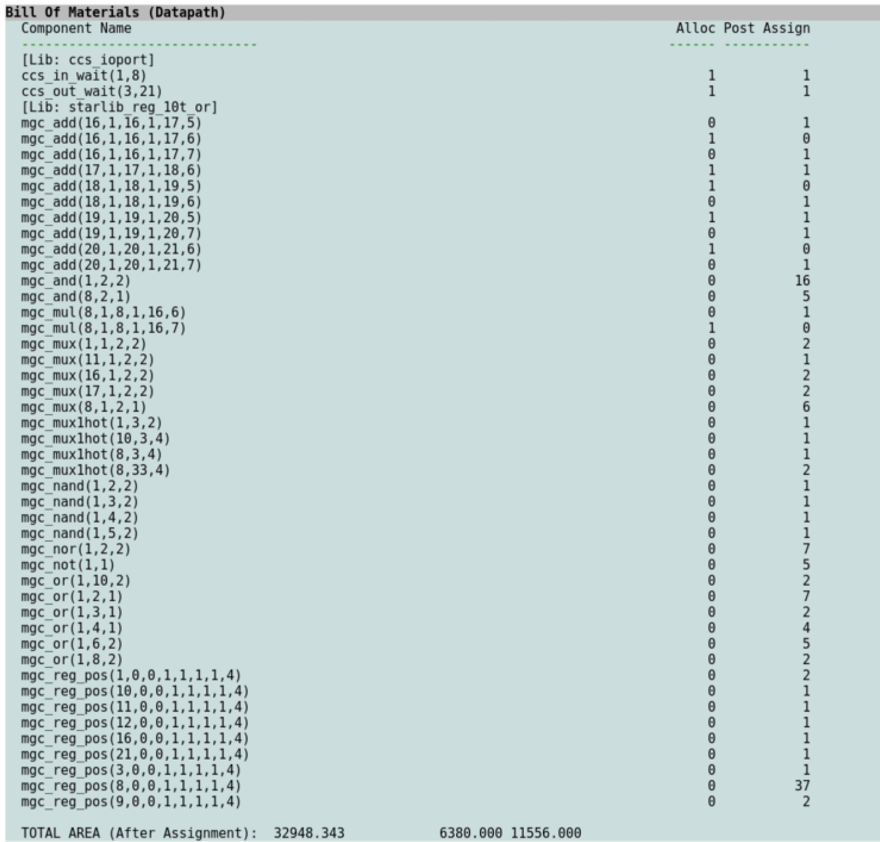
results in a total throughput of 68 ccs at the end.
This throughput is impractical in a real filter, and it will be optimized later on.
However, there a number of optimizations which are carried out on this design to enhance the overall performance.

- **Loop unrolling**: By default, loops are implemented in hardware as a single loop iteration, and the same hardware is used for subsequent iterations of the loop. This type of solution is referred to as a Rolled Loop. So if there are 10 loop iterations, it would take 10 clock cycles to calculate the result (assuming each iteration takes 1 clock cycle to complete). A sequencer is created in the design which keeps reusing the same hardware for each separate instance of the loop.
  If the hardware to implement a loop iteration is duplicated for as many times as there are iterations, the loop is said to be fully unrolled. The hardware resources are generally substantially increased but the result is calculated in a single clock cycle as all the calculations are being done in parallel.
  In my design, loop unrolling has been done for both the SHIFT_LOOP and the MAC_LOOP. The SHIFT_LOOP can be easily and efficiently implemented in a single clock cycle so loop unrolling makes sense here. Although this procedure can be done by using the GUI, I have implemented this step by introducing a design constraint in the source code.

  #pragma hls_unroll yes

  Only unrolling the shift loop improves the area as well as the latency, throughput and timing slack.
  The MAC_LOOP is also then fully unrolled which results in a small increase in area. Although normally, by performing this step, the number of multipliers increases but because we have not set any constraints on the incoming sample

rate, the design still takes 35 ccs to perform the computations by using only one multiplier. Note that the throughput has decreased by half because now, it takes only 1 cc to shift all the samples in the shift register instead of 33.



Bill Of Materials (Datapath)

Component Name	Alloc	Post Assign
[Lib: ccs_ioport]		
ccs_in_wait(1,8)	1	1
ccs_out_wait(3,21)	1	1
[Lib: starlib_reg_10t_or]		
mgc_add(16,1,16,1,17,5)	0	1
mgc_add(16,1,16,1,17,6)	1	0
mgc_add(16,1,16,1,17,7)	0	1
mgc_add(17,1,17,1,18,6)	1	1
mgc_add(18,1,18,1,19,5)	1	0
mgc_add(18,1,18,1,19,6)	0	1
mgc_add(19,1,19,1,20,5)	1	1
mgc_add(19,1,19,1,20,7)	0	1
mgc_add(20,1,20,1,21,6)	1	0
mgc_add(20,1,20,1,21,7)	0	1
mgc_and(1,2,2)	0	16
mgc_and(8,2,1)	0	5
mgc_mul(8,1,8,1,16,6)	0	1
mgc_mul(8,1,8,1,16,7)	1	0
mgc_mux(1,1,2,2)	0	2
mgc_mux(11,1,2,2)	0	1
mgc_mux(16,1,2,2)	0	2
mgc_mux(17,1,2,2)	0	2
mgc_mux(8,1,2,1)	0	6
mgc_mux1hot(1,3,2)	0	1
mgc_mux1hot(10,3,4)	0	1
mgc_mux1hot(8,3,4)	0	1
mgc_mux1hot(8,33,4)	0	2
mgc_nand(1,2,2)	0	1
mgc_nand(1,3,2)	0	1
mgc_nand(1,4,2)	0	1
mgc_nand(1,5,2)	0	1
mgc_nor(1,2,2)	0	7
mgc_not(1,1)	0	5
mgc_or(1,10,2)	0	2
mgc_or(1,2,1)	0	7
mgc_or(1,3,1)	0	2
mgc_or(1,4,1)	0	4
mgc_or(1,6,2)	0	5
mgc_or(1,8,2)	0	2
mgc_reg_pos(1,0,0,1,1,1,1,4)	0	2
mgc_reg_pos(10,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(11,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(12,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(16,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(21,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(3,0,0,1,1,1,1,4)	0	1
mgc_reg_pos(8,0,0,1,1,1,1,4)	0	37
mgc_reg_pos(9,0,0,1,1,1,1,4)	0	2

TOTAL AREA (After Assignment): 32948.343 6380.000 11556.000

Figure 3.4: Bill of materials after loop unrolling

As it can be seen from the BOM, only one multiplier is being used for now. From the NCSim simulation, the throughput can be seen as coming after every 35 ccs.

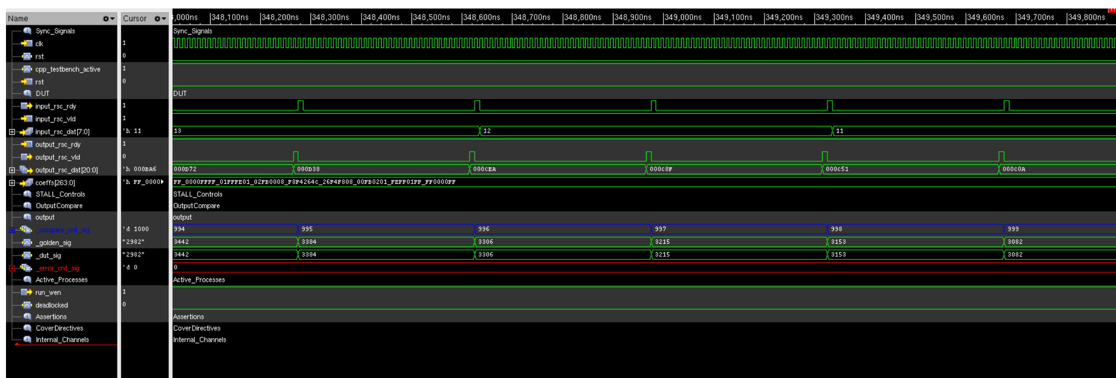


Figure 3.5: Simulation result after loop unrolling

- **Pipelining**: The throughput of the initial design was 68 ccs (680 ns) which is the minimum number of cycles to complete the functionality of a single call of the synthesized function. It means that the design is not reading the next input prior to delivering the previous output i.e. it is not pipelined.
  Even after unrolling the loops, the throughput does not reach 1 which means that next inputs are not being read at every clock cycle. Also, there is some resource sharing of the available multipliers because there are sufficient number of clock cycles in between consecutive input samples. Different multipliers are used at different times for different multiply operations of each unrolled loop iteration. To utilize parallel processing of multiple inputs, the pipelining step is implemented by introducing a design constraint in the source code.

  #pragma hls_pipeline_init_interval 1

  Due to this optimization, the throughput arrives to 1 which means that the design is continuously delivering a result at each clock cycle. But the area has increased by big amount due to extra number of mulitipliers.
  The number of multipliers (33) can be seen from the BOM as indicated in Figure 3.6.

```
Bill Of Materials (Datapath)
  Component Name                                                                    Post Alloc Post Assign
  ------------------------------                                                    ---------- -----------
  [Lib: ccs_ioport]
  ccs_in_wait(1,8)                                                                           1           1
  ccs_out_wait(3,21)                                                                         1           1
  [Lib: starlib_reg_10t_or]
  mgc_add(16,1,16,1,17,6)                                                                   16           1
  mgc_add(16,1,16,1,17,7)                                                                    0          15
  mgc_add(17,0,16,1,17,5)                                                                    0           1
  mgc_add(17,1,17,1,18,5)                                                                    1           0
  mgc_add(17,1,17,1,18,6)                                                                    8           1
  mgc_add(17,1,17,1,18,7)                                                                    0           7
  mgc_add(18,1,18,1,19,5)                                                                    4           0
  mgc_add(18,1,18,1,19,6)                                                                    0           1
  mgc_add(18,1,18,1,19,7)                                                                    0           3
  mgc_add(19,1,19,1,20,5)                                                                    2           1
  mgc_add(19,1,19,1,20,7)                                                                    0           1
  mgc_add(20,1,20,1,21,6)                                                                    1           0
  mgc_add(20,1,20,1,21,7)                                                                    0           1
  mgc_and(1,2,2)                                                                             0           9
  mgc_mul(8,1,8,1,16,7)                                                                     33          33
  mgc_mux(8,1,2,1)                                                                           0           1
  mgc_nor(1,2,2)                                                                             0           4
  mgc_not(1,1)                                                                               0           5
  mgc_or(1,3,1)                                                                              0           2
  mgc_reg_pos(1,0,0,1,1,0,0,4)                                                               0           2
  mgc_reg_pos(1,0,0,1,1,1,1,4)                                                               0           3
  mgc_reg_pos(19,0,0,1,1,1,1,4)                                                              0           2
  mgc_reg_pos(20,0,0,1,1,1,1,4)                                                              0           1
  mgc_reg_pos(21,0,0,1,1,1,1,4)                                                              0           1
  mgc_reg_pos(8,0,0,1,1,1,1,4)                                                               0          33

  TOTAL AREA (After Assignment): 111993.911            5625.000 10270.000
```

```
Area Scores
                     Post-Scheduling    Post-DP & FSM  Post-Assignment
  ----------------- ---------------  ---------------  ---------------
  Total Area Score:  112819.8          114780.1          112075.9
  Total Reg:          14714.9  (13%)    16382.2  (14%)    16382.2  (15%)

  DataPath:          112819.8 (100%)  114698.1 (100%)  111993.9 (100%)
    MUX:                  0.0             131.7   (0%)     131.7   (0%)
    FUNC:             98104.9  (87%)    98104.9  (86%)    95400.7  (85%)
    LOGIC:                0.0             153.3   (0%)     153.3   (0%)
    BUFFER:               0.0               0.0              0.0
    MEM:                  0.0               0.0              0.0
    ROM:                  0.0               0.0              0.0
    REG:              14714.9  (13%)    16308.2  (14%)    16308.2  (15%)
```

Figure 3.6: Bill of materials after unrolling and pipelining

This increase in area as well as the numerous number of multipliers can be seen from the Design Analyzer image as indicated in Figure 3.7.

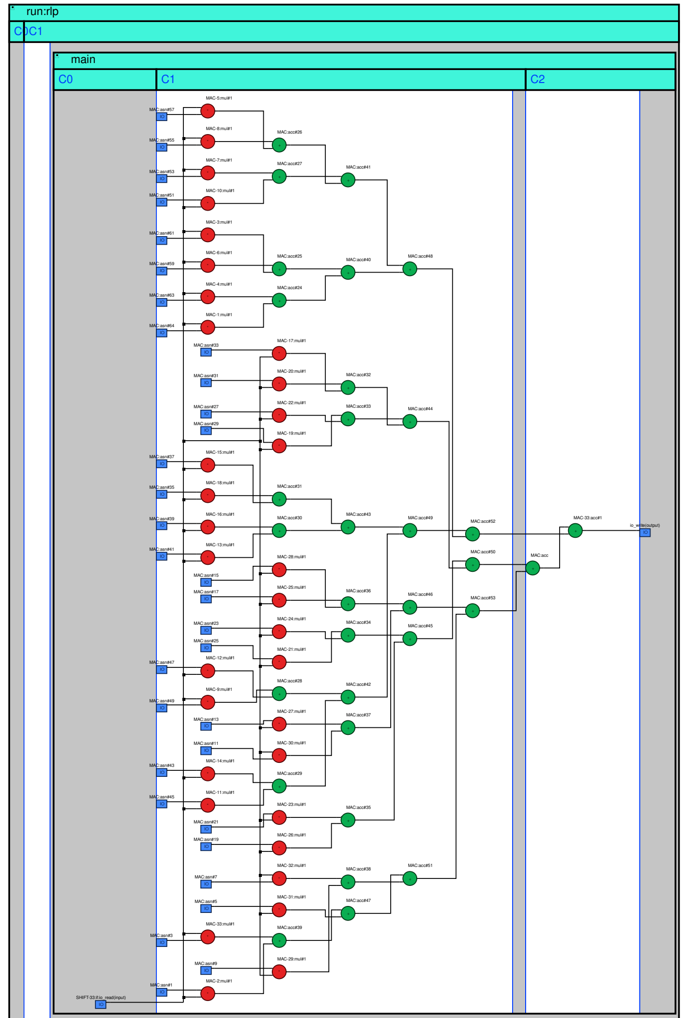


Figure 3.7: Internal architecture after unrolling and pipelining

- **Storing coefficients inside memory**: As mentioned previously, the coefficients are being fed from outside using wire interfaces. A further optimization has been carried out by making use of the fact the coefficient values remain constant in the design so they can also be stored internally in the registers. These coefficients, when synthesized to RTL, offer great dividends in terms of

area savings.

During this process, Catapult performs an automatic optimization such that no multipliers are utilized in the design and all the computations are performed using additions and shift operations. This is depicted graphically in the Design Analyzer, as shown in Figure 3.8 where all the operators are just adders.

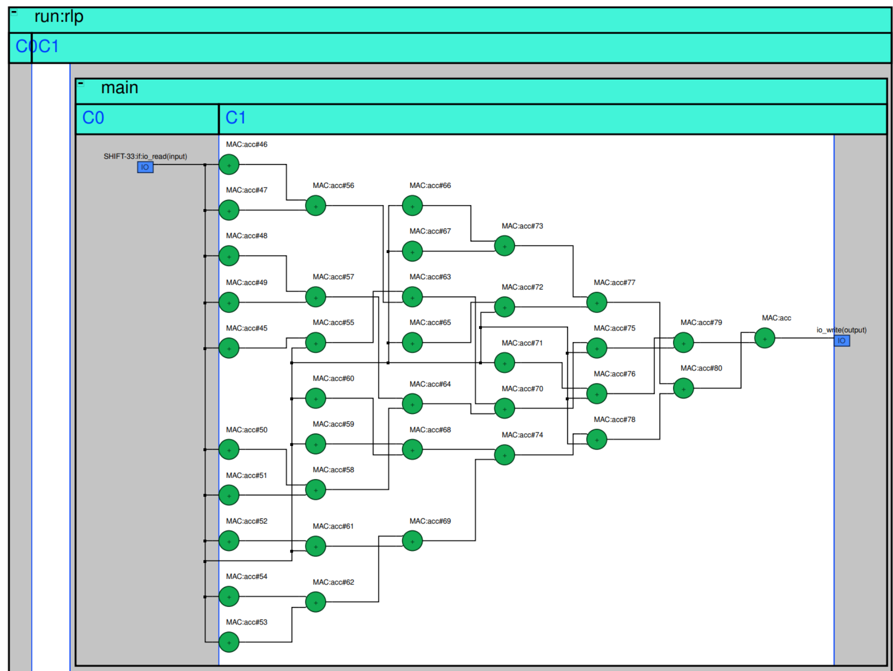
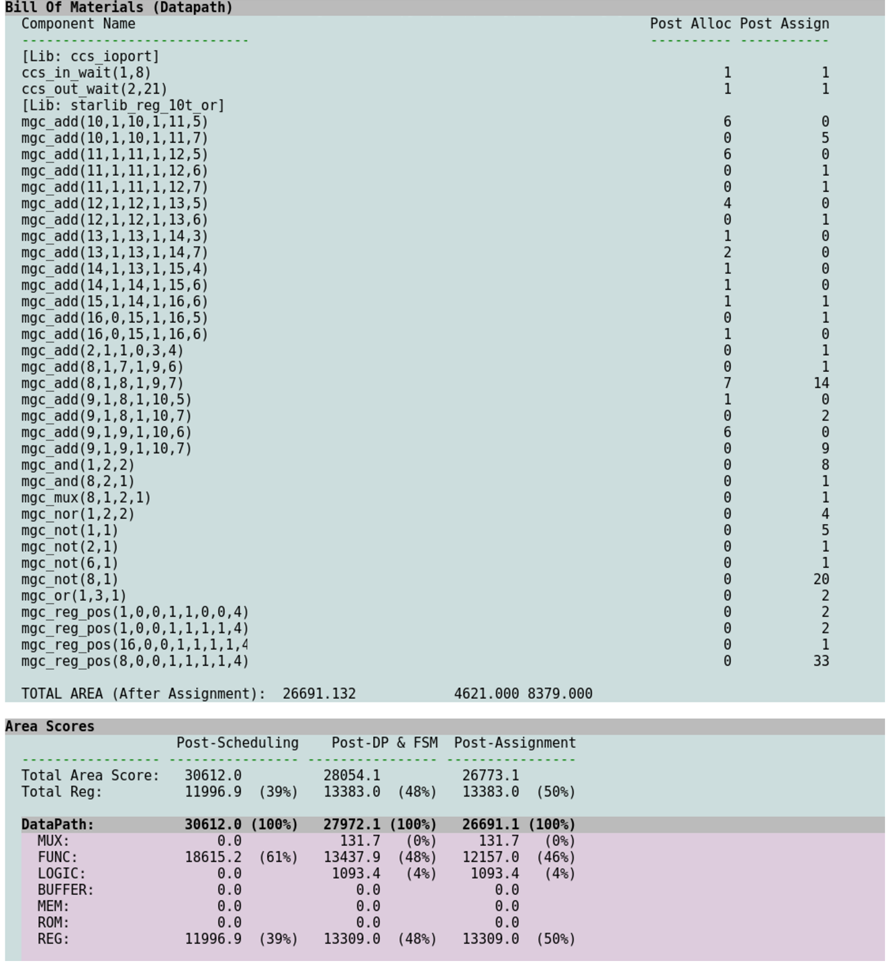


Figure 3.8: Internal architecture after storing coefficients inside registers

Because the values of coefficients are constant, this step can be performed without affecting the results at the output port. The area is drastically reduced to **26773.13** um^2 which is almost 4x times lower than the previous step and even smaller than the original design without any optimizations. The use of adders can also be seen from BOM in Figure 3.9.



Bill Of Materials (Datapath)

Component Name	Post Alloc	Post Assign
[Lib: ccs_ioport]		
ccs_in_wait(1,8)	1	1
ccs_out_wait(2,21)	1	1
[Lib: starlib_reg_10t_or]		
mgc_add(10,1,10,1,11,5)	6	0
mgc_add(10,1,10,1,11,7)	0	5
mgc_add(11,1,11,1,12,5)	6	0
mgc_add(11,1,11,1,12,6)	0	1
mgc_add(11,1,11,1,12,7)	0	1
mgc_add(12,1,12,1,13,5)	4	0
mgc_add(12,1,12,1,13,6)	0	1
mgc_add(13,1,13,1,14,3)	1	0
mgc_add(13,1,13,1,14,7)	2	0
mgc_add(14,1,13,1,15,4)	1	0
mgc_add(14,1,14,1,15,6)	1	0
mgc_add(15,1,14,1,16,6)	1	1
mgc_add(16,0,15,1,16,5)	0	1
mgc_add(16,0,15,1,16,6)	1	0
mgc_add(2,1,1,0,3,4)	0	1
mgc_add(8,1,7,1,9,6)	0	1
mgc_add(8,1,8,1,9,7)	7	14
mgc_add(9,1,8,1,10,5)	1	0
mgc_add(9,1,8,1,10,7)	0	2
mgc_add(9,1,9,1,10,6)	6	0
mgc_add(9,1,9,1,10,7)	0	9
mgc_and(1,2,2)	0	8
mgc_and(8,2,1)	0	1
mgc_mux(8,1,2,1)	0	1
mgc_nor(1,2,2)	0	4
mgc_not(1,1)	0	5
mgc_not(2,1)	0	1
mgc_not(6,1)	0	1
mgc_not(8,1)	0	20
mgc_or(1,3,1)	0	2
mgc_reg_pos(1,0,0,1,1,0,0,4)	0	2
mgc_reg_pos(1,0,0,1,1,1,1,4)	0	2
mgc_reg_pos(16,0,0,1,1,1,1,4	0	1
mgc_reg_pos(8,0,0,1,1,1,1,4)	0	33

TOTAL AREA (After Assignment): 26691.132 4621.000 8379.000

Area Scores

	Post-Scheduling		Post-DP & FSM		Post-Assignment	
Total Area Score:	30612.0		28054.1		26773.1	
Total Reg:	11996.9	(39%)	13383.0	(48%)	13383.0	(50%)
DataPath:	**30612.0**	**(100%)**	**27972.1**	**(100%)**	**26691.1**	**(100%)**
MUX:	0.0		131.7	(0%)	131.7	(0%)
FUNC:	18615.2	(61%)	13437.9	(48%)	12157.0	(46%)
LOGIC:	0.0		1093.4	(4%)	1093.4	(4%)
BUFFER:	0.0		0.0		0.0	
MEM:	0.0		0.0		0.0	
ROM:	0.0		0.0		0.0	
REG:	11996.9	(39%)	13309.0	(48%)	13309.0	(50%)

Figure 3.9: Bill of materials after storing coefficients inside registers

- **Coefficient folding**: If the filter coefficients are symmetric round the center coefficient, it is possible to benefit from this symmetry and decrease the number of multipliers. This is possible because at any single clock cycle, two different inputs are being multiplied with the same coefficient value, but using two multipliers. Instead, as seen from the algebraic equation, it is better to first add the two input samples and then multiply them with the coefficient value. This results in a single multiplier in lieu of two, at a small cost of a

low-area adder.

$$(x[1] * coeff) + (x[2] * coeff) = (x[1] + x[2]) * coeff \tag{3.2}$$

However, this optimization is only useful when coefficient values are being fed from the outside. Because multipliers are being used in such a design, this particular optimization reduces the number of multipliers by half, and is very useful for area savings. In our particular design, this optimization process does not yield better results.

After carrying out the above mentioned optimizations, the result of the newly synthesized filter (showing same output values) shows has an Area = **26773.13** um^2 and throughput = 1.
Figure 3.10 shows a screenshot of Catapult project in which the different versions indicate the different optimization processes as discussed above, and how each one of them is affecting the design performance.

Solution	Latency...	Latency...	Throug...	Throug...	Slack	Total Area
top_template<DATA_I_TYPE,COEFF_TYPE,D... (extract)	66	660.00	68	680.00	4.20	29955.33
top_template<DATA_I_TYPE,COEFF_TYPE,D... (extract)	34	340.00	35	350.00	4.49	24616.02
top_template<DATA_I_TYPE,COEFF_TYPE,D... (extract)	34	340.00	35	350.00	0.57	34294.34
top_template<DATA_I_TYPE,COEFF_TYPE,D... (extract)	2	20.00	1	10.00	0.14	112075.91
top_template<DATA_I_TYPE,COEFF_T... (extract)	**1**	**10.00**	**1**	**10.00**	**0.46**	**26773.13**

Figure 3.10: Design optimizations of FIR filter

As seen from NCSim simulation of Figure 3.11, the output is arriving after each clock cycle and this results in a throughput of 1, as desired.

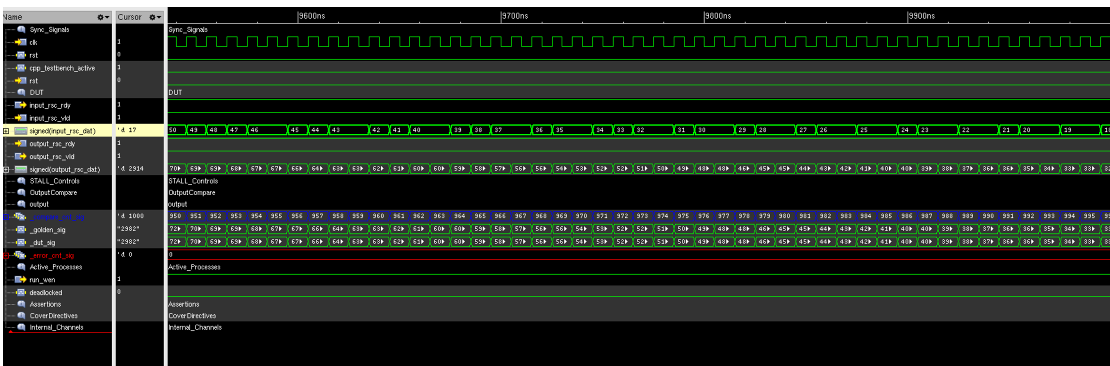


Figure 3.11: Simulation result after optimizations

One additional requirement of this design is the ability to select the the required number of bits at the output (as specified in the parameters above) and most of the time, the MSBs are needed to be kept while discarding the LSBs. This need for selection of *nb_out* number of bits of the output requires manual shifting of the decimal point, which is done easily if all the number are quantized and represented as integer values. Hence, all the inputs, coefficients and outputs have been quantized to integer values. The output results are on the full bit width of the accumulator so that overflow, and hence wrong results, can be avoided.
In order to correctly define the most significant bit (and hence the starting point of the output result), there is a need to know the total number of bits in the accumulator (which has already been explained previously).
This accumulator bit width can be calculated in the Matlab script and then passed as a parameter to the **top.h** header file where it is used by the Catapult source file. It is used to define the **ACC_TYPE** data type which is further used by the accumulator inside the filter source code so that no precision is lost due to overflow. Or it can be calculated as a pre-processor directive inside the C++ source file.
This bit width is also useful in calculating the most significant bit of Catapult design result and it is essential when there is the need to extract a portion of the total bits from the result (the MSBs, in this case). In this design, this extraction of

the required bits is done through a feature available in Catapult known as ***slicing***. It is an in-built method of the form:

slc <W> (int LSB)

where "W" specifies the width of the slice, and "LSB" indicates the starting position of the slice. If W=3 and LSB=2, then Figure 3.12 depicts how the bits are obtained from this slice method.

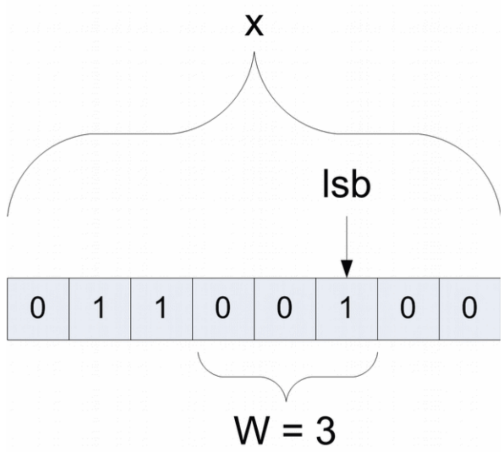


Figure 3.12: Graphical illustration of slice method[17]

Furthermore, half-bit rounding is also needed to be done at the LSB. As mentioned earlier, because we are not using any decimal part in the ac_fixed class here due to conversion to integer values, so automatic rounding using the **AC_RND** parameter is not possible. Hence, this is done manually by selecting *nb_out + 1* number of bits, adding 1 and then right-shifting the result by 1 in order to truncate the LSB. The code snippet for this procedure is shown below and Figure 3.13 depicts it graphically.

```
1    intermediate_out = temp.template slc<NO_OUT_BITS+1>(LSB_START);
2    intermediate_out = intermediate_out + ROUND_ADD;
3    dout = intermediate_out >> ROUND_ADD;
```

ROUND_ADD is the parameter which is obtained depending on the difference between accumulator width and output port width. If the accumulator width is more than the output width, then there is a need to slice the result and hence, half bit rounding is required. In this case, value of ROUND_ADD = 1.
Vice versa, when the accumulator width is less than the output width, then there is no need to slice the result and hence, half bit rounding is not required. In this case, value of ROUND_ADD = 0.

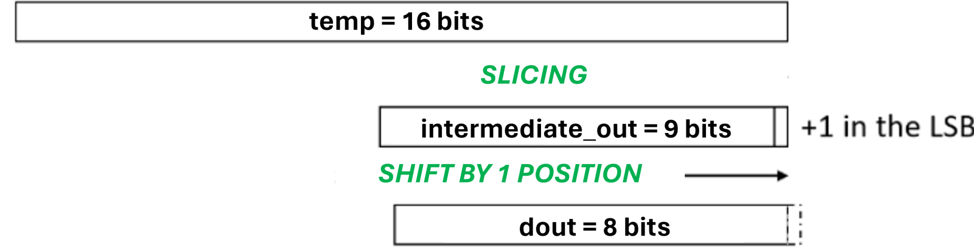


Figure 3.13: Graphical illustration of half bit rounding

3.5 Class-based hierarchical design

Catapult allows creation of hierarchical design modules from C++ classes as well as C++ functions, both of which provide a significant reduction in design times for sub-systems. Addition of hierarchy means partitioning the design into several blocks or processes.
Although both of these these methods offer the same functionality, the coding style is totally different. Also, the class-based hierarchical approach offers some advantages due to which it has been adopted in this thesis work.

One of the biggest advantages of using this coding style is the ability to instantiate multiple instances of the same class which results in increased flexibility. By using the same source code, hardware blocks can be replicated with different parameters which are defined through templates.
Also, data between different blocks can be streamed at different rates by using ac_channels. The data flow will be managed automatically and the sub-blocks will also be able to run in parallel. This parallel computation of the sub-blocks is another advantageous feature of hierarchy and it becomes very easy to meet throughput constraints.

3.5.1 Hierarchy

The project source folder contains 6 files which are used for the FIR design process:

1. **top.h** header file which contains the declarations of the variables, constants, parameters and functions. This file is the only one which needs to be edited for each new design, and this done automatically by the Matlab script.

2. **top_template.h** header file which defines the "top-level" module class and it is parameterized using templates. This block is the one which is connected to the external testbench. It only contains the channels between different classes and hierarchy (no logic). Because it is itself a class, it consists of a constructor (without any arguments) and a public member function ***run*** which is called by the testbench via ac_channels. This member function is used to call / instantiate its sub-block (the fir filter design) by passing inputs and outputs as well as filter coefficients.
   The hierarchical sub-block is declared as a private member.

3. **top_template_inst.cpp** file which is used to explicitly instantiate the top-level module class.

4. **fir_template.h** is the core file which contains the synthesizable description of the FIR filter along with all the loops. It is also parameterized through templates and it is instantiated by the top-level module class which also passes the template parameters. The private data members are the static variables whose values are needed to be retained in between function calls (e.g. input shift register)). The class constructor is used to initialize the values of the private data members. The public member function ***run*** is called by the top-level class and contains the SHIFT loop which acts as a shift register to shift the inputs at each clock cycle.
The FIR filter class contains 2 private member functions *firSymmetricTaps* and *firNonSymmetricTaps*. As the name suggests, the first member function is called when there is a symmetry of the coefficients round the center and the second member function is called when there is no symmetry. This is defined in the top.h header file with the **SYMM_FILT** option which can be changed by the designer in the Matlab script. This if-else logic which determines the function to be called, is written inside the "run" public member function.
The MAC loop is contained inside both these functions and it is fully unrolled to give a throughput of 1. The accumulator is also instantiated inside the functions and it is a temporary register which holds the full precision result during summation of all the taps. At the end, it is typecasted to the output data type.
For Catapult to infer hierarchy of the blocks / classes / functions, we use the following pragma in the code:

hls_design [top][interface]

where the option **'top'** specifies the top level of hierarchy and **'interface'** specifies the sub-block inside the top level.

5. **fir_template_inst.cpp** file which is used to explicitly instantiate the fir_template.h class.

6. **top_tb.cpp** testbench file which is used to read the inputs from an external file, pass the values in the top-level module class, receive the results, compare them and declare the functional test "pass" or "fail" depending upon the comparison.

3.5.2 Template Features

The templatization features of the *top level class* are explained below:

template <class b_intype, class cf_type, class b_otype, class b_acctype, class cf_size>

where class b_intype is the user defined class initialized in the top.h file as

```
typedef ac_fixed<NO_IN_BITS,NO_IN_BITS,true> DATA_I_TYPE;
```

This type is used to denote the input width parameter. As explained above, the values are converted into integers so this type has integer width as the same as total width and it is signed.
Similarly class cf_type is the user defined class which is used to denote the coefficient width parameter and it is initialized as

```
typedef ac_fixed<NO_COEFF_BITS,NO_COEFF_BITS,true> COEFF_TYPE;
```

Class b_otype is the user defined class which is used to denote the output width parameter and it is initialized as

```
typedef ac_fixed<NO_OUT_BITS,NO_OUT_BITS, true> DATA_O_TYPE;
```

And class b_acctype is the user defined class which is used to denote the accumulator width parameter and it is initialized as

```
typedef ac_fixed<NO_ACC_BITS,NO_ACC_BITS, true> ACC_TYPE;
```

The last parameter class cf_size is used to pass down the tap count or the number of coefficients.

The templatization features of the sublock, *fir class* are a little different from the top level and they are explained below:

template <class inType, class coeffType, class outType, class accType, class num_taps, int filt_type, const coeffType coeffs[TAP_COUNT], int instance>

The first four parameters are the bit widths, as mentioned above and class num_taps is the tap count.
int filt_type is equal to the parameter SYMM_FILT which was initialized in the top.h file, and contains information about the filter symmetry. It is used as following in the fir source code:

```
if (filt_type == 0){
    firNonSymmetricTaps(regs, coeffs, data_out);
}
if (filt_type == 1){
    firSymmetricTaps(regs, coeffs, data_out);
}
```

const coeffType coeffs[TAP_COUNT] is the parameter which allows the optimization process related to constant coefficients to happen in a streamlined way. This template parameter is passed down from the top.h file and due to this, there is no need to change the source code of FIR filter, even though every different FIR filter will have different constant coefficients stored inside them, with totally different architectures.
Using this feature, the top level class can instantiate multiple instances of the FIR filters with entirely different values. In previous designs, even if the coefficients were stored in the top level class, they had to be passed as wire inputs to the sub-block of FIR filter. And this procedure resulted in more area. But the ability of Catapult to use template options for entire arrays causes the constant values to be passed as pointers and the software understands the need to initialize the coefficients inside the sub-block, where it offers maximum optimization.
Finally, the **int instance** parameter defines the "number of instance" incase different instances of the same class are initialized. So the first filter will have instance=0, second filter will have instance=1 and so on. This allows Catapult to differentiate between the number of filters it has to synthesize.

Calculation of intermediate bitwidth

As mentioned in **section 3.3**, the intermediate bitwidth of the accumulator / register bitwidth inside the FIR filter is calculated mathematically inside the source files, as a pre-processor directive.
In the C code, a new *typedef* called **int_type_fir** has been defined whose integer width is the same as total bitwidth and it comprises of the minimum number of bits required inside the FIR filter to maintain full precision.
The accumulator inside the filter is defined by this new data type, which can be later sliced according to the required number of output bits. With this implementation,

the **ACC_TYPE** data type is no longer calculated by the Matlab script and it is also not passed as a parameter anymore.
The calculations of this new data type requires understanding of the corresponding values which can be read from the *top.h* file. Alternatively, to make it easier to initiate multiple instances in a filter chain, it is also possible to pass all the required values as parameters during the initialization in the top level file.

Chapter 4

Design of Polyphase decimation filter

4.1 Architecture

Polyphase filters fall under the category of multi-rate filters which use different sample rates for input and output streams. Sample rate conversion is the process of changing the sampling rate of a given input signal. For decimation filters, this is called 'down-sampling', and it is achieved by discarding a certain number of output samples as per the decimation rate. The signal is filtered prior to down-sampling to remove all components which would alias into the new passband. The input is at a higher frequency while the output is taken at a lower frequency. This difference in the sample rate can be used to achieve more area efficient designs by reusing hardware such as adders or multipliers at the lower frequency side. The general principle involves converting an input signal to a lower rate (known as down-sampling), process it by using less number of hardware units and present at the output at the required frequency.

Usually, the process of decimation involves periodically discarding output samples to match the desired reduction in rate. A decimation by a factor of 8 is represented by Figure 4.1.

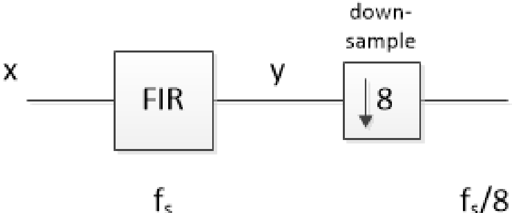


Figure 4.1: Illustration of decimation by 8

A down-sampling rate of 'M' means that every Mth sample is kept and the rest are thrown away. However this method of discarding samples if highly inefficient as it wastes a lot of resources which are otherwise not needed. This problem can be solved by using such a structure which is "enabled" only when the output has to be produced. This splits the filter into M phases (sub-filters) and multiplex the incoming data such that only the relevant input is computed with its coefficient according to the desired output at a certain time (which can be calculated by the mathematical formula). The data arrives at a fast sampling rate, but as soon as asserted to the sub-filter, it is processed at a slower sampling rate.
This way, resources are shared as now they have more number of clock cycles available to process data. This polyphase decomposition originated from the work by Bellanger et al. [19] and plays a fundamental role in multi-rate DSP applications. In this design, the filter is split into its polyphase components such that only those outputs are computed which are needed to be saved, and the separate polyphase components perform their respective calculations.

These individual components receive the samples at a lower rate which is equal to M. This enables to rearrange filtering operation computations, such that computational load per unit time is minimized. Instead of performing a

multiplication per one unit time, now two units of time are available to do the same operation.

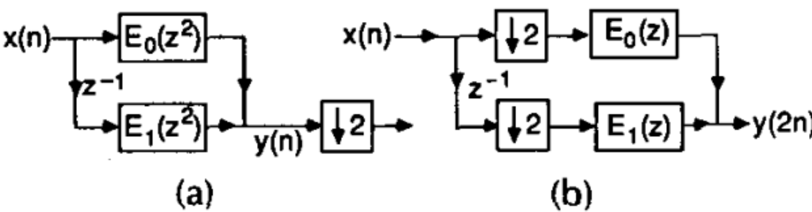


Figure 4.2: Polyphase implementation of decimation filter. (a) Two polyphase components. (b) Equivalent configuration. [20]

4.2 Matlab code and reference model

To check the functionality of a polyphase decimation filter, I have taken a sample FIR filter (as described in the previous section) and only those results are stored which are a multiple of the decimation factor 'M'. This way, the output result mimics the operation of an M-fold polyphase decimation filter. The Matlab model for the FIR is the same as the one described above, having the exact same specifications.

4.3 Mathematical modeling and manual hardware sharing

This type of filter can be implemented in Catapult two ways; adopting a high-level algorithmic approach or a low-level manual hardware sharing approach. Using the first option raises the abstraction level and saves the hardware designer into going into too much detail of the underline architecture whereas the second option requires understanding of how data moves through the tap shift register, and when output data is produced by the filter.

Although the first approach is less time consuming, it does not necessarily provide the best optimized results. Whereas the second approach provides the designer with better control of the underlying architecture and how the resulting RTL will be produced. The second approach has been adopted for this thesis work and to understand it better, have a look at Figure 4.3.

$x_5a_0 + x_4a_1 + x_3a_2 + x_2a_3 + x_1a_4 + x_0a_5$

$x_6a_0 + x_5a_1 + x_4a_2 + x_3a_3 + x_2a_4 + x_1a_5$

$x_7a_0 + x_6a_1 + x_5a_2 + x_4a_3 + x_3a_4 + x_2a_5$

$x_8a_0 + x_7a_1 + x_6a_2 + x_5a_3 + x_4a_4 + x_3a_5$

$x_9a_0 + x_8a_1 + x_7a_2 + x_6a_3 + x_5a_4 + x_4a_5$

$x_{10}a_0 + x_9a_1 + x_8a_2 + x_7a_3 + x_6a_4 + x_5a_5$

$x_{11}a_0 + x_{10}a_1 + x_9a_2 + x_8a_3 + x_7a_4 + x_6a_5$

Figure 4.3: Output equations of order 5 FIR filter

These equations represent the output of normal FIR filter with Order 5. If there is a need to decimate the output of this filter with Rate M=3, we would get only those results at the output which are highlighted in yellow. Adopting the low-level approach requires that the code is written in such a way that only the relevant outputs are calculated and since there are 'M' number of clock cycles in between each output, the hardware resources can be efficiently shared.

First of all, there is a need to calculate how many MACs (Multiply and Accumulate) are needed. The general formula is given by:

$$\#MACs = N/M \tag{4.1}$$

where N = Tap count = Order + 1 and M = Decimation rate. In this case #MACs = 6/3 = 2. Note this is a lot less than the 6 MACs needed for an order 5 filter.

To implement it, there is a need to set some constraints and optimizations at the beginning. These include pipelining the whole design with II = 1 (new input sample at each clock cycle) and unrolling all the loops inside the code.
This design is divided into 'M' phases where each phase is a separate function call to the hardware design (equivalent to sub-filters). There is need to understand how the corresponding input samples are shifted inside the shift register at each clock cycle and how their relevant positions can be used to calculate the correct coefficient index with which the multiplication needs to be done.

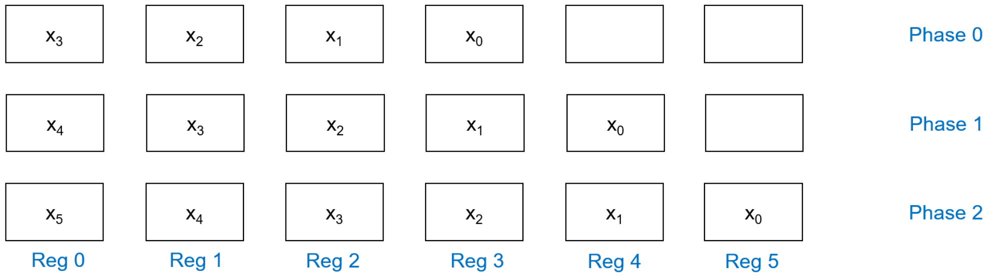


Figure 4.4: Shifting of input samples inside shift register

The last input x5 comes at the last phase so it is calculated at the end, along with x4. (2 MACs only). Similarly, x3 and x2 are calculated in Phase 1; x1 and x0 in Phase 0. Each new phase is a clock cycle and the inputs shift to successive registers. In the whole process, only 2 MACs are used to calculate all the corresponding values. To find the coefficient index and register index as a relation of input position in a register, decimation factor and order of the filter, take a look at figures 4.5 and 4.6 which explain the derivation technique of the formula.

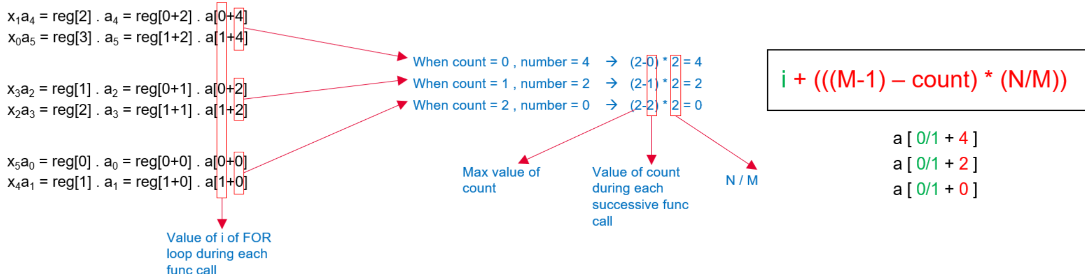


Figure 4.5: Derivation of coefficient index

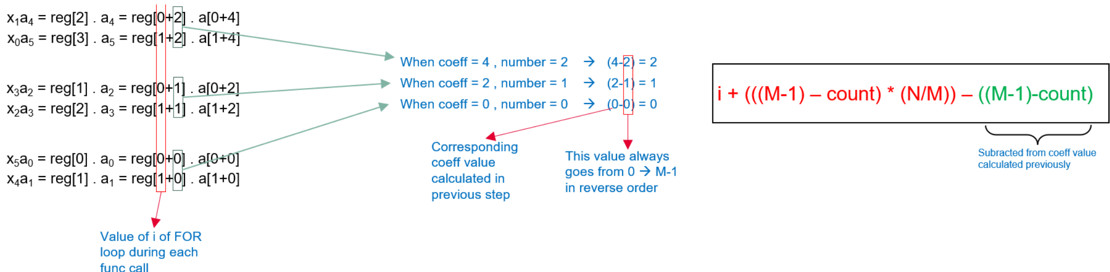


Figure 4.6: Derivation of register index

The formulae derived as shown in the above figures are fully generic and will give the most efficient architectural implementation of the polyphase decimation filter in terms of maximum resource sharing. However, it also introduces corresponding multiplexers which can increase the design area for lower decimation rates. Detailed analysis of this implementation is discussed in the 'results' section of this chapter.

4.4 Synthesizable C++ code (Algorithmic description)

The project is also a class-based design (details mentioned in the previous chapter) and includes six different files;

1. **top.h** header file which contains the declarations of the variables, constants,

parameters and functions. This file is the only one which needs to be edited for each new design, and this done automatically by the Matlab script.

2. **top_poly_template.h** header file which defines the "top-level" module class and it is parameterized using templates. This block is the one which is connected to the external testbench. It only contains the channels between different classes and hierarchy (no logic). Because it is itself a class, it consists of a constructor (without any arguments) and a public member function ***run*** which is called by the testbench via ac_channels. This member function is used to call / instantiate its sub-block (the polyphase filter design).

3. **top_poly_template_inst.cpp** file which is used to explicitly instantiate the top-level module class.

4. **poly_template.h** is the core file which contains the synthesizable description of the polyphase filter along with all the loops. It is also parameterized through templates and it is instantiated by the top-level module class which also passes the template parameters. The private data members are the static variables whose values are needed to be retained in between function calls (e.g. input shift register, accumulator and counter). The class constructor is used to initialize the values of the private data members. The public member function ***run*** is called by the top-level class and contains the MAC and SHIFT loops.

5. **poly_template_inst.cpp** file which is used to explicitly instantiate the poly_template.h class.

6. **top_tb.cpp** testbench file which is used to read the inputs from an external file, pass the values in the top-level module class, receive the results, compare them and declare the functional test "pass" or "fail" depending upon the comparison.

The testbench is used to read the inputs from an external file, pass the values in the core file, receive the results, compare them and write them to an external file. As compared to the FIR filter code, the major change implemented is in the MAC loop which takes into the account the above derived formula for calculation of the results. The temporary results after each function call are stored in an accumulator and when the phase count reaches the decimation factor, it sends the value as the output and resets itself back to zero. Initializing it as a static variable allows it to retain its value in between function calls. The code snippet for this formula being used inside the polyphase filter source code of HLS design is reported below.

```
1    #pragma hls_unroll yes
2    MAC: for (unsigned j=0; j<MACS; j++)
3    {
4        acc += coeffs_regs[j + (((DEC_FACTOR-1)-cnt) * MACS)] *
     data_i_regs[j + (((DEC_FACTOR-1)-cnt) * MACS) - ((DEC_FACTOR-1)-
     cnt)];
5    }
```

4.5 Architectural optimizations

For this design purpose, the following optimizations have already been incorporated in order to avoid repetition of the optimization process:

1. Pipelining.
2. Loop unrolling.
3. Use of class-based hierarchy.

However, one of the circumstances for which the previous design is not optimized for is when tap count of the filter is not an integer multiple of decimation rate (N /

M != 0). If the combination of rate and tap count results in a sub-filter which is not fully populated with coefficients, the reorganization of the filter coefficients results in a change in the phase response of the filter. To avoid this problem, **coefficient padding** is done in which the empty slots of the sub-filters are filled with zero coefficients. This way, the general formula for polyphase implementation holds true without affecting the filter response of the filter. To introduce coefficient padding, the extra zeros are added into the design in place of extra coefficients.
The following code snippet from top.h file shows hows the number of MACs is calculated in the polyphase design, depending on the ratio between tap count and decimation rate. Afterwards, the total number of coefficients are again calculated so that padding of zeroes can be done.

```
1    const unsigned MACS = (TAP_COUNT % DEC_FACTOR==0) ? TAP_COUNT /
   DEC_FACTOR : (TAP_COUNT / DEC_FACTOR) + 1;
2    const unsigned TOTAL_COEFFS = DEC_FACTOR * MACS;
```

The method of implementing the operation of coefficient padding depends on how coefficients are used in the design. The following two ways explain the details:

1. **Coefficients fed as direct inputs:** The addition of extra zeros in this case is done from the testbench. The following code in the testbench file allows for reading of the coefficients from an external file. The coefficient values are mapped to a "Direct Input" constraint. Doing so makes it so that the coefficient feed is a set of wires, with no pipeline registers or handshaking logic. The absence of pipeline registers and handshaking logic results in lower power consumption and reduced area.

```
1    unsigned i;
2    FILE *fp_in;
3    COEFF_READ_LOOP: for (unsigned s=0; s<NO_FILTERS; s++)
4    {
5     i=0;
6     fp_in = fopen(coefficients_file.c_str(), "r");
7     fscanf(fp_in, "%lf", &double_coeff_i);
8     COEFF_TYPE coeff_i = double_coeff_i;
          // Typecast to COEFF_TYPE
9     do{
10    filterCoeffs_reg[i] = coeff_i;
11    fscanf(fp_in, "%lf", &double_coeff_i);
12    coeff_i = double_coeff_i;
13    i++;
14    if (i>TAP_COUNT) {
15         cout<<" -Too many coefficients found in txt file.
   Expected are "<<TAP_COUNT<< " values."<<endl;
16    }
17    cout<<" -Read in "<<i<< " coefficients from "<<
   coefficients_file<<" "<<endl;
18     }while(!feof(fp_in));
19    fclose(fp_in);
20    }
21    while (i < TOTAL_COEFFS){
          // Adding Zero coefficients at the end of the list
22        filterCoeffs_reg[i] = zero_value;
23        i++;
24        cout<<" -Added zero value coefficient at position "<<i<<
   " "<<endl;
25    }
26    cout << endl;
```

The 'while' loop at the end is responsible for adding zero values as the extra coefficients which are introduced as a result of coefficient padding.

2. **Coefficients stored inside memory:** In case of design optimization where the coefficients are stored internally in the memory, it is not possible to add the zeros from the outside. Instead there has to be a way to initialize the correct number of total coefficients inside the memory (actual number + extra zeroes) and this needs to be generic, so that different specifications always yield the correct calculations.
   For this purpose, we again make the use of pre-processor directives to 'internally' calculate the total number of coefficients required. Then, a new array (internal registers) is initialized with a value of '0' in all the registers. Finally, this array is overwritten with the values of *'N'* filter coefficients where *'N'* is the original tap count. The remaining registers will retain the value of zero, which is equivalent to coefficient padding implementation.

```
1  ac::init_array <AC_VAL_0> (coeffs, total_coeffs_type::width);
2
3  // Automatic coefficient padding by zeroes incase tap count
   is not a multiple of decimation factor
4  for (int j = 0; j <= filt_order; j++) {
5    coeffs[j] = coeffs_regs_temp[j];
6  }
```

4.6 Results

4.6.1 Order:5 , Decimation:2 & 3

Figure 4.7 shows that 2 multipliers are used inside a polyphase filter with *Order = 5* and *decimation rate = 3*. As explained previously, this is due to the fact that six computations are needed to be performed over the course of three clock cycles. If one multiplier takes 1 cc to perform 1 computation, then 2 multipliers can perform six computations in three clock cycles.

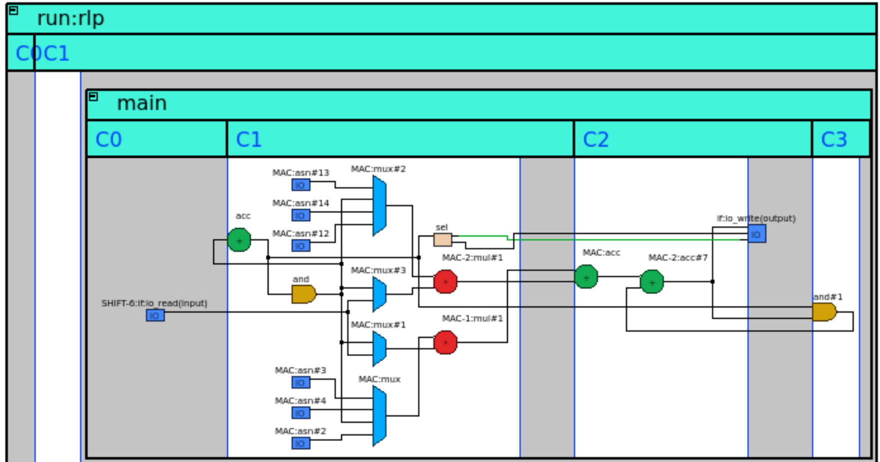


Figure 4.7: 2 multipliers used for 6 taps

Figure 4.8 shows the simulation result in which it can be seen that the output throughput is every 3 clock cycles.

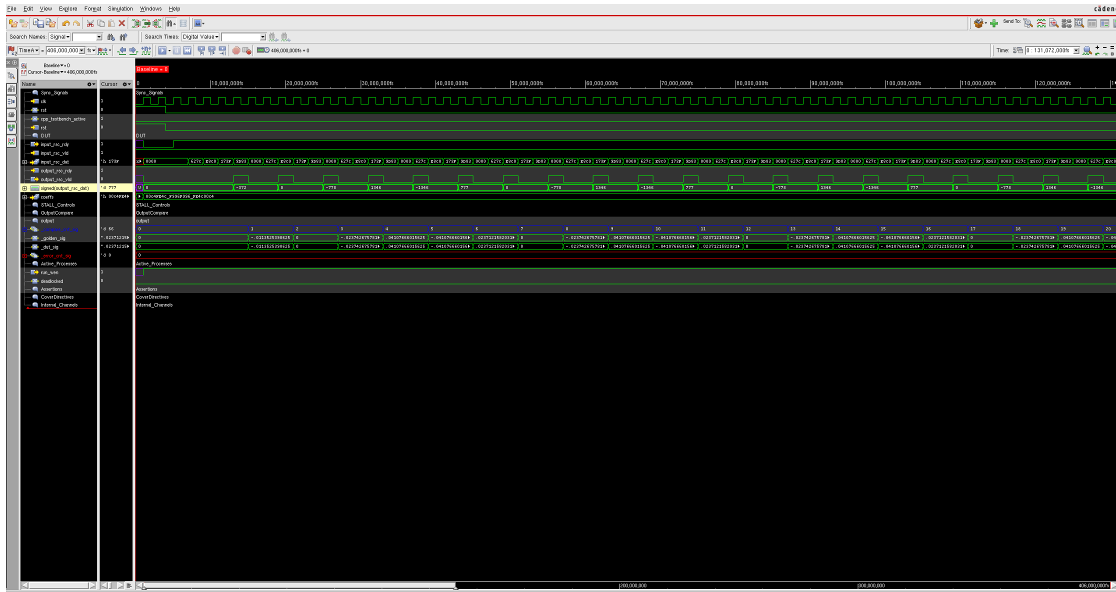


Figure 4.8: Simulation on NCSim showing output with decimation rate of 3

```
 -Read in 1 coefficients from ../../matlab/coefficients_POLY_INT.txt
 -Read in 2 coefficients from ../../matlab/coefficients_POLY_INT.txt
 -Read in 3 coefficients from ../../matlab/coefficients_POLY_INT.txt
 -Read in 4 coefficients from ../../matlab/coefficients_POLY_INT.txt
 -Read in 5 coefficients from ../../matlab/coefficients_POLY_INT.txt
 -Read in 6 coefficients from ../../matlab/coefficients_POLY_INT.txt

SCVerify intercepting C++ function 'top_poly_template<DATA_I_TYPE, COEFF_TYPE, DATA_O_TYPE, ACC_TYPE>::run'
                       DUT instance '0x2b5eb6172930'

Info: HW reset: TLS_rst active @ 0 s


 -Read in 300 input samples.
 -300 input samples are being streamed to DUT.
 -DUT has returned with 100 y samples in output channel.
 -Output file for storing results has been created.
 -Reference file containing MATLAB results has been opened.

*********************************************** TEST PASS ********************************************

 -All results have been saved into the output file
 -End of testbench


Info: Execution of user-supplied C++ testbench 'main()' has completed with exit code = 0

Info: Collecting data completed
   captured 300 values of data_i
   captured 300 values of coeffs_regs
   captured 100 values of data_o

Info: scverify_top.user_tb: Simulation completed

Checking results
'data_o'
   capture count        = 100
   comparison count     = 100
   ignore count         = 0
   error count          = 0
   stuck in dut fifo    = 0
   stuck in golden fifo = 0
```

Figure 4.9: Result of HLS generated design as compared to Matlab results

Now, if we decrease the decimation rate of the same filter, it would mean that

the output needs to be generated at a quicker rate. And hence, more hardware resources are needed. Figure 4.10 shows the area result of a filter with *Order = 5* and *decimation rate = 2*. As expected, now that the output has to be delivered in a lesser number of clock cycles, more multipliers and multiplexers are needed to implement the functionality and there is an increase in area. Figure 4.11 shows the simulation result in which it can be seen that the output throughput is every 2 clock cycles.

Solution	Latency...	Latency...	Throug...	Throug...	Slack	Total Area	
top_poly_template<DATA_I_TYPE,... (extract)	1	10.00	1	10.00	2.13	10555.76	DEC RATE = 3
top_poly_template<DATA_I_T... (extract)	**1**	**10.00**	**1**	**10.00**	**0.29**	**13511.64**	DEC RATE = 2

Figure 4.10: Comparison of areas for two different decimation rates

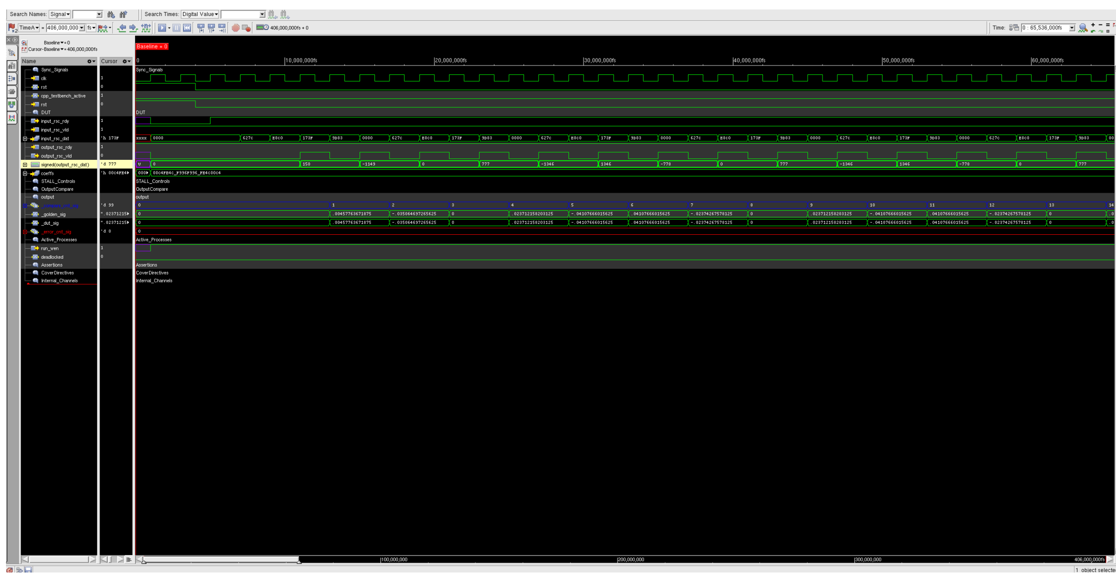


Figure 4.11: Simulation on NCSim showing output with decimation rate of 2

The number of multipliers and multiplexers are shown in the Design Analyzer result in in Figure 4.12, which are more than those in the previous design.

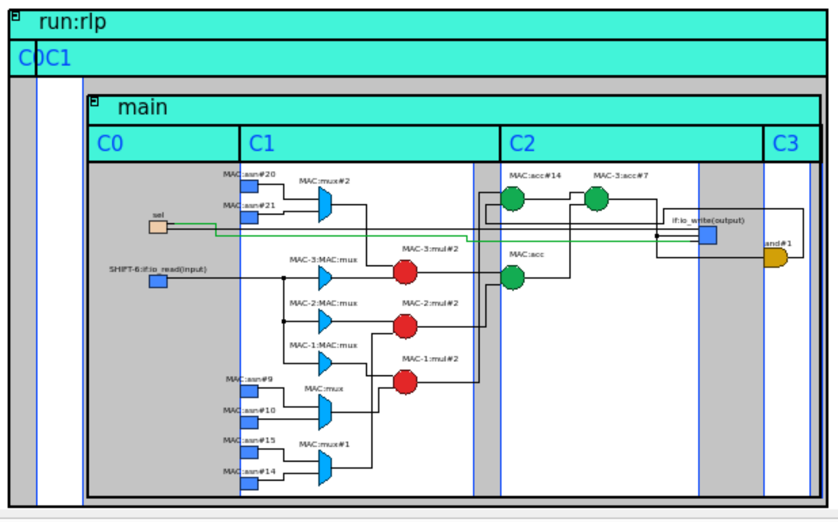


Figure 4.12: 3 multipliers used for 6 taps

The results indicate that this polyphase implementation is highly effective in resource sharing and is able to provide big dividends in area savings. Normally, the same filter would require 6 multipliers and an area of **20017.29** um^2. Increasing the decimation rate allows the same multipliers to be re-used in between consecutive clock cycles.

4.6.2 Order:123 , Decimation:2 & 62

A polyphase filter of bigger order (123) is compared with an RTL-coded implementation of the same specifications. To better analyze the effect of different decimation rates on the area comparisons, two different rates are chosen (2 and 62).

Both the polyphase filter designs are the result of automated design flows which aim to build a hardware design when provided with any set of filter specifications. Although functionally equivalent, these two filters are different from the design implementation point of view and how the datapath has been arranged inside the core.

Figure 4.13 shows the area comparisons of the two polyphase filters having different decimation rates; for the HLS design flow and the RTL design flow.

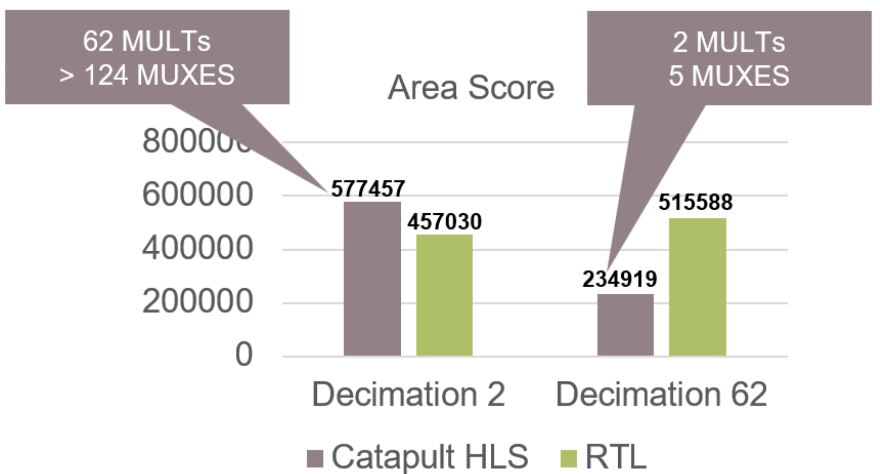


Figure 4.13: Area comparison: Polyphase filter of order 123 designed using HLS flow and RTL flow

As mentioned previously, our filter made with the HLS automated design flow utilizes resource sharing for effective reuse of hardware. Each multiplier has two multiplexers which feed the operands to it. The first multiplexer selects the input value from the shift register and the second multiplexer selects the value from the coefficients. For our HLS design flow, the sharing of resources and their corresponding effect can be summarized as:

1. A smaller decimation rate means a higher number of multipliers and multiplexers.
2. A higher decimation rate means a smaller number of multipliers and multiplexers.

To understand the big difference in the areas of the filters designed by the HLS flow using the generic algorithm, figure 4.14 and figure 4.15 show the area scores of

our two polyphase filters. Figure on the left has the smaller decimation of 2 and the figure on the right has the larger decimation of 64. The differences in the area of the **FUNC** part and **MUX** are clearly visible, giving clear insight to the implementation details of using a generic algorithm to design filters with different specifications.

Area Scores	Post-Scheduling		Post-DP & FSM		Post-Assignment	
Total Area Score:	784695.0		778848.0		767984.5	
Total Reg:	122171.1	(16%)	125056.8	(16%)	125056.8	(16%)
DataPath:	**784695.0**	**(100%)**	**778766.0**	**(100%)**	**767902.5**	**(100%)**
MUX:	87528.1	(11%)	89584.3	(12%)	89012.1	(12%)
FUNC:	574712.4	(73%)	563017.3	(72%)	552702.7	(72%)
LOGIC:	283.5	(0%)	1181.6	(0%)	1204.9	(0%)
BUFFER:	0.0		0.0		0.0	
MEM:	0.0		0.0		0.0	
ROM:	0.0		0.0		0.0	
REG:	122171.1	(16%)	124982.8	(16%)	124982.8	(16%)
FSM:	**0.0**		**82.0**	**(0%)**	**82.0**	**(0%)**
FSM-REG:	0.0		74.0	(90%)	74.0	(90%)
FSM-COMB:	0.0		8.0	(10%)	8.0	(10%)

Figure 4.14: Order 123 and Decimation 2

Area Scores	Post-Scheduling		Post-DP & FSM		Post-Assignment	
Total Area Score:	159533.5		159810.3		162366.7	
Total Reg:	106987.5	(67%)	109826.4	(69%)	109826.4	(68%)
DataPath:	**159533.5**	**(100%)**	**159728.3**	**(100%)**	**162284.7**	**(100%)**
MUX:	18961.5	(12%)	19257.9	(12%)	19257.9	(12%)
FUNC:	33055.2	(21%)	29717.2	(19%)	32236.8	(20%)
LOGIC:	529.2	(0%)	1000.7	(1%)	1037.6	(1%)
BUFFER:	0.0		0.0		0.0	
MEM:	0.0		0.0		0.0	
ROM:	0.0		0.0		0.0	
REG:	106987.5	(67%)	109752.4	(69%)	109752.4	(68%)
FSM:	**0.0**		**82.0**	**(0%)**	**82.0**	**(0%)**
FSM-REG:	0.0		74.0	(90%)	74.0	(90%)
FSM-COMB:	0.0		8.0	(10%)	8.0	(10%)

Figure 4.15: Order 123 and Decimation 62

4.6.3 Order:28 , Decimation:8

The last comparison is done against a polyphase filter provided by Infineon, of order 28 and decimation 8. The input bitwidth is 2 and the output bitwidth is 13. Figure 4.16 shows the graph, correlating the area and static power consumption. The HLS design shows better results due to resource sharing. The design provided by Infineon, although functionally equivalent, is having a different architecture and does not utilize reuse of hardware resources. Hence, the difference in the overall area and the power consumption.

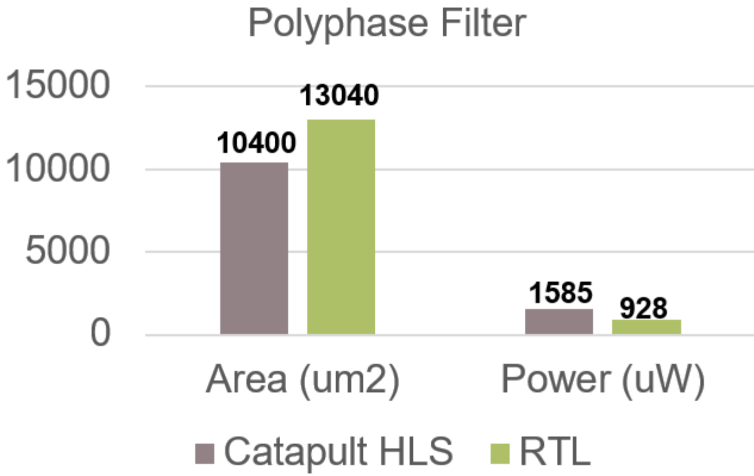


Figure 4.16: Area comparison of order 28 and decimation 8 filter

As shown from figure 4.17, this filter reuses the hardware multipliers, which results in a total of only 4 MULTs instead of 29.

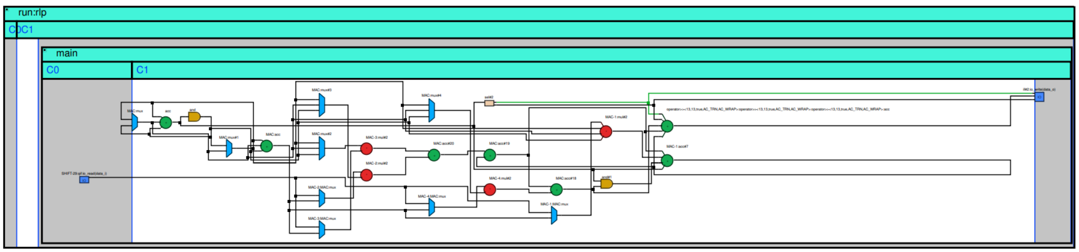


Figure 4.17: Resource sharing inside polyphase filter

The final design of this polyphase filter was obtained after implementing one small optimization which was able to save an area of **600** um^2 as well as improve the slack. Instead of using 16 bits for the accumulator bitwidth, we use 13 bits. This is possible only because we know the values of the coefficients beforehand and that can be used to calculate the maximum possible result that can be obtained.

This small difference in the final area can be seen from the Catapult synthesis flow, as depicted in figure 4.18.

Solution	Lat...	Laten...	Thr...	Throu...	Slack	Total Area
top_poly_t... (extract)	1	10.00	1	10.00	0.78	9756.70
top_poly... (extract)	**1**	**10.00**	**1**	**10.00**	**2.12**	**9182.78**

Figure 4.18: Area saving after optimization

Chapter 5

Design of CIC decimation filter

5.1 Architecture and frequency response

CIC (Cascaded Integrator Comb) decimation filters also fall under the category of multi-rate filters and are used in sample rate conversion. They fall under the class of narrowband low-pass filters and they are highly computation-efficient[21]. Although they can be used both for the purpose of decimation (down-sampling) and interpolation (up-sampling), this thesis work deals with only the design of CIC decimation filter.
One of the biggest advantages of CIC filters is the absence of arithmetic multipliers in their structure. Because they do not need any coefficients, there are no multiplication operations and this also saves area in terms of less data storage. The arithmetic operations only consist of addition and subtraction.[22]

The feedback portion (on the left of decimator, operating a higher sample rate) is called the 'integrator' section and the feedforward portion (on the right of

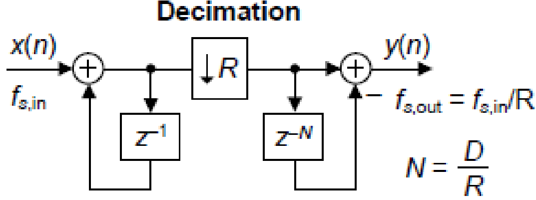


Figure 5.1: CIC decimation filter of order 1

decimator, operating at lower sample rate) is known as the 'comb' section. This was one of the key features of CIC filters introduced to the world by Hogenauer in 1981[22]. The integrator can be viewed as an accumulator whereas the comb stage is used to subtract a delayed input sample from the current input sample. The parameter 'N' is known as the 'differential delay' and it is typically kept at 1 or 2 (in my designs, I have kept it constant at 1).

To provide full accuracy of results, an important thing which has to be kept in mind are the register bit widths. Because the integrator stage is accumulating results at each stage and with new input samples, there occurs an arithmetic overflow. However, if the following two conditions are met, this overflow is of no consequence:

- Each stage uses two's complement arithmetic (non-saturating).[23]
- The maximum value expected at output of a stage is less than or equal to the range of stage's number system.

To avoid overflow, the following formula is implemented:

$$Register_bit_width = x(n) + ceil[M.log2(N.R)] \tag{5.1}$$

Where x(n) are number of bits of input samples, M is the filter order, N is the differential delay and R is the decimation rate. This implies that the input samples are first sign extended to the higher number of bits before entering the integrator

section.

5.2 Matlab code and reference model

A sample code in a script is written in Matlab to get the results of CIC decimation filter. The inputs are passed through a Matlab system object (dsp.CICDecimator) which is obtained by passing the parameters of decimation rate, differential delay and filter order. The outputs are then stored in a file. The specifications of the filter in order to design a generic source code are given below:

- Filter order ***M = 8***
- Number of bits at input ***nb_in = 12***
- Number of bits at output ***nb_out = 12***
- Decimation rate ***R = 4***

After running the Matlab script, the results of this particular CIC filter are obtained and stored in a file which will be used as golden reference model for the Catapult design. It is to be noted that all the inputs and outputs have been quantized to integer values. These results are on the full bit width of the accumulator so that overflow, and hence wrong results, can be avoided.
Similar to the previous two designs, here also one of the requirements of this design is the ability to select the required number of bits at the output (as specified in the parameters above) and most of the time, the MSBs are needed to be kept while discarding the LSBs. This need for selection of *nb_out* number of bits of the output requires manual shifting of the decimal point, which is implemented in the Matlab script.

In order to correctly define the most significant bit (and hence the starting point

of the output result), there is a need to know the total number of bits in the accumulator. This is calculated from equation (5.1). This accumulator bit width is then also passed as a parameter to the **top.h** header file where it is used by the Catapult source file for slicing, as explained in the previous chapters.
At the end, the Matlab script is used to automatically edit the **top.h** header file which contains all the parameters and information used by the CIC decimator source files for the design process.

5.3 Algorithmic description in C++

The project is also a class-based design (details mentioned in the previous chapters) and includes four different files:

1. **top.h** header file which contains the declarations of the variables, constants, parameters and functions. This file is the only one which needs to be edited for each new design, and this done automatically by the Matlab script.

2. **top_cic_template.h** header file which defines the "top-level" module class and it is parameterized using templates. This block is the one which is connected to the external testbench. It only contains the channels between different classes and hierarchy (no logic). Because it is itself a class, it consists of a constructor (without any arguments) and a public member function ***run*** which is called by the testbench via ac_channels. This member function is used to call / instantiate its two sub-blocks (integrator and comb classes) which are connected via an ac_channel.

3. **cic_decimator_template.h** is the core file which contains the synthesizable description of the CIC decimation filter and it further contains two classes:

   - **cic_integrator_stage** class which contains the source code for integrator

part of the CIC filter. It is the high-frequency part of the filter which accepts the input data samples at each clock cycle and consists of the chain of adders and feed-back paths. It sends its output only when the counter reaches the value of decimation factor.

- **cic__comb__stage** class which contains the source code for comb part of the CIC filter. It is the low-frequency part of the filter which accepts the input data samples from the integrator at slower rate (equal to decimation rate) and consists of the chain of subtractors and feed-forward paths.

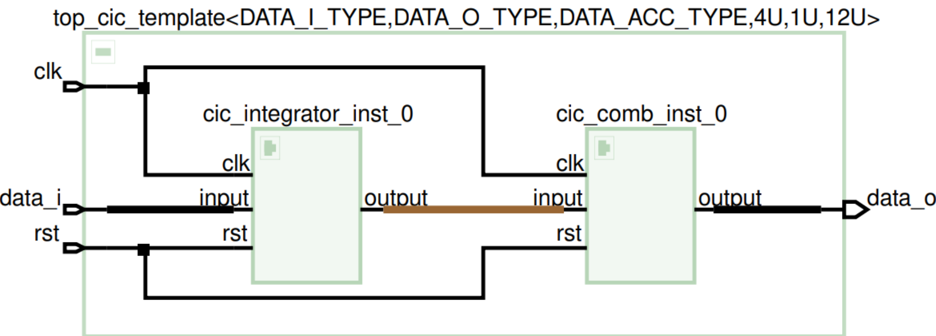


Figure 5.2: Design Analyzer view of CIC filter composed of separate sub-blocks

Both these classes are parameterized through templates and they are instantiated by the top-level module class which also passes the template parameters. The private data members are the static variables whose values are needed to be retained in between function calls (e.g. shift registers and counter). The class constructor is used to initialize the values of the private data members. The public member functions ***run*** of both the classes are called by the top-level class.

4. **top__tb.cpp** testbench file which is used to read the inputs from an external file, pass the values in the top-level module class, receive the results, compare

them and declare the functional test "pass" or "fail" depending upon the comparison.

The parameter "total number of bits in the accumulator" **NO_ACC_BITS**, as mentioned in the previous section, is calculated by the Matlab script and passed to the top.h header file. Here it is used to define the **DATA_ACC_TYPE** data type which is further used by the accumulator inside the filter source code so that no precision is lost due to overflow.
Similar to the FIR and polyphase filter, another useful way to determine the intermediate bitwidth of the accumulator / register bitwidth inside the CIC filter is to calculate it mathematically inside the source files, as a pre-processor directive. In the C code, this is used in the following way:

```
1  //===========================================================================
2  // Struct: power
3  // Description: templatized struct for computing power of a number
4  //---------------------------------------------------------------------------
5  template <int base, int expon>
6  struct power {
7    enum { value = base * power<base, expon - 1>::value };
8  };
9
10 //===========================================================================
11 // Struct: power
12 // Description: templatized struct for computing power of a number
     with a
13 // zero component
14 //---------------------------------------------------------------------------
15 template <int base>
16 struct power<base, 0> {
17   enum { value = 1 };
```

```
18 };
19
20 //===========================================================================
21 // Struct: find_inter_type_cic_dec
22 // Description: provides parameterized bitwidths to ensure a lossless
23 // intermediate type for the CIC filter. W, I and S are word width,
24 // integer width and signedness of the input.
25 //---------------------------------------------------------------------------
26 template <class data_i_type, int R, int M, int N>
27 struct find_inter_type_cic {
28   enum {
29     W    = data_i_type::width,
30     I    = data_i_type::i_width,
31     S    = data_i_type::sign,
32     outF = W - I,
33     outW = ac::log2_ceil<power<R, N>::value *power<M, N>::value >::val
       + W + int(!S),    // ...(i)
34     outI = outW - outF
35   };
36   typedef ac_fixed<outW, outI, true> int_type_cic;
37 };
```

The **int_type_cic** datatype is the one which defines the internal registers width. All incoming data is first sign extended to this type to avoid wrong results. Manual slicing of the output result is also done in the source code, as with previous designs. Furthermore, half-bit rounding is also needed to be done at the LSB. As mentioned earlier, because we are not using any decimal part in the ac_fixed class here, so automatic rounding using the **AC_RND** parameter is not possible. Hence, this is done manually by selecting $nb_out + 1$ number of bits, adding 1 and then right-shifting the result by 1 in order to truncate the LSB.
Implementation details of manual slicing as well as half-bit rounding have been

explained in chapter 1.

5.4 Architectural optimizations

For this design purpose, the following optimizations have already been incorporated in order to avoid repetition of the optimization process:

1. Pipelining.

2. Loop unrolling.

3. Use of class-based hierarchy.

Apart from the above 3 optimizations, 2 more optimizations have also been incorporated, which are explained below:

1. **Resource sharing in the comb part:** The *cic_comb_stage* class takes the decimation rate as input parameter. Firstly, by setting the initiation interval of the pipeline design equal to decimation rate, Catapult makes use of resource sharing by realizing the extra number of clock cycles available for computation. Hence, same adders are re-used in between the "free" clock cycles. This is realized by:

   template <class inType, class outType, int dec, int instance>
   #pragma hls_pipeline_init_interval dec

   For example, normally a CIC filter with order = 8 will require 8 adders (subtractors) in the comb part. However, with a decimation rate of 4, only 2 adders are used since both of them are working each clock cycle to produce a total of 8 computations as originally required.
   This difference in the number of adders inside the comb part of the filter can

be be seen from Figure 5.3 and Figure 5.4, in front of **Quantity** caption.

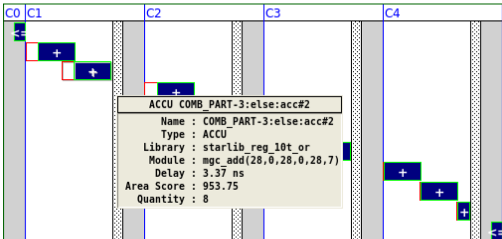


Figure 5.3: Comb part before optimization

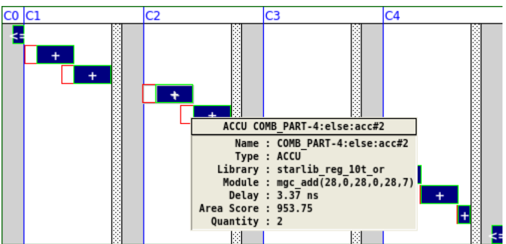


Figure 5.4: Comb part after optimization

However, it must be noted that this is not true in all the cases. Mostly when **Decimation rate < Filter order**, we see better results, but in the opposite scenario, it is also possible to obtain worse area results. There are two primary reasons for this. One is that Catapult adds extra registers in the output FIFO queue, which are not needed. And secondly, the re-use of adders results in introduction of some multiplexers which sometimes increases the overall area, despite using less adders. Therefore, it is recommended to check the final area results after making use of this resource sharing option in order to arrive to the best possible implementation.

2. **Removing the FIFO queue between integrator and comb part:** As mentioned previously, the CIC design is split into two separate classes which are connected by ac_channel. By default, this introduces a pipe which is basically a FIFO queue, which is not needed in this case. For better results, this internal pipe connecting the two classes can be removed, without changing the functionality of the filter. This step can be done in one of two ways:

   - By using the GUI inside Catapult, go to "Architecture" panel from the left, expand the **interface** drop down option, select the given pipe and

set the FIFO depth = 0. By default, the value is set to **-1** which means that there is no upper limit.

- By using a design constraint in the source code, which is the method used here in our design. This is done by adding the following statement above the defined ac_channel inside the file **top_cic_template.h**.

#pragma hls_fifo_depth 0

Figures 5.5 and figure 5.6 show the block diagram of this optimization.

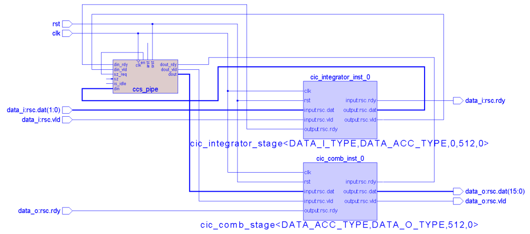


Figure 5.5: Block diagram view: With pipe

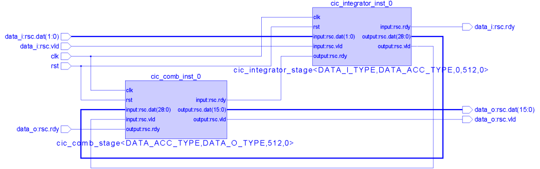


Figure 5.6: Block diagram view: No pipe

These optimizations result in some additional area savings. The non optimized design has Area = **54075.66** um^2 and the optimized design has Area = **47603.16** um^2, which can be seen from Figure 5.7.

Solution	Late...	Laten...	Thr...	Thr...	Slack	Total Area
top_cic... (extract)	**5**	**50.00**	**1**	**10.00**	**2.72**	**54075.66**
top_cic_... (extract)	5	50.00	4	40.00	2.35	47603.16

Figure 5.7: Comparison of areas for CIC filter architectures

From Figure 5.8, it can be seen the output is still being produced after the given decimation rate of 4, after implementing all the optimizations.

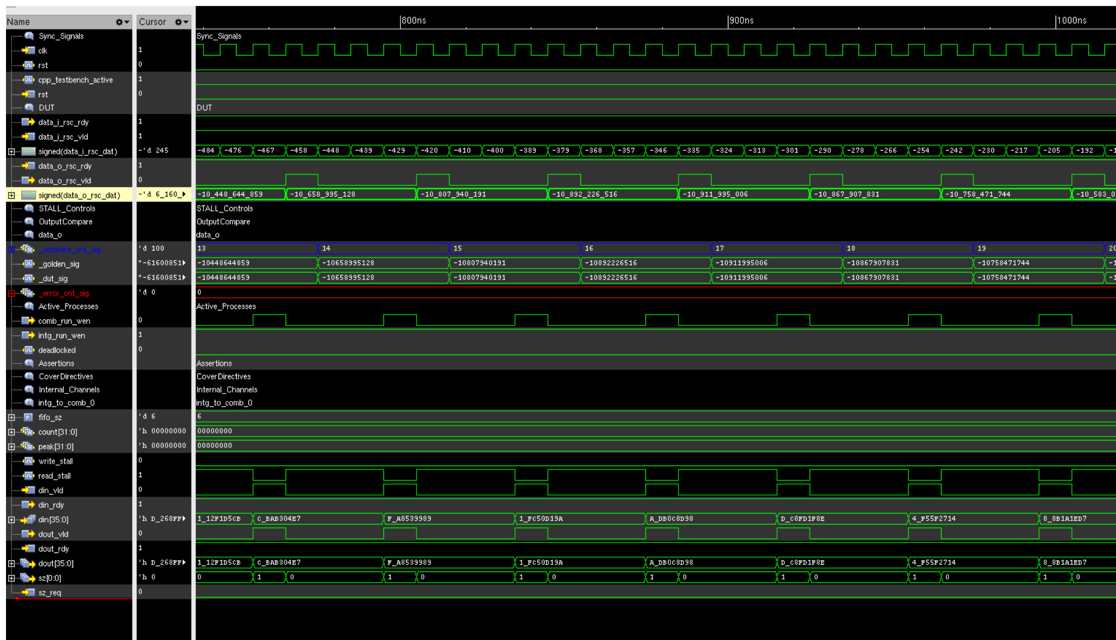


Figure 5.8: NCSim simulation of given CIC filter

5.5 Results

5.5.1 Order:3 , Decimation:512

For evaluation of results using the automated and generic CIC filter flow using HLS, the complete structure, hierarchy, template, source codes and constraints have been setup as the first step. Now it is possible to implement CIC filters of various specifications and analyze the corresponding results.

The first CIC filter to be designed using the automated HLS flow has **Order = 5** and **Decimation = 512** with input bitwidth of 2 and output bitwidth of 16. Only these four parameters are changed in the Matlab script, and the rest of the header files are updated automatically, providing the hardware design of this filter via Catapult HLS.

The resulting RTL netlist is then synthesized using **Cadence Genus Synthesis**

Solution at a frequency of **100 MHz** to obtain an area of **28443 um²**.

```
Bill Of Materials (Datapath)
  Component Name                                                  Post Alloc Post Assign
  ------------------------------                                  ---------- -----------
  [Lib: ccs_ioport]
  ccs_in_wait(1,2)                                                         1           1
  ccs_in_wait(4,29)                                                        1           1
  ccs_out_wait(2,29)                                                       1           1
  ccs_out_wait(5,16)                                                       1           1
  [Lib: starlib_reg_10t_or]
  mgc_add(16,0,1,0,16,7)                                                   1           1
  mgc_add(29,0,2,1,29,7)                                                   1           1
  mgc_add(29,0,29,0,29,7)                                                  5           5
  mgc_add(9,0,1,0,10,7)                                                    0           1
  mgc_add(9,0,2,1,10,7)                                                    1           0
  mgc_and(1,2,2)                                                           0          19
  mgc_and(10,2,1)                                                          1           0
  mgc_mux(2,1,2,2)                                                         0           1
  mgc_mux(29,1,2,4)                                                        0           1
  mgc_nor(1,2,2)                                                           0           9
  mgc_not(1,1)                                                             0          11
  mgc_not(29,1)                                                            0           3
  mgc_or(1,3,1)                                                            0           4
  mgc_reg_pos(1,0,0,1,1,0,0,4)                                             0           4
  mgc_reg_pos(1,0,0,1,1,1,1,4)                                             0           5
  mgc_reg_pos(16,0,0,1,1,1,1,4)                                            0           1
  mgc_reg_pos(2,0,0,1,1,1,1,4)                                             0           1
  mgc_reg_pos(29,0,0,1,1,1,1,4)                                            0           9
  mgc_reg_pos(9,0,0,1,1,1,1,4)                                             0           1

  TOTAL AREA (After Assignment):  21728.475          4077.000 8351.000

Area Scores
                    Post-Scheduling    Post-DP & FSM   Post-Assignment
  ----------------- ---------------  ---------------  ---------------
  Total Area Score:  16771.9          22067.2          21892.5
  Total Reg:          9981.8  (60%)   14066.2  (64%)   14066.2  (64%)

  DataPath:          16771.9 (100%)   21903.2  (99%)   21728.5  (99%)
    MUX:                 0.0            489.5   (2%)     489.5   (2%)
    FUNC:             6710.5  (40%)    6710.5  (31%)    6535.8  (30%)
    LOGIC:              79.6   (0%)     785.0   (4%)     785.0   (4%)
    BUFFER:              0.0              0.0              0.0
    MEM:                 0.0              0.0              0.0
    ROM:                 0.0              0.0              0.0
    REG:              9981.8  (60%)   13918.2  (64%)   13918.2  (64%)

  FSM:                   0.0            164.0   (1%)     164.0   (1%)
    FSM-REG:             0.0            148.0  (90%)     148.0  (90%)
    FSM-COMB:            0.0             16.0  (10%)      16.0  (10%)
```

Figure 5.9: BOM of Order 3 CIC filter

The BOM of this filter is shown in figure 5.9. As expected, the total number of adders are **8** in which **6** of them are used for the integrator and comb parts (3 each), **1** adder is used as counter and the remaining **1** is used for the purpose of half bit rounding of the final result.

Out of the 6 internal adders, 5 of them are shown as:

mgc_add(29,0,29,0,29,7)

which means that the two operands are 29 bits each as well as the output is on 29 bits. This is the internal accumulator bitwidth which allows for full precision and correct results, without loss of accuracy.

mgc_add(29,0,2,1,29,7) is the first adder in the integrator part as it receives a 2-bit input at the start. **mgc_add(16,0,1,0,16,7)** is the adder used for half bit rounding. as it adds the final result on 16 bits with 1. And **mgc_add(9,0,1,0,10,7)** is used for counting purposes (9 bits are used to represent the maximum value of 511).
This filter is compared with another filter having the same specifications and provided by Infineon. The comparative area results are depicted in figure 5.10.

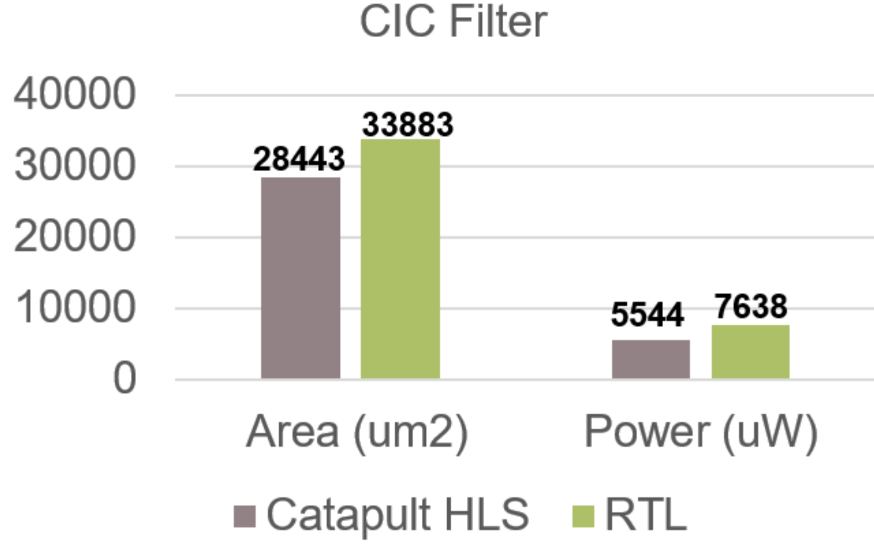


Figure 5.10: Area comparison of order 3 and decimation 512 filter

The number of flip flops in our HLS design are more than that of Infineon because of 2 sets of extra registers i.e. the output register of 29 bits at the integrator stage and another output register of 16 bits at the comb stage (which is also the final

output). Hence, the flop area of our design is greater.
On the other side, the combinational area is less because of two main reasons.

- Our design incorporates the delay registers in the straight, feedforward path inside the integrator stage. This is done to match the results with the Matlab algorithm which uses the same implementation. This eases the timing constraints and hence, slower and smaller adders can be used. In contrast to this, the design provided by Infineon incorporates these registers in the feedback path, which leaves a single, straight combinational path, directly from the input to the decimation block. Due to this, faster adders have to be used in order to meet the timing constraints.

- The Infineon design has saturation logic at the end to saturate the final result to maximum positive or maximum negative, depending on the value of first 2 MSBs. This also includes some extra logic in the design.

5.5.2 Order:4 , Decimation:8

The next CIC filter to be designed using the automated HLS flow has **Order = 4** and **Decimation = 8** with input bitwidth of 2 and output bitwidth of 14.
This is compared with two other implementations of the same filter; one is provided by Infineon and the other is RTL-coded automated flow. The results are shown in figure 5.11.

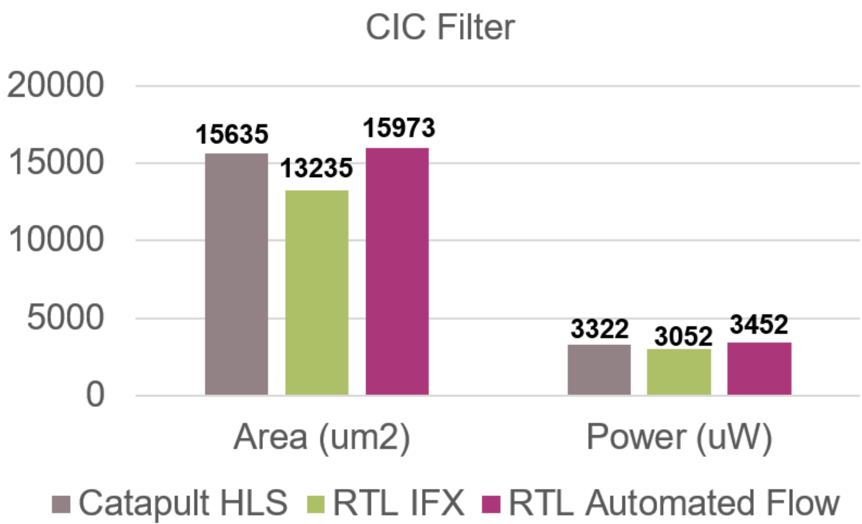


Figure 5.11: Area comparison of order 4 and decimation 8 filter

As evident from the results, it can be seen that the area and power results of Catapult HLS automated flow and RTL automated flow are almost exactly the same.

For the comparison with Infineon design, the difference in the area can be attributed to one major factor i.e. the total number of registers used in the design. In our HLS flow, there are a total of **153** flip flops, whereas in the Infineon design, there are a total of **108** flip flops. These lesser number of flip flops arise from the architectural details inside the filter and depend solely on how the design has been implemented. The Infineon design, being part of a bigger IP, does not have any output register at the integrator stage and also no register with the last subtractor in the comb part. This difference in the number of flip flops accounts for the disparity between the results.
One should also account for some extra *net area* incase of extra flip flops. The scheduling operations of this CIC filter are depicted in Figure 5.12.

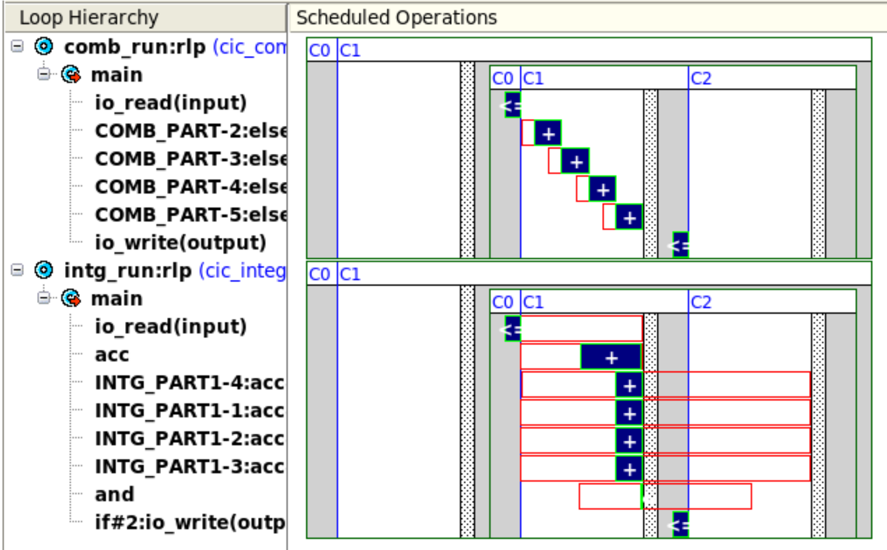


Figure 5.12: Schedule of order 4 and decimation 8 filter

Chapter 6

Design of Filter Chain

The last three chapters have dealt with the architecture, algorithmic description and the optimizations with regards to three types of filters: FIR, Polyphase and CIC. The implementation of these filters with Catapult HLS has also been discussed in great detail along with the results and comparison with contemporary solutions. However, in the world of signal processing, it is often desirable to work with a chain of digital filters where the output of one filter is directly connected to the input of the next one. Such an implementation can be used to derive overall better results, which would otherwise be difficult to achieve using a single filter. Using multiple filters also lets the designer to put less constraints on the individual filters which results in simpler architectures.

This chapter deals with the ***design of filter chains using the parameterized filter structures already available***. We also discuss the method to successfully integrate all the different filters into a single top level design such that it behaves as one entity or block, but comprising of different independent sub-blocks.

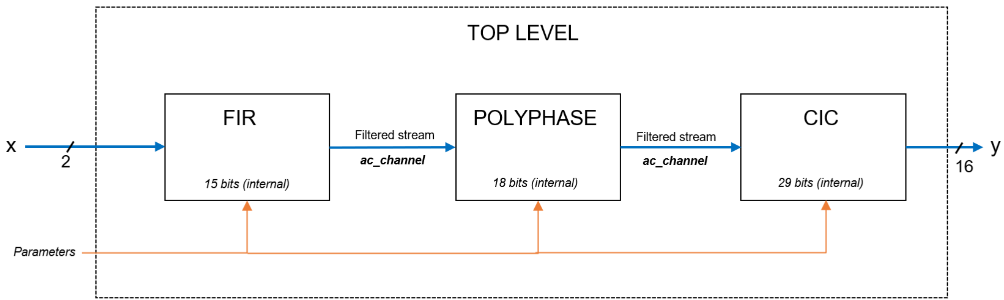


Figure 6.1: Filter chain in a hierarchical design

6.1 Architecture

Figure 6.1 depicts the general architecture of such a filter chain where the different blocks of filters are connected together in a seamless fashion. All the parameters are fed into the design by the user from the outside, before synthesis. Doing so allows Catapult to generate the corresponding architectures of each filter in the most optimized way. The incoming samples 'x' are usually a 2-bit digital stream coming from the Delta Sigma modulator which are fed into the 1st block which could be a FIR filter, a polyphase filter or a CIC filter.
The 1st filter performs the filtration process and outputs its result on a specific number of bits (specified by the user) which is then fed into the 2nd filter. This process of filtration and feeding the next filter keeps on repeating until the whole chain has been exhausted.

6.2 Matlab code and reference model

As with the previous filters, the filter chain also needs to be generic such that the user has the option to customize the whole chain along with the internal filter

specifications as per his requirement.

Because we are using the available filters (FIR, CIC, polyphase), their general parametric structure remains unchanged. For the Matlab part, two major changes need to be incorporated which are distinct from the previous Matlab codes.

- **Modification of source files:** Previously, only the *top.h* header file was changed / updated because it contains the parameters / variables needed by other source files. For the filter chain, in addition to this header file, the *top_template.cpp* and *testbench.cpp* files are also updated.

  The top.h header file contains the directive to include the coefficients as constants, and these coefficients are also automatically stored by the Matlab code in a separate file.

- **Creation of golden reference model:** The previous filter designs were easier to implement in Matlab as they only consisted of a single set of inputs and outputs.

  In the filter chain, as the output of each filter is connected to the input of next filter, there is a need to connect the sub-blocks in the correct way. To automate this process, we use loops whose number is defined by the number of filters in the chain. Secondly, the "type" of filter further decides which specific implementation to execute during that loop iteration.

```
1          for i=1:FILT_NUM
2
3      xi = y_msbs;
4
5 %                         **************
6 %                           FIR filter
7 %                         **************
8      if CHAIN_TYPE(i) == 1
```

```
9
10         fir_no = fir_no+1;
11         coefficients_filename_FIR = sprintf('
     coefficients_CHAIN_FIR_%d.txt',fir_no);
12
13         fp=fopen(coefficients_filename_FIR , 'r');
14         bi = fscanf(fp, '%d');
15         fclose(fp);
16
17         [line_coeffs] = func_write_coeffs_vec(line_coeffs,bi,
     fir_no,poly_no,CHAIN_TYPE(i));
18
19         yi = filter(bi, 1, xi);
20         if i == 1
21             nb_acc(i) = ceil(log2(  sum(abs(bi)) * 2^(nb_in-1)  )
     ) + 1;
22         else
23             nb_acc(i) = ceil(log2(  sum(abs(bi)) * 2^(nb_out(i-1)
     -1)  )) + 1;
24         end
25
26         lsb_start = nb_acc(i) - nb_out(i);
27
28         if lsb_start < 1
29             lsb_start = 0;
30             round_add = 0;
31         elseif lsb_start >= 1
32             lsb_start = lsb_start - 1;
33             round_add = 1;
34     end
35
36
37     y_msb_plus_1 = floor(yi/(2^(lsb_start)));
```

```
38        y_msb_plus_1 = y_msb_plus_1 + round_add;
39        y_msbs = floor(y_msb_plus_1/(2^(round_add)));
40
41
42
```

This code snippet is extracted from the Matlab code and shows the part where the FIR filter implementation is carried out inside the chain (which depends on the parameter **CHAIN_TYPE**.

The coefficients are extracted from a separate text file and further saved into a new file with the correct format, which can be read by C++ files.

Then the filtration function is performed where 'bi' are the coefficients and 'xi' are the inputs. These inputs can be either from the external environment (if this is the first filter) or they can be the output of some previous filter. The number of accumulator bits are also updated in the meanwhile.

At the end, each output is half-bit rounded on the correct number of bits before going as input to the next filter.

6.3 Algorithmic description in C++

Similar to previous individual filters, this chain of digital filters is also designed using hierarchical C++ blocks, composed up of classes. The connection of the different blocks into a sub-system is one of the major challenges in chip design, which can be handled in a seamless way by following a consistent set of rules.

Adding hierarchy allows the different process or blocks to run in parallel. Another major advantage of using hierarchy here are the different rates at which data is being transferred between blocks. Although synchronization of the different data rates can be a tricky process in RTL design, it is handled easily by defining *ac_channels* on the data ports. These channels manage the data flow for the whole

system while taking care of the throughput requirements. During synthesis of these channels, Catapult automatically adds the proper handshaking signals to the RTL netlist.

Another problem of different data rates is the availability of input data at some specific block in the chain. The rate at which a block receives data is totally dependent on the output data rates of the previous blocks. During a C++ simulation, if a block attempts to read an empty channel (because the data from the previous block is coming at a slower rate), then the simulation crashes. To avoid this, we make use of the **available()** function which checks the number of values in the channel and returns true if are "num" values available, where "num" is the argument to the function. The main body of the filter function block is only executed when there is sufficient data available. This function acts as a safety check to make sure that the C++ model does not encounter the "empty channel" assertion.

The first step is to add a scope with a while-loop around it that is not synthesized by Catapult, similar to the concept of sensitivity list in RTL design. This while-loop contains the "available" method so that it is only called when the function has enough data. This while-loop makes sure that the functions running at different rates will execute enough times to complete the simulation.

```
1
2  #pragma hls_pipeline_init_interval 1
3  #pragma hls_design interface
4  void CCS_BLOCK(intg_run)(ac_channel<inType> &input,
5                           ac_channel<outType> &output) {
6
7      inType data_in_initial;
8
9      #ifndef __SYNTHESIS__
10     while (input.available(1))
```

```
11      #endif
12      {
13          < design body >
14      }
```

The code snippet above shows how this *available* function is implemented. It requires the input channel to have one value before the body of the design is executed. It guarantees that enough values are in the input channel before it is called to "read".
Moreover, all the intermediate results coming out of the individual filters as well as the overall final result are half-bit rounded before going as input into the next filter.

This filter chain being a class-based design, also includes different different files, the details of which are mentioned below:

1. **top.h** header file which contains the declarations of the variables, constants, parameters and functions. The user has to input the number of filters inside the chain as well as the corresponding parameters of each filter e.g. filter order, bitwidth, decimation rate etc. This file needs to be edited for each new design, and this is done automatically by the Matlab script.

2. **top_template.cpp** is the "top-level" module class which is connected to the external testbench and receives the input as well as sends the final output of the DUT.
   The code of this file can be further divided into multiple portions whose explanation is given as:

   (a) First, there is the need to instantiate all the individual filters inside the chain using the templatized classes which have already been defined.

```
// Instantiation of filters

cic_decimator_template <data_i_type, DATA_O_TYPE_1, FB_METHOD
    , DEC_FACTOR_CIC_1, FILTER_ORDER_CIC_1, 0> cic_inst_0;
cic_decimator_template <DATA_O_TYPE_1, DATA_O_TYPE_2,
    FB_METHOD, DEC_FACTOR_CIC_2, FILTER_ORDER_CIC_2, 1>
    cic_inst_1;
cic_decimator_template <DATA_O_TYPE_2, DATA_O_TYPE_3,
    FB_METHOD, DEC_FACTOR_CIC_3, FILTER_ORDER_CIC_3, 2>
    cic_inst_2;
cic_decimator_template <DATA_O_TYPE_3, DATA_O_TYPE_4,
    FB_METHOD, DEC_FACTOR_CIC_4, FILTER_ORDER_CIC_4, 3>
    cic_inst_3;
cic_decimator_template <DATA_O_TYPE_4, DATA_O_TYPE_5,
    FB_METHOD, DEC_FACTOR_CIC_5, FILTER_ORDER_CIC_5, 4>
    cic_inst_4;
fir_template <DATA_O_TYPE_5, COEFF_TYPE, DATA_O_TYPE_6,
    DATA_ACC_TYPE_6, SYMM_FILT, FILTER_ORDER_FIR_1,
    coeff_array_fir_1, 0> fir_inst_0;
poly_template <DATA_O_TYPE_6, COEFF_TYPE, data_o_type,
    DATA_ACC_TYPE_7, DEC_FACTOR_POLY_1, FILTER_ORDER_POLY_1,
    coeff_array_poly_1, 0> poly_inst_0;

```

The filters take their parameters from the *top.h* header file and this determines the characteristics / architecture of each filter. The data type of the first input coming from outside (*data_i_type*) is passed as the input parameter to the first filter. Then the data type of the output of this filter needs to match the data type of the input of the second filter. This is shown from the code above where the **DATA_O_TYPE_1** is passed as output of the first filter and input of the second.

The same process is repeated for all the other filters as well. Finally, the data type of the last filter should match the data type of the final output, which is passed to the testbench (*data_o_type* in this case).

(b) Secondly, this top level module is used to define all the interconnecting channels which connect the different sub-blocks of filters together. For the example given above, there would be six channels. The FIFO depth for these channels is also kept at zero.

```
1
2  #pragma hls_fifo_depth 0
3  ac_channel <DATA_O_TYPE_1> channel_1;
4  #pragma hls_fifo_depth 0
5  ac_channel <DATA_O_TYPE_2> channel_2;
6  #pragma hls_fifo_depth 0
7  ac_channel <DATA_O_TYPE_3> channel_3;
8  #pragma hls_fifo_depth 0
9  ac_channel <DATA_O_TYPE_4> channel_4;
10 #pragma hls_fifo_depth 0
11 ac_channel <DATA_O_TYPE_5> channel_5;
12 #pragma hls_fifo_depth 0
13 ac_channel <DATA_O_TYPE_6> channel_6;
14
```

(c) At the end, it consists of a constructor (without any arguments) and a public member function ***run*** which is called by the testbench via ac_channels. This member function is used to call all of the sub-blocks (individual filters) which are connected via ac_channels.

3. **cic_decimator_template.h** is the core file which contains the synthesizable description of the CIC decimation filter and it further contains two classes:

- **cic__integrator__stage** class which contains the source code for integrator part of the CIC filter.

- **cic__comb__stage** class which contains the source code for comb part of the CIC filter.

Both these classes are parameterized through templates and they are instantiated by the *cic__decimator__template.h* class which also passes the template parameters.

4. **fir__template.h** is the core file which contains the synthesizable description of the FIR filter along with all the loops. It is also parameterized through templates and it is instantiated as shown in the code above. The details of the implementation have already been discussed in chapter 4.

5. **poly__template.h** is the core file which contains the synthesizable description of the polyphase filter. It is also parameterized through templates and it is instantiated as shown in the code above. The details of the implementation have already been discussed in chapter 5.

6. **top__tb.cpp** testbench file which is used to read the inputs from an external file, pass the values in the top-level module class, receive the results, compare them and declare the functional test "pass" or "fail" depending upon the comparison.

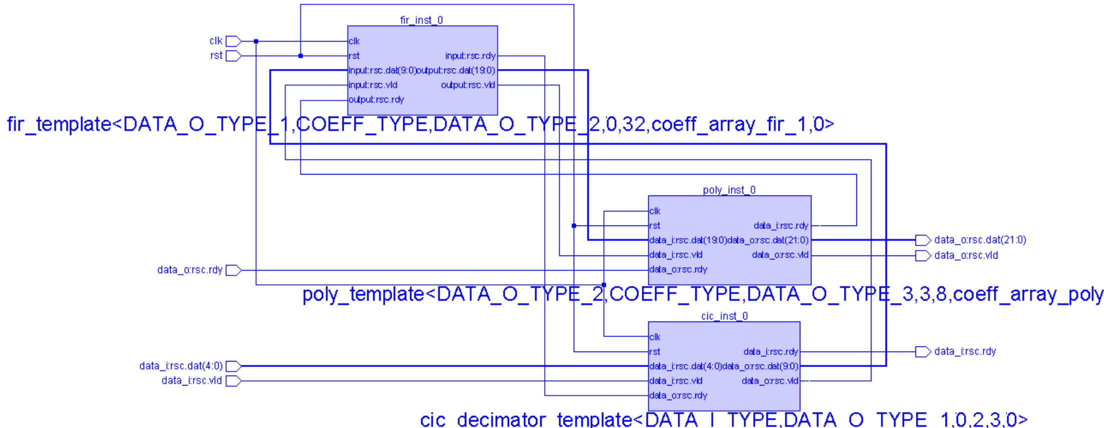


Figure 6.2: Block diagram view of the filter chain inside Catapult

6.4 Architectural optimizations

As with the previous designs, in order to avoid repetition of the explanation, the following optimizations have already been carried out in the individual filters contained inside the chain:

1. Pipelining.
2. Loop unrolling.
3. Use of class-based hierarchy.
4. Resource sharing in the lower frequency (comb) part of the CIC filters.
5. Removal of FIFO queue inside the CIC filters.
6. Removal of FIFO queue between the sub-blocks.

Regarding Point 4, this resource sharing in the comb part of the CIC filters results in increased area savings in the overall filter chain. This is especially true for this case in which the cumulative growth of the decimation rates of the individual

filters results in an overall increase in the decimation rate progressively. Hence, the frequency of the sampling gets further reduced as you move down the chain, and there is more possibility to share the hardware resources.

This situation is depicted in the the following two figures where Figure 6.3 shows the scheduling operation of an unshared adder *(in the comb part a CIC filter inside the chain)*, and Figure 6.4 shows the scheduling operation of a shared adder.

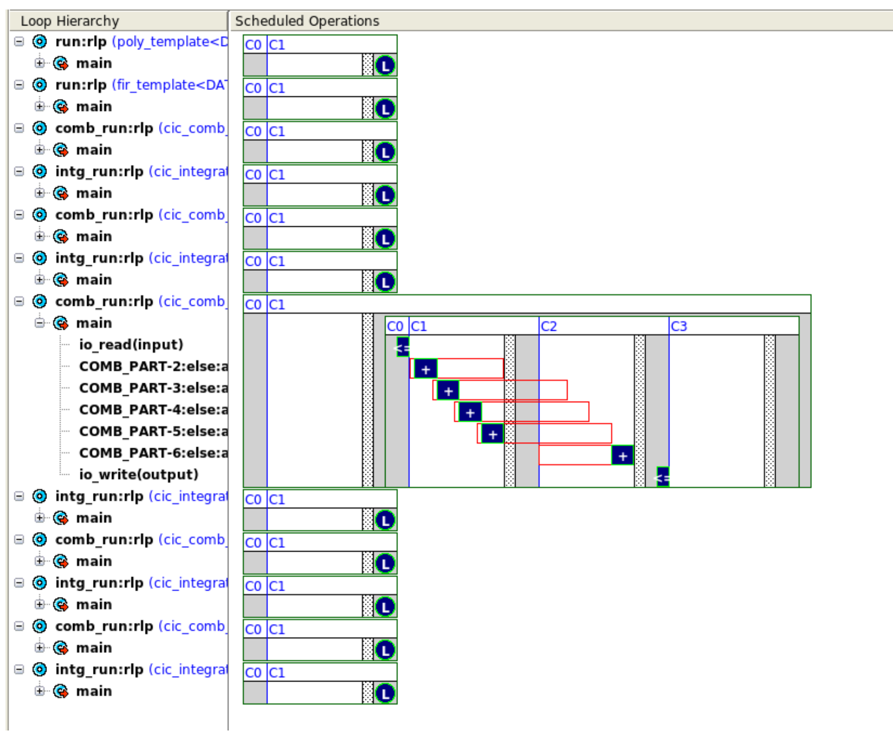


Figure 6.3: Scheduling operation showing an un-shared adder

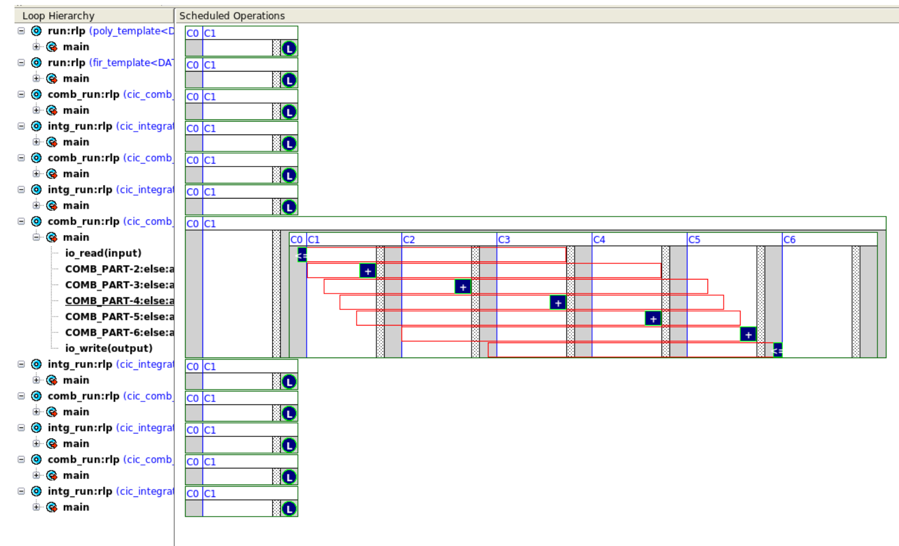


Figure 6.4: Scheduling operation of a shared adder over 4 ccs

6.5 Results

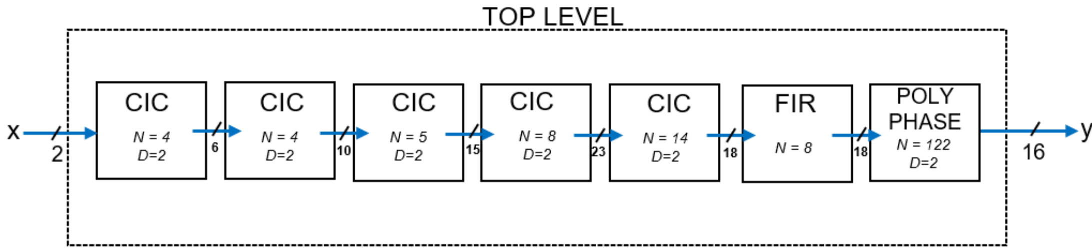


Figure 6.5: Block diagram view of the filter chain

Figure 6.5 is the filter chain architecture which has been implemented using the automated HLS flow. It contains a total of 7 filters and has the specifications as shown in table 6.1:

By putting in the required parameters inside the Matlab file and running the script, all the files are updated accordingly, as explained in the previous sections.

Filter Type	CIC	CIC	CIC	CIC	CIC	FIR	Polyphase
Order	4	4	5	8	14	8	122
Decimation	2	2	2	2	2	1	2
nb_in	2	6	10	15	23	18	18
nb_out	6	10	15	23	18	18	16

Table 6.1: Filter chain specifications

Then by running the Catapult script, we get the final architecture of the complete chain as per the given specifications.

The RTL netlist, after synthesis by **Cadence Genus Synthesis Solution**, provides a final area of **658996 um^2** at the frequency of **100 MHz**.

The area score of the chain is shown in Figure 6.6.

Area Scores

	Post-Scheduling	Post-DP & FSM	Post-Assignment
Total Area Score:	685673.4	735704.2	697235.7
Total Reg:	216130.8 (32%)	237620.9 (32%)	237620.9 (34%)
DataPath:	**685673.4 (100%)**	**732863.2 (100%)**	**694394.7 (100%)**
MUX:	78638.2 (11%)	86196.7 (12%)	97805.0 (14%)
FUNC:	390491.9 (57%)	402208.3 (55%)	351969.5 (51%)
LOGIC:	412.5 (0%)	9582.4 (1%)	9744.3 (1%)
BUFFER:	0.0	0.0	0.0
MEM:	0.0	0.0	0.0
ROM:	0.0	0.0	0.0
REG:	216130.8 (32%)	234875.9 (32%)	234875.9 (34%)
FSM:	**0.0**	**2841.0 (0%)**	**2841.0 (0%)**
FSM-REG:	0.0	2745.0 (97%)	2745.0 (97%)
FSM-COMB:	0.0	96.0 (3%)	96.0 (3%)

Figure 6.6: Area score of filter chain

This filter chain has been compared with an RTL-coded implementation of the same specifications. As discussed in the previous chapters, both of these implementations are the result of automated design flows.

Figure 6.7.

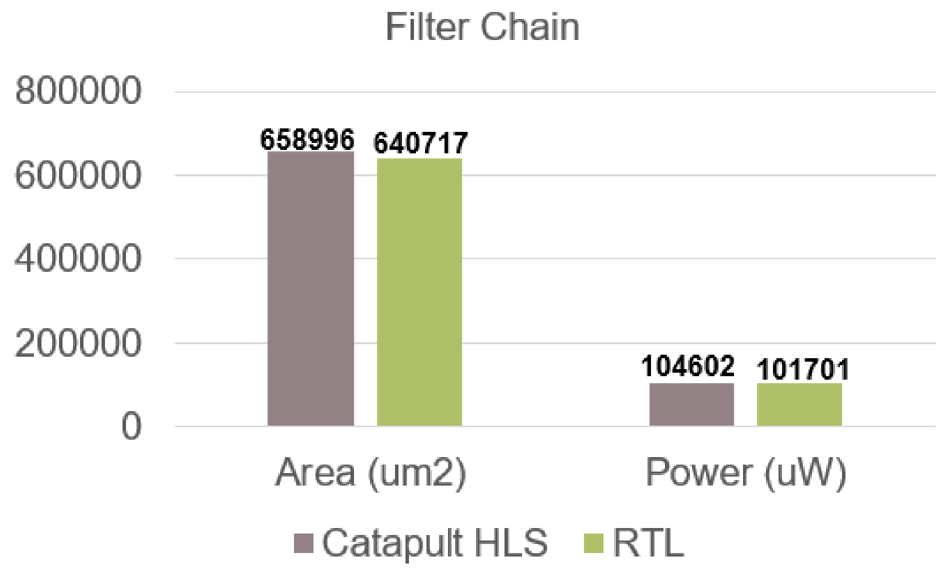


Figure 6.7: Area comparison of filter chain with 7 filters

As can be seen from the graph, both the design flows result in almost equivalent area results. However, upon further scrutiny, it is shown that the area scores of the individual filters vary in size.
The reasons for these differences have already been discussed in the previous chapters and can be summed up as:

- The FIR and CIC filters of the HLS implementation have lesser area due to optimizations such as resource sharing and usage of constant adders / shifters instead of multipliers.
- The polyphase filter of the RTL implementation has lesser area because of the smaller decimation rate of 2.

This shows that the different filter chains will result in different area comparisons of the two implementations.

Chapter 7

Conclusion

This thesis which has been done in association with Infineon Technologies Austria, offers a novel method of digital hardware design (specifically digital filter IPs). The complete design process is not only automated but also parameterizable by using a combination of Matlab and HLS tool Catapult.

This fusion of methodologies to create a new process yields a refined approach which not only offers dividends in time savings but also provides the user with the option to explore different architectures by just setting the required parameters in a single file.

The performance results in this case study demonstrate that the use of HLS technology combined with an automated flow can be considered as a valid design methodology, with results at par with the classical RTL design. Because hardware optimizations in Catapult can be achieved by properly setting constraints, the extra amount of effort generally required for this task by writing HDL code can be eliminated. Furthermore, the reduced time and effort for the verification flow offers an added advantage in this process. These generic and templatized DSP IPs can also be made part of a virtual 'toolbox' which can serve as the basis for architectural exploration vis-a-vis best quality of results.

In conclusion, this modus operandi can serve as a very effective medium for designing digital structures, producing high-quality outcomes with almost zero or a little effort on the part of the designer who can do away with the additional work of manually coding HDL code.

Bibliography

[1] Yousef Iskander, Cameron Patterson, and Stephen Craven. «High-level abstractions and modular debugging for FPGA design validations». In: *IACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.1 (2014), pp. 1–22.

[2] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. «An Introduction to High-Level Synthesis». In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17.

[3] T. Ogunfunmi and S. Desai. «Fast FIR filter implementation using high-level synthesis tools». In: *Proceedings of 1994 37th Midwest Symposium on Circuits and Systems*. Vol. 1. 1994, pp. 58–61.

[4] Liu Hanbo, Wang Shaojun, and Zhang Yigang. «Design of FIR filter with high level synthesis». In: *2015 12th IEEE International Conference on Electronic Measurement & Instruments (ICEMI)*. Vol. 2. 2015, pp. 1067–1071.

[5] Kazutoshi Wakabayashi, Takashi Takenaka, and Hiroaki Inoue. «Mapping complex algorithm into FPGA with High Level Synthesis reconfigurable chips with High Level Synthesis compared with CPU, GPGPU». In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2014), pp. 282–284.

[6] Affaq Qamar, Claudio Passerone, Luciano Lavagno, and Francesco Gregoretti. «Design space exploration of a stereo vision system using high-level synthesis». In: *MELECON 2014 - 2014 17th IEEE Mediterranean Electrotechnical Conference*. 2014, pp. 500–504.

[7] Shahzad Ahmad Butt and Luciano Lavagno. «Design space exploration and synthesis for digital signal processing algorithms from Simulink models». In: *2013 8th IEEE Design and Test Symposium*. 2014, pp. 1–6.

[8] Mohammad Amir Mansoori and Mario R. Casu. «Hardware Acceleration of Biomedical Microwave Techniques using High Level Synthesis». In: *2022 16th European Conference on Antennas and Propagation (EuCAP)*. 2022, pp. 1–5.

[9] Gordon Inggs, Shane Fleming, David Thomas, and Wayne Luk. «Is high level synthesis ready for business? A computational finance case study». In: *2014 International Conference on Field-Programmable Technology (FPT)*. 2014, pp. 12–19.

[10] Christian Fibich, Stefan Tauner, Peter Rossler, Martin Horauer, Herbert Taucher, and Martin Matschnig. «Preliminary Evaluation of High-level Synthesis Tools - Xilinx Vivado and PandA Bambu». In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–4.

[11] Christian Pilato and Fabrizio Ferrandi. «Bambu: A modular framework for the high level synthesis of memory-intensive applicationsu». In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013, pp. 1–4.

[12] Tero Joentakanen. «Evaluation of HLS modules for ASIC Backend». MA thesis. Finland: Tampere University of Technology, 2016.

[13] *Catapult High-Level Synthesis and Verification - Design Platform Empowering Engineers*. Fact Sheet. Siemens, 2024.

[14] *Siemens Digital Industries Software*. URL: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/.

[15] *Working smarter, not harder: NVIDIA closes the design complexity gap with HLS*. White Paper. NVIDIA, 2016.

[16] Shawn McCloud. *Catapult C Synthesis-Based Design Flow: Speeding Implementation and Increasing Flexibility*. White Paper. Mentor Graphics, 2004.

[17] Siemens. *Catapult Synthesis HLS Bluebook*. 2023.

[18] «Customers Discuss their Real-World Use of High-Level Synthesis». In: (2019). URL: https://webinars.sw.siemens.com/en-US/customers-discuss-their-real-world-use-of-high-level-synthesis?bc=eyJwYWdlIjoiNG9JNGVvZnVOSk1LUTVpa3ZFSUZtTyIsInNpdGUiOiJlZGEiLCJsb2NhbGUiOiJlbi1VUyJ9.

[19] M. Bellanger, G. Bonnerot, and M. Coudreuse. «Digital filtering by polyphase network:Application to sample-rate alteration and filter banks». In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24.2 (1976), pp. 109–114.

[20] P.P. Vaidyanathan. «Multirate digital filters, filter banks, polyphase networks, and applications: a tutorial». In: *Proceedings of the IEEE* 78 (1990), pp. 56–93.

[21] Rozita Teymourzadeh and Masuri Othman. *VLSI Implementation of Cascaded Integrator comb filters for DSP applications*. Tech. rep. VLSI Design Research Group, National University of Malaysia, June 2018.

[22] E. Hogenauer. «An economical class of digital filters for decimation and interpolation». In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29.2 (1981), pp. 155–162.

[23] Richard G. Lyons. *Understanding Digital Signal Processing (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.