UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

MASTER'S THESIS

C++ CODING PRINCIPLES FOR HIGH-LEVEL SYNTHESIS

Author	Heikki Vuopio
Supervisor	Jukka Lahti
Second Examiner	Jussi Jansson
Technical Advisor	Teemu Vikamaa

June 2021

Vuopio H. (2021) C++ CODING PRINCIPLES FOR HIGH-LEVEL SYNTHESIS. University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 63p.

ABSTRACT

High-level synthesis (HLS) raises the level of abstraction on digital integrated circuit design from traditional register transfer level (RTL) to behavioural system description level. This methodology offers great advantages such as increased designer productivity. The adoption of HLS, however, has been slowed down by the RTL code mistakenly generated with HLS which potentially results in poor quality compared to the traditional hand-written RTL.

This thesis aims to solve this problem by finding the best programming practices for hardware-oriented C++. A digital downconverter and decimator are designed and implemented with Catapult HLS as a case study, where different coding practises are experimented with, and the best ones are generalized and presented. The quality of results of this case study is compared against a hand-written RTL design of the same intellectual property created by other designers. A few examples are presented as well demonstrating that small changes in the source code might have a major effect on the generated RTL.

It is found that understanding how the HLS tool analyses the source code and executes operations in parallel greatly helps to improve the quality of results in the generated hardware. Also, by having a clear target architecture it is a simple task to verify the hardware in Catapult analysis views such as schedule and schematic view. By optimizing the source code, it is possible to generate similar quality hardware compared to traditional RTL flow. In this case, the area of the HLS design is about 19 % smaller than the RTL design with the same throughput, slightly lower latency, and roughly the same power consumption.

Key words: high-level synthesis, HLS, digital IC design, hardware-oriented C++, Catapult HLS.

Vuopio H. (2021) C++ OHJELMOINTIKÄYTÄNNÖT KORKEAN TASON SYNTEESIIN. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 63p.

TIIVISTELMÄ

Korkean tason synteesi (HLS) nostaa digitaalisten integroitujen piirien suunnittelun abstraktiotason perinteiseltä rekisterinsiirtotasolta (RTL) systeemikuvaustasolle. Tämä metodologia tuo suuria etuja, kuten suunnittelijan korkeampi tuotteliaisuus. HLS:n laajempaa käyttöönottoa on kuitenkin hidastanut erheellisesti HLS:llä generoitu RTL-koodi, josta usein seuraa heikohko laatu käsin kirjoitettuun RTL-koodiin verrattuna.

Tämän tutkimuksen tavoite on ratkaista tämä ongelma löytämällä parhaat ohjelmointikäytännöt korkean tason synteesiin suunnattuun C++-ohjelmointiin. Digitaalinen alasmuunnin ja desimaattori suunnitellaan ja implementoidaan käyttäen Catapult HLS-työkalua. Eri ohjelmointikäytäntöjä testataan ja parhaat yleistetään ja esitellään, minkä jälkeen tulosten laatua verrataan samaan lohkoon, jonka on ohjelmoinut eri suunnittelijat rekisterinsiirtotasolla. Tutkimus sisältää myös koodiesimerkkejä siitä, miten pienet muutokset lähdekoodissa voivat vaikuttaa merkittävästi lopputulokseen.

Tutkimuksessa todetaan, että synteesityökalun toiminnan ymmärtäminen on kriittistä hyvien tulosten saavuttamisen kannalta. Suunnittelijalla tulisi olla selvä tavoitearkkitehtuuri generoitavasta RTL-koodista, jolloin sen varmentaminen synteesin jälkeen olisi helppoa Catapultin analyysinäkymissä. Optimoimalla lähdekoodia generoidun RTL-koodin tulosten laatu saadaan samaksi kuin käsin kirjoitetun RTL-koodin. Tässä tapauksessa generoidun RTL-koodin pinta-ala on 19 % pienempi kuin käsin kirjoitetun mallin samalla siirtonopeudella. Latenssi on hieman pienempi ja tehonkulutus samaa suuruusluokkaa.

Avainsanat: korkean tason synteesi, digitaalisuunnittelu, laitteistosuuntautunut C++, Catapult HLS.

TABLE OF CONTENTS

FOREWORD

The purpose of this thesis was to find the best practises for HLS-oriented C++ programming. The thesis work was done at Nokia Oulu from late 2020 to summer 2021.

I would like to thank Nokia SoC organization for the opportunity to do this thesis. Special thanks to the thesis supervisor Jukka Lahti for guidance with the thesis work, Teemu Vikamaa for technical advising, and Esa-Matti Turtinen and Richard Langridge from Siemens EDA for the support in tool-related matters and the guidance on the research subject itself.

Finally, I want to thank my family, friends, and my girlfriend Emma for all the support provided during the thesis work and the recent years of studying.

Oulu, June 30, 2021

Heikki Vuopio

LIST OF ABBREVIATIONS AND SYMBOLS

AC	Algorithmic C
ASIC	Application Specific Integrated Circuit
CDFG	Control Data Flow Graph
CCORE	Catapult C Optimized Reusable Entity
DCT	Discrete Cosine Transform
DFE	Digital Front-End
DFG	Data Flow Graph
DL	Downlink
DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HBF	Half Band Filter
HEVC	High-Efficiency Video Coding
HLL	High-Level Language
HLS	High-Level Synthesis
HLV	High-Level Verification
HW	Hardware
IC	Integrated Circuit
II	Initiation Interval
IO	Input/Output
IP	Intellectual Property
KPN	Kahn Process Network
LSB	Least Significant Bit
MatchLib	Modular Approach to Circuits and Hardware Library
MSB	Most Significant Bit
NCO	Numerically Controlled Oscillator
OOP	Object-Oriented Programming
PSD	Polyphase Sub-band Decimation
QoR	Quality of Results
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level
SoC	System on Chip
SW	Software
TCL	Tool Command Language
UL	Uplink
UVM	Universal Verification Methodology
VHDL	Very high-speed integrated circuit Hardware Design Language

1 INTRODUCTION

High-level synthesis [1] (HLS) is a methodology that raises the abstraction level in digital integrated circuit (IC) design. HLS tool takes the untimed or loosely timed high-level source code, technology library, and a set of design constraints as input and outputs the clock accurate register transfer level (RTL) model of the design that is traditionally written by hand.

The first HLS tools were introduced in the 1980s, but only in recent years, HLS has been adopted by a wider audience due to tool development and growing design complexity in digital IC projects [1]. The promises of HLS are great since the tool developers promise similar, or better quality of results (QoR) with greatly decreased design and verification effort and true code reusability across projects via technology-independent source code.

A challenge with HLS is that the synthesis tool abstracts away some details of the generated RTL and the input source code is written in a high-level language that only describes the behaviour of the design. Because of this, it might not be completely clear for the designer how the tool interprets the source code and generates RTL [2]. This may lead to suboptimal quality hardware with the designer having no clue how to improve it since the hardware-oriented programming principles differ from traditional software ones.

In this thesis, the coding style for HLS is being experimented with to find best practices for hardware-oriented C++ and some examples are presented. A digital front end (DFE) IP-block describing a signal process algorithm will be implemented using Siemens EDA Catapult HLS tool and C++ as input language. The IP-block itself has been originally designed in RTL methodology by other designers and this design is used as a reference for evaluating the results of HLS design.

This thesis is done at Nokia in Oulu. Active support from Siemens EDA is applied through the entire design process to guide with both coding style and the HLS tool usage to avoid mistakes and speed up the HLS learning curve since the writer has practically no previous experience in designing hardware in HLS methodology. Some previous trials and studies on HLS have been made within Nokia with reasonable results [3]. These studies have been focusing on different areas and challenges in HLS flow, like HLS-extracted RTL's backend flow compatibility [3] or HLS in IP-based SoC development [4]. Each of these studies includes a case study with comparisons between HLS and RTL designs and the QoR is reported similarly to each other with a little variance.

Chapters 2, 3, and 4 present an introduction to HLS, C++, and Siemens EDA Catapult HLS tool more specifically. Chapter 5 contains hardware-oriented coding principles for C++ and best practices with examples. Chapter 6 presents a case study example exploiting these coding practises and a comparison of results against the handwritten RTL model of the design in terms of QoR and designer productivity. Chapters 7 and 8 contain discussion and summary to provide conclusions and closure to the thesis.

2 HIGH-LEVEL SYNTHESIS

High-level synthesis is a system-level abstraction methodology to generate register transfer level code directly for targeted ASIC or FPGA technology from behavioural algorithmic source code using a specific electronic design automation (EDA) tool. Traditionally digital IC's are designed on a register transfer level of abstraction using hardware design languages (HDL) such as VHDL or Verilog. Writing RTL code, however, is time-consuming and prone to bugs. HLS aims to solve these problems by raising the abstraction level from RTL to system-level C/C++ description of the desired algorithm, of which the RTL code can be generated with the HLS tool by an HW designer guiding the process. [7]

2.1 History

HLS has had three generations, the first of which originates from the 1980s. The first generation was a commercial failure due to specific input language, the lack of need for such technology, poor quality of results, and domain specialization but it offered vital research for the following generations. [1]

In the second generation, from the mid-1990s to early 2000s, major EDA tool vendors offered HLS solutions that made clear progress from the first generation but failed as well due to overmarketing, narrow applications, and various other reasons specified in [1].

The third generation began in the early 2000s. An actual breakthrough was not made but success stories were heard. Tool vendors promised similar Quality of Results (QoR) to handwritten RTL and although most of the papers and case studies reported poor QoR, the designer productivity was undeniably increased significantly. The third-generation HLS tools accept common input languages such as C to ease the adoption of the HLS tools. The vendors identified a suitable market and therefore, HLS tools focused on data-driven DSP applications as they are more suitable to those than complex control applications. [1]

The ever-increasing complexity of ASIC and FPGA development increases the need to raise designer productivity. In the past years, HLS has been used and experimented with on a variety of applications and tools have been greatly developed. There is great potential in the technology but still, in recent years it has been reported that some synthesis tools may seem like black boxes to designers and it might be unclear what happens in the synthesis. [2]

HLS tools have been even further developed in recent years. Today, the QoR gap is nearly closed and HLS is being adopted by a wider variety of users. Open source MatchLib library by NVidia is also enabling a wider variety of applications for HLS as well as encapsulating some optimization details with preoptimized components that are commonly used in HW design [8].

2.2 Research on previous HLS use

There are various case studies about HLS being applied to different designs. They have been analysed from different points of view for example some compare HLS design QoR against hand-written RTL QoR and some analyse software developer's ability to adopt hardware design with high-level synthesis.

Reference [9] compiles 46 different case studies from 2010 to 2016 with acceptable reporting numbers comparing QoR and designer productivity. The main conclusion is that QoR with HLS remains marginally lower than hand-written RTL designs, but designer productivity is highly increased, the development cycle is shorter, and the lines of code number is lower using HLS than traditional RTL design flow. With some exceptions, of course.

When comparing QoR in terms of performance (depending on the application) and resource usage of HLS and RTL designs it seems that results are very similar although RTL slightly outperforms HLS. The big picture indeed suggests that the QoR gap between HLS and RTL would be closing and modern HLS tools generate RTL code with similar quality as handwritten, when used properly. [9]

Designer productivity is compared in terms of development time and lines of code. 25 case studies reported development time. In 72 % of the cases, the development time in HLS is 50 % or less compared to development time in RTL design. Only three of those studies report a longer development time for HLS than RTL. All three come from the same work [10] and the reason as they explain it is the learning of the HLS tool and necessary source code modifications to meet the performance requirements. [9]

35 of the 46 case studies reported lines of code for both RTL and HLS. In 75 % of the reported cases, the number is lower for HLS than RTL. In small designs the nonbehavioural part of the source code in HLS designs becomes dominant and it favours RTL design in this sense. In all the cases where this number was reported larger in HLS design, the total lines of code is less than 250. According to these numbers, it seems indeed undeniably true that HLS remarkably increases designer productivity. [9]

The development time ratio between HLS and RTL projects was plotted as a function of absolute development time to figure out if HLS benefits are upscaling in large-scale designs. Interestingly they found no such correlation. The ratio development time in both large and small-scale applications remained approximately the same [9]. One could imagine that HLS benefits from large-scale applications due to code reusability, verification effectiveness, and scalability of code, especially when the object-oriented programming (OOP) paradigm is used. However, the ratio of lines of code seemed to favour HLS more and more as the design size increased [9].

The designer experience in RTL and HLS is neglected in the case study compilation [9]. In such trials where HLS and RTL are being compared the setting commonly is that it is either software developer or hardware designer that creates the designs compared. In the case of experienced hardware designers, RTL design is familiar and it is easy to create high-quality RTL, but in terms of HLS, the lack of knowledge about the input language and HLS tool might be the limiting factor. Software developers, on the other hand, most probably have no problem describing the algorithm in given HLL but translating that algorithm to hardware with the synthesis tool might lead to poor results if no specific RTL architecture is targeted.

There is also a case study on the compilation paper to verify the results of the quantitative research. In this case study, six people with moderate SW development experience and little experience in HW design implement a DCT algorithm in HEVC encoder with both HLS and RTL methodologies. The results suggest that HLS design flow is easier to adopt than traditional RTL flow. This point of view is not too interesting for this thesis though, because the aim is to produce production-ready RTL with the highest possible QoR with increased design productivity. [9]

2.3 Benefits of HLS

When adopted, HLS would augment traditional RTL design flow, so pros and cons should be in reference between these two methodologies. More specifically in this thesis, the target is to produce production-ready high-quality RTL code for ASIC technology with increased productivity. The benefits will be mostly discussed from this point of view and not for example HLS vs RTL adoptability for newcomers in HW design.

2.3.1 Productivity

HLS promises to increase designer productivity [10]. This is one of the biggest and most remarkable factors in HLS adoption as modern digital IC projects take large teams and up to several years of development time before a commercial product is ready.

HLS takes hardware design to a higher level of abstraction allowing the designer to focus on the behavioural model of the desired hardware. Writing higher-level code simply means less code is needed to describe an algorithm which again means the code can be written faster. Of course, writing the code is not the whole process but a big part of it. Higher-level code is usually more intuitive and therefore, faster to write and easier to understand.

HLS abstracts away some details from the designer like finite state machine creation and interface handshaking. These details in RTL design must be written to the source code in each place they are desired, basically leading to code duplication. HLS fully automates some of these features, and if not, the designer still has the power to make modifications in the synthesis tool to control the generated RTL code. This is arguably easier and more intuitive and thus increasing productivity.

2.3.2 Bug freedom

As HLS design flow hides some details from the designer and less source code is required to implement the desired functionality the source code is less prone to bugs. As long as the source code compiles and does not conflict with the HLS-friendly coding style, the tool is supposed to generate bugless RTL with identical functionality to the source code.

Algorithmic bugs in the source code functionality are also easier to identify and fix in HLL than RTL. Code reusability allowed by HLS and OOP ensures that bugs are not duplicated in the code. Reusing a buggy class multiple times in the design exposes the bugs more effectively and they only need to be fixed once, unlike in duplicated code. [10]

2.3.3 Verification

Verification in traditional RTL flow can take the majority of the time before the product is ready for publishing for the market. Complex digital IC has so much functionality it may take thousands of test cases to reach near 100 % RTL coverage and functional coverage.

In traditional RTL design flow verification is commonly done with simulations. In large RTL designs, this can be very slow as running a single test case can take from minutes to hours depending on the design and test case complexity. Long regression runs might take up to weeks to finish. RTL simulation can only be started when a functional version of the design is completed.

HLS allows the testbench to be created in the same HLL as the design is originally written in. For example, C++ simulations run over a hundred times faster than RTL simulations [10] exposing bugs much faster. The behavioural model of the design can be verified when the source code is written, and no RTL is needed for this. Therefore, the verification process can be started earlier, and the bugs can be fixed faster. Once the RTL is generated, high coverage test cases can be copied to the UVM test environment and run for RTL to quickly achieve high RTL coverage. This way, shorter regressions can be run for RTL and individual UVM test cases can be created to fill the coverage gaps.

HLS tools have built-in verification tools, such as Siemens EDA Catapult's SCverify. SCverify is a hardware designer smoke test environment that automatically creates a testbench and verifies the equivalence of the outputs of C++ source code and generated RTL simulations. There are still some structures in the source code that SCverify has trouble with and traditional RTL simulations are needed to verify the correct functioning of such structures, but there is usually a workaround to verify the functionality of such structures in SCverify as well.

2.3.4 Scalability

Using OOP in HLS design allows flexible code reusability. Using templatized classes allows modules of similar functionality to be reused with different kinds of interfaces or data types, for example.

Small structures commonly used in RTL architecture like different buffers can be written once and used everywhere. Larger entities like whole IP can also be simply rescaled for different applications. All of this reduces code duplication and increases productivity as designers can focus on creating new instead of modifying old code.

HLS tool also allows easy adjustability of clock frequency and target technology. The same source code can be used when adjusting clock frequency and the HLS tool optimizes the data paths so that the maximum amount of operations is fit into each clock cycle with given constraints. The pieces of timing information in Catapult are only estimates though since the HLS tool has no information about the gate-level structure that is eventually done with RTL synthesis tools such as Design Compiler.

HLS source code is technology-independent [10]. This enables real IP reusability over projects using different technology libraries. Technology retargeting from ASIC to FPGA and vice versa though might require some changes to the source code as the available resources are different.

2.3.5 Source code readability

Easy code readability should always be a goal when writing it. By its low-level nature, RTL code is difficult to read for any other than an experienced HW designer. RTL must be very descriptive and detailed for the compiler and RTL synthesis tool to understand it and create the desired HW.

High-level code is much more intuitive and readable by its nature. Very common programming languages like C++ also help other than HW designers to understand and comment on the functionality of the design. Files describing the same functionality in HLL as RTL are commonly smaller making them easier to read and understand large entities in them. Even though HLS coding style might sometimes even conflict with common coding in SW development, source code readability is also one of the big advantages HLS has to offer.

2.3.6 Design space exploration

DSE or architectural exploration is one of the big advantages in HLS compared to traditional RTL. Whereas in RTL design flow usually the algorithm is defined first, then the desired architecture what is needed in the hardware and finally after having a specific target architecture it is being committed to and the actual RTL-coding work can start. At this point, it should be relatively clear what the outcome should look like. In HLS, as the source code itself is high-level code describing the algorithm and the RTL outcome is defined by that and the tool directives, making changes to the architecture can be done in the late phases of the design flow. Making minor changes to the source code or fine-tuning the tool directives to explore different architectural options and finding the optimal one for the application is called design space exploration or architectural exploration.

Loop handling is one of the features in HLS to exploit parallelism in the C-code. Naturally, C++ does not support parallelism, but the code is executed sequentially which means that lines of code are executed one after another. This applies to loop iterations as well, meaning that a loop iteration is not started before the previous one is finished. In HW this is not always desired as it leads to poor data throughput in the generated RTL. HLS tools allow loops to be unrolled or pipelined. Unrolling a loop means that the loop iterations are executed in parallel, and it can be done for the entire loop or partially [11]. Pipelining a loop means that the next loop iteration is started before the previous one is finished [11]. Loop handling is just one example of design space exploration.

Not all configurations of directives result in optimal RTL. Given a specific set of area and timing requirements, there might be more than one optimal architecture but for sure there are many suboptimal ones. If an architecture has the lowest area possible for a specific latency requirement, it is called Pareto-optimal [12]. Finding the Pareto-optimal architectures can be a complex task especially when targeting an FPGA technology having a specific set of resources available.

DSE is commonly done manually, but research on automatic DSE and the analysis of the results have been made to find the Pareto-optimal results in HLS designs [13]. Targeting ASIC technology, the Pareto-optimal solutions are quite straightforward to find but that is not the case in FPGA [14].

2.3.7 Easy source code modifications

Since the source code is written in a higher level of abstraction in HLS, it allows the designer to make changes to the source code to try different architectural options in the resulted RTL. This is game-changing as in traditional RTL design flow the designer must specify the details of the target architecture before writing the RTL code. Once the architecture is decided and agreed upon the designer must commit to it even if they found out later that there could be a better solution. Writing RTL code targeting specific architecture, change of plan could mean that all the work that has been done was useless. Changing the architecture in the late phase could affect neighbouring IP and other designers as well. HLS allows making changes and experimenting with different architectures even in late phases due to high-level code flexibility and code reusability [10].

2.4 Challenges of HLS

Despite the great promises, there are a couple of main challenges that have slowed down the adoption of HLS.

2.4.1 Tool specificity

By this day there are over twenty high-level synthesis tools, both academic and industrial, most of which use different input language from each other [9]. The most common input language is C or its subset.

Most high-level languages are natively not designed for hardware description and thus poor for that purpose. The challenges of C-like languages as an HLS tool input language have been discussed in [15], but most of the problems described in the paper have been solved by the day. The paper covers twelve different subsets of C used in HLS for different tools and discusses the strengths and weaknesses of each one from the point of view of hardware description. Even though the paper claims that the main reason for C-like languages to be used in HLS is its familiarity, C/SystemC/C++ has become an industry-standard in HLS.

Having such a high number of HLS tools can be confusing for the users. Before getting to design anything, they must commit to a specific tool and possibly learn a new programming language or at least relearn how to write it as HLS sets specific restrictions to most common HLLs.

The common problems in generating hardware from HLL code are known: specifying parallel algorithms, specifying timing, and having proper data types for hardware. As each of the tools solves these problems their way it seems unclear for the designer how the tools work. For example, there are multiple different arbitrary-precision data type libraries for C used by different HLS tools. Which one is the best? Which one should be used?

Comparison between different tools is also difficult. As they use different input languages it is difficult to directly compare the tools' different properties such as synthesis run-times or "*Which tool generates the highest quality hardware?*". Committing to a specific product, adopting, and learning it is a relatively big risk to potential customers.

The lack of unified methodology for HLS limits both hardware designers' desire and capability and the tool developers' ability to create the highest quality tools. Only with standardized methods can designers learn to create consistently high-quality hardware and the tool developers could focus on the designers' needs instead of creating another tool that no one but the developers themselves can efficiently use [2]. Surely, sometimes it depends on the application which tool and input language are the optimal ones, but not twenty languages are needed. A lot of research has been made on the topic during the past forty years and agreeing upon some standards could greatly speed up both tool development and adoption for wider use.

2.4.2 Quality of results

The main challenge slowing down the adoption of HLS is that the quality of results (QoR) has tended to stay significantly lower compared to handwritten RTL. Significantly doesn't need to be very much in this context since the companies producing ASICs are usually willing to invest a lot of time and money to achieve the optimal results in the area, timing, and power consumption, and anything less than that is not acceptable. However, the QoR gap between HLS generated and handwritten RTL-code has been closing in recent years as the HLS tools have been further developed.

Poor QoR of HLS generated hardware is often related to designers trying to blindly translate software algorithms into hardware. HLLs, like C, are often sequential by nature, and the optimal hardware usually exploits high parallelism. The designer must aim to the optimal RTL architecture in the source code writing phase or the outcome RTL will not be optimal. This must be considered to avoid poor results or even failure in the resulted RTL. Having clear target architecture and an idea of how to describe it in HLL can result in comparable QoR with HLS and handwritten RTL [16].

HLS tools often also fail to capture the programmer's intention, leading to poor QoR. This is a matter of the designer and the tool interpreting the source code differently. Describing algorithms on a system-level C/C++ can be relatively simple, but when targeting ASIC or

FPGA the designer must have an idea of what kind of resources are available and how does the HLS tool uses them.

3 C++ AS CATAPULT INPUT LANGUAGE

C++ is one of the possible input languages for HLS. More specifically for Catapult HLS tool that will be used in the case study in chapter 6. This chapter introduces C++ as the input language for Catapult HLS tool. The elements of this chapter will be used in the code examples in chapter 5 as well.

3.1 Overview of C++

C++ is a multi-function programming language. C++ is an extension of C as it adds classes and object-oriented paradigm into C. C++ is compiled, strongly typed unsafe language. C++ requires the programmer to know what he or she is doing but allows a lot of control while doing that. [17]

C++ was created by Bjarne Stroustrup. It was published in 1985 with *The C++ Programming Language* book [18] as Stroustrup's reference to the language. In 1998 C++ was standardized as C++ ISO/IEC 14882:1998, informally known as C++98. In 2011, C++11 was released, and this standard is also used in the case study [19].

C++ does not have a concept of time. Traditionally C++ programs target standard CPUs assuming there is an operating system allowing system calls and all operations are executed sequentially. Native C++ data types such as int and char have fixed sizes of 8, 16, 32, etc. making them not ideal for HLS as in hardware design it is desired to optimize the resource usage.

3.2 Object-oriented programming

C++ supports object-oriented programming (OOP) paradigm. OOP means that the data is processed in the code as objects. All objects are instances of classes that have certain attributes or member variables and methods or member functions [20].

One example of a class could be a vehicle: it has traveller capacity and speed as member variables and turn right, turn left, accelerate, and decelerate as member functions. Classes work as type definitions in the programming language so now it is possible to instantiate multiple different vehicles with different member variables. To further specify that the vehicle is a truck, a class truck is created which has all the attributes and methods that the vehicle does, but they don’t need to be rewritten as OOP offers a feature called inheritance. Class truck inherits from the class vehicle and it can have some truck-specific attributes, like storage capacity for example. Another class could be a wheel. A Wheel has a diameter as a member variable and roll as a member function. A truck instantiates four wheels as member variables, optimally with the same diameter. The concept of objects inside objects is called object composition in OOP. A bicycle class would inherit from the vehicle, it would have its member variables and functions and it would instantiate two wheels, etc.

Encapsulation means that both member functions and variables are declared as private or public whether the programmer wants them to be visible outside the class or not. A common practice in SW coding is that member functions are public and member variables are private. If the programmer wants to change the value of member variables from the outside, he or she has to declare a member function that changes the value of the variable for example *set_x()*-function to change the value of *x*.

In OOP, all classes must have a constructor as a member function. Constructor is called when a new instance/object of the class is instantiated. A constructor can be default or parameterized. Default constructor sets default values to member variables and with parameterized constructor programmer can set member variables when instantiating the object.

Figure 1 presents an example class *example_c*. In line 1 there is the declaration of the *example_c* class. Suffix *_c* is fully optional, and it is just a good practice telling the designer that *example_c* is a class and not for example an object. Everything between the wave brackets in lines 1 and 20 is the contents of the class.

Lines 2 and 5 define member visibility outside of the class. Anything declared private can only be accessed from inside the class. The default visibility is private so line 2 is optional but for clarity in this example case, it is written. Common practice is that member variables are declared private and there are specific member functions for accessing those variables.

This class has two attributes, integer type variables *var1* and *var2*. They are declared private. In the constructor starting from line 6, the attributes are initialized to zeros. The class has *set_var1()*-method to assign *var1* to a specific value that it takes as an argument and *get_var1()*-method to extract the value of *var1* to the outside of the class. *var2*-attribute cannot be accessed at all from the outside of this class. This is not very practical and in an actual class like this, there would probably be a similar accessing method for *var2* as there is for *var1*. The *print()*-method prints out the values of the attributes.

```
1    class example_c{
2      private:
3        int var1;
4        int var2;
5      public:
6        example_c(){
7          var1=0;
8          var2=0;
9        }
10       void set_var1(int value1){
11         var1 = value1;
12       }
13       int get_var1(){
14         return var1;
15       }
16       void print(){
17         std::cout << "var1 = " << var1 << std::endl;
18         std::cout << "var2 = " << var2 << std::endl;
19       }
20   };
```

Figure 1. Example class declaration in C++.

3.3 HLS restrictions and additions to C++

OOP suits HLS well since modules in IP design can be similar but have minor differences. Think for example two FIR-filters with a different number of coefficients. OOP in HLS enables the designer not to repeat written code but to reuse it. This makes the design more readable and enables easier bug fixing. Siemens EDA Catapult, which is used in the case study as the synthesis tool, advises the designer to create the designs in class-based OOP rather than traditional function-based code.

Since HLS generates RTL from the C-code that is eventually synthesized to actual hardware, some of the C++ constructs are prohibited in HLS design due to the limited resources in HW. Such constructs are for example dynamic memory allocation, function pointers, and recursive functions [21]. Also, good programming practises in HLS differ a little bit from the common SW practises. In Catapult HLS, all the member variables and functions are declared private except the class constructor and the interface function that must have *#pragma hls_design interface* before it, if used as module interface [11]. If there is no such pragma, the function will be synthesized as logic within the higher level in the hierarchy.

The constructor in the HLS design is interpreted as a hardware reset [22]. Therefore, there must be only one constructor in an HLS targeted class. All the member variables must be reset in the constructor body if that is desired in the hardware reset. In C++11 it is also possible to initialize a member variable in the variable declaration.

The interface function is treated as a clocked process in HLS design. Interface function has the modules inputs, outputs, and possible controls as parameters, and the module functionality is described in the interface function body. Function calls to classes member functions must occur inside the interface function. Therefore, everything else but the constructor and interface function should be declared private; there is always a chance of user error. [22]

3.3.1 AC datatypes

AC (Algorithmic C) data types is an open-source C++ library that includes bit accurate data types for C++ [23]. Native C++ data types, such as int, float, and char, have fixed bit widths. They can be used for HLS design but usually, it would result in poor QoR as in quite rare cases optimal hardware only uses these bit-widths. For this reason, specific arbitrary-precision data types have been developed for HLS. These bit accurate datatypes ensure that enough bits are allocated for every step of the design for maintaining the desired data accuracy and scale without having any unnecessary bits. Table 1 summarizes the AC datatypes used in the examples and case study in chapter 6.

Table 1. Bit accurate data types used in C++ HLS design in Catapult

Type	Description
ac_int<W,S>	Bit accurate integer signed or unsigned variable.
ac_fixed<W,I,S,Q,O>	Fixed point variable with specified overflow mode, quantization mode, and decimal point location.
ac_complex<T>	Complex type variable with specified numeric type T as real and imaginary part.

Data type ac_int is a fixed-point integer with a bit-width of template parameter *W*. *S* is a boolean type template parameter representing signedness. Value *true* means that the variable is signed resulting in two's complement representation in hardware. [11]

Data type ac_fixed extends the use of ac_int to be able to represent fractional values. Template parameters *W* and *S* work just like in ac_int defining variables total bit-width and signedness. The parameter *I* defines the number of integer bits specifying the radix point location. *Q* and *O* are optional parameters defining quantization and overflow modes. The sign bit is counted as one of the integer bits in ac_fixed. [11]

Data type ac_complex declares a complex number variable with both real and imaginary parts of type *T* given as a template parameter. ac_complex doubles the bit width of T as it stores two values for real and imaginary parts of a complex number. [11]

Example ac_int variable declarations can be seen in figure 2 rows 1 and 2. Now *x*, declared in row 1 can have values that can be represented with unsigned 3-bit variable, from 0 to 7. Template parameter *S* is *false* so all the bits represent the absolute value, but negative values cannot be represented. The declaration of *y* is identical to the *x* declaration except for the template parameter *S* being *true* in this case. The MSB of *y* represents signedness with '1' being negative and '0' positive. Variable *y* can have values between -4 and 3.

Rows 3, 4, and 5 in figure 2 show example declarations of ac_fixed variables. Variable *x2* is identical to variable *x*. Having 3 bits total, 3 integer bits and *false* as *S*, *x2* can represent the same values as *x*. In row 4 *z* is declared as a two-bit variable with one integer and a sign bit. This means that the only integer bit represents signedness and the other bit of the two represents absolute value. The LSB is on the right side of the radix point, so this variable can have values of -1, -½, 0, and ½. Quantization mode is defined as *AC_TRN* meaning truncate. If this variable is assigned with higher precision than it can handle, the LSBs are simply clipped away. Overflow mode is defined as *AC_WRAP* meaning wrap-around. If this variable is assigned with a greater value than it can represent, the MSBs are clipped away as well. These quantization and overflow modes are the default ones so they could have been just left away in this variable and the same functionality is in *x2* and all the ac_int variables.

Row 5 in figure 2 declares another ac_fixed variable *n*. This one has non-default *AC_RND* as quantization mode and *AC_SAT* as overflow mode. *AC_RND* is a rounding mode that rounds up if one bit right of LSB is high. *AC_SAT* is a saturation mode that assigns the variable with the highest absolute value it can have if the variable is assigned with a value that is out of bounds. There are more rounding and saturation modes but using these in HLS code will lead to rounding and saturation logic in resulting hardware so they should be used carefully.

Row 6 shows ac_complex type declaration. The element type of ac_complex is now a five-bit unsigned integer making *c* a 10-bit complex number with 5-bit real and 5-bit imaginary parts.

```
1    ac_int<3,false> x;
2    ac_int<3,true> y;
3    ac_fixed<3,3,false> x2;
4    ac_fixed<2,1,true, AC_TRN, AC_WRAP> z;
5    ac_fixed<3,2,false, AC_RND, AC_SAT> n;
6    ac_complex<ac_int<5,false> > c;
```

Figure 2. Example declarations of bit accurate variables.

AC datatypes also offer a bunch of different functions as well as overloaded arithmetic operators with them since hardware designer typically wants to perform some specific operations to the bit accurate data types that might not be so typical in software design.

Bit select operation is typical in hardware design. It allows individual bit selection from any of the AC datatypes with square brackets (*[]*). Bit selections allow getting bit from AC datatypes or assigning specific bit with a value of '1' or '0'. Bit selection can only be used for single bits and longer bit queue selections there are other functions.

In figure 3 row 1 variable *k* is declared and initialized with a decimal value of ten. The bit-level representation of the value is shown in the commented section of the row. In rows 2 and 3 two boolean type variables are declared and initialized with a value using the bit select operator from *k*. *b0* is assigned with the LSB of *k* being '0' in this case and *b1* is assigned with the MSB of *k* being '1' in this case. The designer must be careful not to select a bit out of the variable bounds as it will trigger an error in the compilation or crash the C++ simulation.

Shift operation is also very common in hardware design. Shifting bits left or right can usually be thought of as multiplying or dividing by a factor of two in decimal representation. Shifting a bit over the MSB or LSB of the variable, however, has different functionality compared to multiplying and dividing. Shifted-out bits get removed and empty bits get padded with zeros, or ones in negative numbers MSB.

In figure 3 row 4 a new variable *k1* is declared and initialized with a value of *k* shifted right by one bit. The commented section of the row shows the bit representation of the *k1* value. The zero in parenthesis is the original LSB of *k* that is shifted out and removed. The MSB is padded with a zero. The decimal value equals five so the original value was divided by two.

In row 5 there is a left shift operation to the same *k* variable and the value is assigned to a new 5-bit variable *k2*. Since there is one more bit available intuition would say that all the bits are shifted left, LSB is padded with zero and the resulting decimal value would be twenty. This is not what happens. Shift operation always returns the same number of bits that is in the original shifted variable. On the right side of the equals-to operator *k* gets shifted left, the MSB is removed and LSB is padded with a zero. After that, this 4-bit value is assigned to a 5-bit variable and the MSB is padded with zeros. The result is shown in the commented section of the row and the decimal value equals four, which is far from the original value of ten multiplied by two.

Row 6 shows the correct way to maintain all bits when performing the same operation. The return value of the left shift is cast to a specific type, in this case being the same type the value is eventually assigned to. Now the MSB doesn't get removed and the resulting decimal value equals twenty, the original value multiplied by two.

In the bit selection section, it was said that there are functions for extracting or inserting bit queues in AC datatypes. These functions are *slc()* and *set_slc()* methods. *slc()*-method takes slice width as a template parameter and an LSB bit as a function parameter and returns specified slice width bit queue starting from the specified LSB of the target variable. *set_slc()*-method takes LSB and AC datatype variable as parameters and inserts given variable to target variable starting from specified LSB. Line 7 shows an example of *slc()* and line 8 shows an example of the *set_slc()*-method.

ac_complex datatype has *real()* and *imag()* to insert or extract real and imaginary parts from the complex datatype. The return value will be the same as the one defined for the ac_complex variable. Lines from 9 to 11 show an example of ac_complex declaration and separate value assignments for real and imaginary parts.

```
1    ac_int<4,false> k = 10;           // 4'b1010
2    bool b0 = k[0];                   // LSB of k
3    bool b1 = k[3];                   // MSB of k
4    ac_int<4,false> k1 = k >> 1;            // 4'b0101(0)
5    ac_int<5,false> k2 = k << 1;            // 5'b00100
6    k2 = ac_int<5,false>(k << 1);           // 5'b10100
7    ac_int<2,false> m = k.slc<2>(1);  // 2'b01
8    k2.set_slc(0,m);                  // 5'b10101
9    ac_complex<ac_int<4,false> > comp;
```

```
comp.real() = 7;                // 4'b0111
comp.imag() = 15;               // 4'b1111
```

Figure 3. Examples of bit-level operations for AC datatypes.

There is also a helper function for AC type array initialization or un-initialization. Init_array function takes a constant or “don’t care” value as a template parameter and the base address of the array and the number of elements in the array to be initialized as function arguments. For example, the constant value could be *AC_VAL_0* meaning that the elements of the array will be assigned to zero. Un-initializing an array can be done by giving the function a “don’t care” template parameter *AC_VAL_DC*. The reason why one would do this is to prevent the generation of initialization logic of array elements stored in large memories that have limited access bandwidth. After doing this in the constructor, the designer must be careful not to read from the un-initialized memory element before writing there as it may lead to unexpected and undesired behaviour. [11]

3.3.2 ac_channel

3.3.2.1 Overview

ac_channel is a special channel data type used in interconnections between leaf blocks. It is essentially a C++ FIFO that is infinite in C++ simulation, but the depth must be defined in Catapult directives to get the correct FIFO size also in synthesized hardware. FIFO depth 0 removes the FIFO replacing it with only wires. The modelling paradigm used in Catapult HLS to model concurrent HW processes in sequential C++ is called Kahn Process Network (KPN) [24] and ac_channels are used as infinite FIFOs between processes. ac_channel can only be used in point-to-point connections. If the same data needs to be routed for more than one block an additional routing block, multiple ac_channels in the driving end of the FIFO or merging the receiving blocks is required. ac_channel is a template class that takes the data type of the channel as a template parameter. [11]

3.3.2.2 Read and write

ac_channels are declared in top-level as private member variables. They are referenced in the interface functions of the leaf blocks to store the data outside of blocks in the channel variables in top-level. ac_channel has member functions *write()* and *read()* for sending and receiving data from the channel. Channel accesses are bandwidth limited so the functions can only be accessed once in a single clock cycle per channel. This must be considered in the coding and kept in mind if the main function is going to be pipelined or not. If this rule is violated and there are multiple channel accesses within a single clock cycle, Catapult will trigger an error claiming that the design couldn’t be scheduled even with unlimited resources.

Using a *read()*-function on an empty channel will trigger an assertion and crash C++ simulation and therefore it should be used with the *available()*-function to make sure that does not happen if the data availability is unknown. *available()*-function should be inside *#ifndef __SYNTHESIS__* condition that has the ac_channel member function since in Catapult synthesis, *read()*-function transforms into a blocking read that stalls until data is available in the channel. *available()* member function takes an integer number as an argument and returns *true* if there is an equal or greater amount of data in the FIFO than the specified integer is. In

RTL synthesis *available()* synthesizes always *true* optimizing away and leaving only the blocking channel read.

3.3.2.3 Non-blocking read and write

If there are multiple ac_channel inputs for a block and the data availability for each channel is unknown or the block must do something even if there were no input data, a non-blocking read can be used. Whereas the normal *read()* function stall until data is available in the channel and returns the read value from the channel, *nb_read()* returns boolean value *true* if a value was read from the channel and *false* if the channel was empty. Normal *read()* doesn't take any arguments but *nb_read()* takes the variable that the value is read to if there is some. This variable must be the same type as specified for the ac_channel. *nb_read()* from an empty channel does not trigger assertion nor crash the C++ simulation.

Non-blocking write is a more rarely needed feature of the channel. As a non-blocking read, it takes the value to be written to the channel as an argument and returns boolean *true* or *false*, if the write was successful or not. In C++ simulation the return value will always be *true* since the channel FIFO is infinite. In RTL where the FIFO depth is defined, the return value might be *false* as well. To verify the correct functionality of the C++ source code, manual *false* must be inserted to the "return value".

SCverify co-simulation and equivalence checking might cause data mismatch errors if the design is using non-blocking reads and/or writes. Manual verification of the data correctness or other verification method is needed in this case. Therefore, the use of non-blocking reads and writes should be avoided in the source code if possible, but in some cases, they might be useful and required features.

3.3.2.4 Arrays and structs

Often it is required to send large amounts of data through an interface or there might be some control signals for individual samples travelling through the entire pipeline with the data. An intuitive way of programming this could be to create an array of ac_channel, each containing its type of data that travels in parallel to keep data in alignment. This is not an optimal way since each ac_channel synthesizes an interface handshaking logic and a FIFO if the depth is not defined to 0. A general rule is that an ac_channel array should not be used for point-to-point connections.

A better way to solve the problem is to create a C++ struct that contains all the data that is known to be tied together and travel in parallel. ac_channel can take a struct as a data type and the amount bits in the struct is directly the sum of bits in the AC data types that the struct contains. This way, all the data gets sent through the same ac_channel, and only one FIFO or set of wires and handshaking logic is generated, saving area and complexity.

3.3.3 File structure

In traditional C++ programming, it is common that header file includes forward declarations of classes and functions and other data shared by multiple different source files. That practice can be used in HLS as well, but it is also possible to declare a class per header file and all the information that is needed by that class. This is just a matter of designer preference. In this thesis and the following examples, the latter one will be used.

Figure 4 presents an example header file content. This is a very simple example of a class that only writes the data from the input port to the output port when the interface function is called.

Lines 1, 2, and 23 perform a guard preventing multiple inclusions of this class. The first time this class is included *_EXAMPLE_* string is defined, and possible further inclusions are blocked by the *#ifndef* condition. The string must be unique inside the project. After this, there is an empty space for possible file inclusions or constant declarations. This file could include a common type definitions file as the *DATA_TYPE* used in line 14 in this file is not declared at all in this file. The first two lines and the *#endif* at the end of the file can be replaced with *#pragma once* at the beginning of the file to achieve equivalent functionality.

In HLS design good practice is that all the member variables and those member functions that are not the interface function should be declared private. This is not mandatory, but the variables and functions should not be accessed from the outside of the class, for example, testbench so there is no reason to declare them public. If they are declared private, they will trigger an error in compilation if misuse is occurring. Since this example is very simple, there are no private members in this class.

Only public members can be accessed from the outside of the class. As mentioned before, only the class constructor and exactly one interface function should be declared public in HLS designs. Constructor, found in lines from 9 to 11, is called when a class is instantiated inside another class or in the testbench. The constructor initializes member variables with default values. Catapult only supports the default constructor, meaning no arguments can be given to the function since it is interpreted as a reset in hardware. Default values in the constructor are the reset values to each member.

The interface function in line 14, specified by the *#pragma hls_design interface* directly above, defines what will be synthesized to hardware. It takes the interfaces, input, output, and possible controls, as function arguments and describes the processing inside it. In this case, the interfaces are ac_channel type *din* and *dout* containing *DATA_TYPE* type data.

Lines from 15 to 17 make sure of the correct functionality of the class in C++ simulation. *__SYNTHESIS__* macro is not defined in C++ simulations, so the condition makes sure there is enough input data available in the channel before trying to read it. Read from an empty channel would crash the C++ simulation. Line 16 makes sure there is always at least one sample in the input channel before trying to read it. The *available()* function always synthesizes to *true*. When the design is being synthesized the lines of code within *#ifndef __SYNTHESIS__* condition are ignored.

In line 19 there is the entire functionality of this class. When the interface function is called and if there is data in the *din* channel, one sample is read from the *din* and written to *dout.*

```
1     #ifndef _EXAMPLE_
2     #define _EXAMPLE_
3     // file inclusions, constant declarations here
4     class example_c{
5       private:
6         // member functions and variables declared here
7       public:
8         // constructor
9           example_c(){
10            // member variable initializations here
11          }
```

```
    // interface function
    #pragma hls_design interface
    void run(ac_channel<DATA_TYPE> &din, ac_channel<DATA_TYPE> &dout){
      #ifndef __SYNTHESIS__
        while(din.available(1))
      #endif
      {
         dout.write(din.read());
      }
   }
};
#endif
```

Figure 4. Example contents of a header file containing an example class.

3.3.4 Leaf blocks and top-level

Programming larger designs is best to do by dividing the design into a hierarchy according to a predetermined block diagram. Doing this enables easy code reuse and scaling. Functional blocks, that perform the algorithm or whatever is desired in the design, are called leaf blocks or sub-blocks. Top-level is the class where the object declarations of the leaf blocks and ac_channels are, and no logic is allowed in top-level [11].

Figure 5 shows an example of a small design divided into a few leaf blocks. The signal processing does not make any sense in this example, but the class hierarchy is the point. There are two types of leaf blocks, *block1_c* and *block2_c* classes. Objects of these classes are declared as private members of the *top_c* class, there are two *block1* objects and one *block2*. There are also two ac_channels declared for interconnections between the leaf blocks.

```
#include <ac_channel.h>
class block1_c{
public:
      block1_c(){}
      #pragma hls_design interface
      void run(ac_channel<int> &din, bool control, ac_channel<int> &dout){
            int dinTmp = din.read();
            if (control){
                  dout.write(dinTmp);
            }
      }
};
class block2_c{
public:
      block2_c(){}
      #pragma hls_design interface
      void run(ac_channel<int> din[2], bool control, ac_channel<int> &dout){
            int dinTmp[2];
            dinTmp[0] = din[0].read();
            dinTmp[1] = din[1].read();
21          if (control){
22              dout.write(dinTmp[0]);
23          } else {
24              dout.write(dinTmp[1]);
25          }
26      }
27  };
28  class top_c{
29      ac_channel<int> connect[2];
30      block1_c block1[2];
31      block2_c block2;
32  public:
33      top_c(){}
34      #pragma hls_design interface
35      void run(ac_channel<int> din[2],bool control[2],ac_channel<int> &dout){
36          block1[0].run(din[0], control[0], connect[0]);
37          block1[1].run(din[1], control[0], connect[1]);
38          block2.run(connect, control[1], dout);
39      }
40  };
```

Figure 5. Example of small hierarchical design with three leaf blocks and top-level stitching them together.

As mentioned, ac_channels can only be used in point-to-point connections. Control signals, however, can be routed for multiple different leaf blocks. An example of this in figure 5 is the *control[0]* signal which is routed for both block1's in top-level class in lines 36 and 37. In Catapult, these control signals must be mapped to *DirectInput* and in testbench, they are assumed to be held stable during the tests to ensure no data in the pipeline won't get corrupted since *DirectInputs* have no synchronization logic. [22]

3.4 Templatizing

Templatizing is a great way to enhance code reuse in C++. Being supported by Catapult HLS, DSP blocks and buffers, for example, can be created for different kinds of data with a single template class. A very simple example of a shift register template class is introduced in figure 6. The problem with this kind of template class is that multiple different data types must be supported to get the most use of it. For example, this shift register is desired to be used with AC data types as well as data structs which introduces some challenges for the programmer.

The key to avoiding any compilation errors is the template parameter *isStruct* which, by default, is declared *false*, but if the shift register is used with a data struct it should be manually declared *true*. In the constructor line 7, the *isStruct* parameter is then checked and if it is left *false*, the data type is assumed to be AC data type and the shift register element values are initialized to zero with a loop. *init_array* function cannot be used in this case since the code needs to compile also in case the data type is a struct, but the functionality of the function and an initialization loop are identical.

```
1    template<class DATA_TYPE, unsigned N, bool isStruct = false>
2    class shiftreg_c{
3      private:
4        data_struct<DATA_TYPE,N> reg;
5      public:
6          shiftreg_c(){
7            if (!isStruct){ for(int i=0; i<N; i++) reg[i] = (DATA_TYPE)0; }
8          }
9          #pragma hls_design interface
10         void run(DATA_TYPE din, data_struct<DATA_TYPE,N> &dout){
11           #pragma hls_unroll
12           SHIFT:for(int i=N-1;i>0;i--){
13                reg.data[i] = reg.data[i-1];
14           }
15           reg.data[0] = din;
16           dout = reg;
17         }
18   };
19   #endif
```

Figure 6. An example of a shift register with user-defined depth and data type that can be used with AC data types and data structs.

If a struct is given as data type for the shift register, the example above does not initialize the data at all in the constructor. This comes with an assumption that the struct has a built-in constructor. Figure 7 shows an example of a struct that would work with the shift register. The struct has two ac_int type data fields and a boolean flag variable. In line 5 there is the struct constructor, similar to a class constructor, where all the data fields are initialized. Line 10 defines the struct behaviour when an integer type is cast to it. This must be defined since in the shift register constructor value 0 is cast to the struct. By default, there is no conversion from an integer to a struct so this would trigger a compilation error even though with the struct as data type, line 7 would never be fully executed because of the condition. In this case, nothing is done when an integer is cast to the struct.

```
1    struct myStruct_t{
2      ac_int<8,false> field1;
3      ac_int<8,false> field2;
4      bool flag;
5      myStruct_t(){
6        field1 = 0;
7        field2 = 0;
8        flag = false;
9      }
10     myStruct_t(int){}
11   };
```

Figure 7. An example data struct with constructor and definition of behaviour when an integer is cast to the struct.

Another challenge that templates present is that bit widths inside the template in some cases might vary depending on the template parameters. To solve this issue, a group of helper functions is included in the AC data types. Table 2 summarizes these helper functions.

The first one of these, the *log2_ceil* function, is probably the most useful since it returns the base 2 logarithm value of *N*, which equals the number of bits required to index *N* elements. The difference to the second one, the *log2_floor* function, is pretty obvious; if *N* is a power of two, the functions return the same value, but in case of *N* is anything between these, the two functions round the return value up or down. [11]

The difference between *log2_ceil* and *nbits* functions is the behaviour at the power of two values. For example, if *N* equals 8 is given to both functions, base 2 logarithm of 8 equals 3, so *log2_ceil* returns 3. With three bits (unsigned) it is possible to represent numbers from zero to seven, a total of eight different numbers. However, number eight requires one more bit to be represented in binary format, so *nbits* return value 4.

Table 2. Ceil, floor, and nbits helper functions

Function	Description
`ac::log2_ceil<N>::val`	Returns log2 of N, rounded up
`ac::log2_floor<N>::val`	Returns log2 of N, rounded down
`ac::nbits<N>::val`	Returns number of bits required to represent N

4 CATAPULT HLS

Siemens EDA Catapult is one of the industrial HLS tools currently in the market. Catapult will be used in the case study in chapter 6. This chapter introduces Catapult's main features and functionality to provide a basic understanding of the HLS tool.

4.1 Overview

Catapult is an HLS tool that generates RTL implementations from C/SystemC/C++ high-level descriptions of desired hardware architecture. Catapult was released in 2004 as a C++ synthesis tool mainly targeting data-driven applications in ASIC, where the design receives an input, processes it, and sends an output, such as found, for example, in wireless communications and video coding. By this day Catapult has evolved into an HLS-tool basically for any hardware application targeting both ASIC and FPGA technologies. [10]

Catapult provides several graphical tools such as Gantt-chart viewer, resource viewer, and schematic viewer for easier design analysing and debugging. Newer versions of Catapult also provide the Design Analyzer tool to enable easy analysis of the relation between the source code and generated RTL, which can otherwise be challenging to interpret. [10]

Catapult integrates HLV-flow that engages the verification team early in the design process to enable early bug catching, even before any RTL is generated. Catapult Design Checker tool enables catching bugs, hard to find with C++ or RTL simulation, without a testbench. Catapult Coverage tool provides complete SystemC/C++ coverage metrics and SCverify verifies the equivalence of C++ and RTL simulation outputs with an automatically generated testbench that runs a co-simulation between the two models. [10]

In terms of numbers, Catapult promises up to ten times more productivity compared to hand-written RTL flow. Up to 80 %, less source code is needed enabling easier code writing, reading, and debugging. The verification team's time and effort are saved due to HLV flow adding up to 80 % saving in verification cost. [10]

4.2 Input languages

Catapult supports C/SystemC/C++ as input languages. C++’s class-based hierarchy is preferred over standard C’s function-based code. SystemC is a class library addition to C++ that allows clock accurate timing and parallel operations written directly into the source code [25].

Despite the discussion about C-like languages not being optimal for hardware description [15], SystemC and C++ have become the dominant options for input language in HLS. The benefits of using SystemC/C++ are briefly discussed in [7].

C++ maintains a higher level of abstraction and designer productivity compared to SystemC. SystemC, on the other hand, offers more control to designers with clock definition and increased parallelism in the source code by compromising a bit of the increased level of abstraction in C++. In data-driven applications, it is beneficial to describe the algorithm in fully untimed C++ and compromise a bit of the designer’s control over the details of the final RTL [7],[26]. On the other hand, cycle-accurate timing requiring control logic blocks may benefit from the features brought to the table by SystemC.

4.3 HLS C++ synthesis

What does the HLS tool actually do? How is an untimed C++ system-level description of an algorithm transformed into clock accurately functioning hardware? The main steps before RTL extraction within the synthesis process are compilation, allocation, scheduling, and binding [7].

The first task an HLS tool performs to a C++ source code is the compilation and generation of a data flow graph (DFG). DFG is created by analysing the operations and data dependencies inside the design. DFG contains the data flow through the design and each operation performed in it. DFG will be extended to the control data flow graph or CDFG, which adds edges to the graph representing control states. [11]

Allocation refers to resource allocation in which the operations in DFG are mapped into hardware resources found in the specified pre-characterized technology library. Each operation might have multiple different resource options with different characteristics in area, power, and latency. Catapult chooses the optimal one based on the optimization target (latency or area) for each operation and uses these characteristics in the scheduling phase. The designer can also manually allocate specific operations to resources to affect the results of the final RTL. [11]

Scheduling is the phase where time is added to the design process. Analysing the DFGs operations and the resources, Catapult decides when to execute these operations and adds registers to match the clock edges. The placement of these pipeline registers is fully automated making a design very flexible at aiming for different clock frequencies. To perform the pipeline register placement Catapult needs information about the clock, like frequency and uncertainty. These must be specified by the user, again, in TCL-file or Catapult GUI. It is also possible to manually move operations from a clock cycle to another, but this is not recommended. [11]

The binding phase binds operations to allocated resources with characteristics most suitable for each operation. Variables carrying information over clock edges are bound to storage elements such as registers and memories and variables between operations are bound to wires and buses. Multiple variables with non-overlapping or mutually exclusive lifetimes can be bound to the same storage elements. [11]

After these steps, the RTL code is ready to be extracted from the tool and multiple different reports and graphs are presented for the user to analyse if the RTL is satisfactory or not. [7],[11]

4.4 Design flow

Catapult performs the operations in a specific order. The designer set specific constraints in specific steps within Catapult to guide the tool to generate the desired RTL. The design steps can be seen in the Catapult taskbar in figure 8. The steps are interdependent, so proceeding to the next step is impossible if the mandatory actions in the previous one are left undone, or if an error in the previous step is not fixed. The following sections will cover each step explaining the design phases. [22]

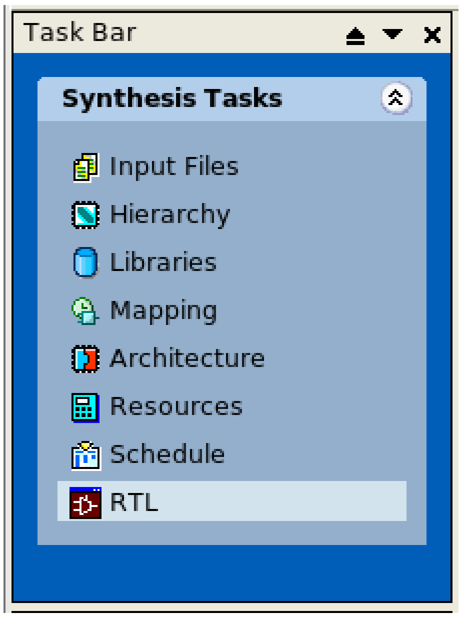


Figure 8. Catapult GUI taskbar.

4.4.1 Hierarchy

After the initial Input Files -step where the input files of the design are defined, the top-level of the design is specified in the hierarchy phase. In this phase Catapult analyses the design and determines which functions are synthesizable blocks. These should be the functions marked with *#pragma hls_design interface* included in the input files. Exactly one of them will be defined as top-level. Other blocks defined as “block” will be their modules and blocks defined as “inline” will be logic inside leaf blocks. These definitions can also be done already in the source code with pragmas for example *#pragma hls_design interface top* right before the interface function declaration. [22]

4.4.2 Libraries

After compiling the design, the technology library can be specified. A pre-characterized technology library includes area and timing estimates for the components that will be used in the design allocation phase. Memory libraries should also be included if they are used in the design. Any previous Catapult solutions can also be added as a technology component to be used. [22]

4.4.3 Mapping

Once the technology libraries have been determined, a clock signal can be specified. Catapult enables setting clock frequency, high time, offset, active edge, and clock uncertainty. All but clock frequency has default values that do not require a setting. Catapult supports multi-clock designs. In this phase, pre-synthesized solutions added in the Libraries-phase can be mapped to the objects if that is desired. Reset signal can be determined as well to be synchronous or asynchronous active high or low. Clock and reset signals can be named in this phase as well. [22]

4.4.4 Architecture

At this point, the entire design has been read into Catapult. The clock and reset structures have been built and the final RTL will be determined by the constraints that will be set in the following steps. Now it is time to evaluate how different objects in the design can be implemented.

The interface types of the block being synthesized can be defined at this phase. Catapult offers different handshaking protocols for channel interfaces. The default protocol is valid/ready handshaking that ensures that both driving and receiving blocks are ready for the data sample, but for example valid only handshaking is available as well. By default, the channels between interfaces are implemented as FIFOs. The length of the FIFOs in C++ simulations is infinite, but the depth of the FIFO in HW is driven by a constraint by the designer. Infinite FIFOs can't be implemented in hardware. If not defined by the user, the default FIFO length is -1 meaning that Catapult automatically determines the FIFO depth, but this is not recommended and often leads to suboptimal values. By setting the depth to 0, the FIFO is removed leaving only the handshaking logic through wires between blocks. Control signals can be set as *ccs_in* with data synchronization but no handshaking or *DirectInput* with no synchronization or handshaking protocol, but in that case, the control values must remain stable during simulations to avoid data corruption in the pipeline. [10]

Arrays in the design can be mapped into registers, RAM, or ROM. The global *MEM_MAP_THRESHOLD* and *REGISTER_THRESHOLD* directives define the default mapping of arrays. If an array has more elements than *MEM_MAP_THRESHOLD*, the array will be mapped into a memory component by default, if such component is available in the libraries inputted to Catapult. If there are no suitable memory components available and array size exceeds *REGISTER_THRESHOLD*, Catapult triggers an error. [22]

It is possible and common practise to map arrays to desired resources independently. Mapping to memory components should be done carefully since memory accesses are limited and might become a bottleneck in the design. It is also possible to further avoid memory access bottleneck with memory configuration directives such as interleave, block size, and word width.

Handling loops is arguably one of the most important steps in HLS. In a rolled loop in Catapult, each iteration takes at least one clock cycle to execute. Loops can be unrolled, pipelined, and/or merged. A pipelined loop initiation interval (II) tells how often a new loop iteration starts. Catapult reports the iteration count of each loop in the source and if that is not satisfactory to the designer, source code must be modified. Changing the number in Catapult won't affect the generated RTL, but only the reported number. [22]

4.4.5 Resources

The resources phase allows the user to specify which resources are used to execute different operations. Catapult shows qualified resources from the technology library for each operation. Catapult allows users to allocate specific resources for operations, limit the usage of some operations or add input or output registers to resource components. [22]

4.4.6 Schedule

Now that the architecture of the design is set with source code and design constraints and the operations are allocated to actual technology components, it is possible to schedule the design.

If Catapult can schedule with given constraints, the schedule will be shown as a Gantt chart where loops, real operations, C-steps, and data dependencies can be seen.

C-steps represent FSM-states and they are roughly equivalent to clock cycles. Within C-step there is a sequence of operations executed in that C-step and by selecting those operations, lines will show the data dependencies to other operations. The schedule can also be adjusted manually if that is required, but this should not be a common practice. [22]

4.4.7 RTL

The RTL phase finally extracts the actual RTL netlist and report files containing information about the outcome, for example, QoR estimates. Automatically generated *cycle.rpt* and *rtl.rpt* files give estimated information about the timing and area properties of the design. The estimates are typically conservative and expected to improve after RTL synthesis. This also depends on the technology library characterization conservativeness though.

Catapult shows comparative data of all the solutions' timing and area properties in the same project. The data can be viewed in a table, bar chart, or XY-plot form. [22]

4.5 Top-down / Bottom-up

Catapult allows using top-down and/or bottom-up flow in the design depending on user preference. Top-down flow means that when the design source code is done, the top-level of the hierarchy is set as the top in Catapult and the whole design is synthesized at once. In bottom-up flow, each leaf block is synthesized separately by setting them top in Catapult, and the actual top level of the design is used to integrate these leaf blocks. [22]

Top-down flow compromises designers’ control over the leaf block implementation details. Possible pros of top-down flow could be constant propagation in some interfaces improving QoR of the design. The downsides of the top-down flow on the other hand are long run times for the synthesis tool and compromising control over the design. Lower blocks of the design must have *#pragmas* in the source code to guide the synthesis process as the details might be unreachable from the top level in Catapult GUI. If changes are needed to the source code after the initial synthesis the whole design must be synthesized again. Also, the amount of platform system memory is limiting the design complexity in top-down flow since the whole design is synthesized at once. In large designs, the top-down flow is not recommended.

Bottom-up flow enables the synthesis of each block separately and using these blocks as technology library components. This means that blocks can be synthesized before the whole design source code is ready. If changes are needed for the source code, only those blocks that are changed are needed to be re-synthesized. On the top level, the latest solutions of the block must be included in the libraries. This saves a great amount of synthesis runtime. The design complexity is no longer limited by the amount of platform system memory as the design is partitioned and synthesized in small sections.

Pre-synthesized solutions must be included in the *Libraries* phase and blocks must be mapped to those solutions in the *Mapping* phase of the synthesis flow to exploit bottom-up flow, otherwise, they will be synthesized as top-down blocks under the current Top defined in *Hierarchy* phase. A mixture of top-down and bottom-up is allowed. [22]

4.6 CCORE

Catapult C Optimized Reusable Entity (CCORE) is a custom operator synthesized from a source code function. CCOREs are defined in the source code as leaf blocks having their class with constructor and interface function declared as public members and everything else as private. CCORE can be identified with the *#pragma hls_design interface ccore* statement before the interface function. CCOREs can be combinational or sequential and they are most practically used as small to medium size functions that appear repeatedly in the design.

CCOREs must only have wire-type interfaces. This means that no handshaking logic is included. Pipelining with initiation interval 1 is required for sequential CCOREs for the data not to stall in the pipeline and not get lost when running the CCORE. Combinational CCOREs must be able to execute within a single clock cycle. [22]

If there is no *#pragma* statement identifying CCORE, a block can be defined as a CCORE in Catapult GUI. Input and output registers can be added for sequential CCOREs if that is desired. Synthesizing a CCORE is like synthesizing a leaf block, it is possible to use top-down or bottom-up flow. Bottom-up flow enables more control over the details of the CCORE since they cannot be accessed in top-down flow. After the synthesis, bottom-up CCORE library components are retained in the project folder and top-down CCOREs in the Catapult cache that is in the user home directory by default. Whenever a top-down CCORE is used, Catapult checks the cache if there is a corresponding component already synthesized and available for use to reduce runtime.

If a sequential CCOREs latency can be statically determined the parent process of the CCORE will wait for the CCORE execution by the amount of max latency clock cycles. If the latency cannot be determined by Catapult, an interface will be generated for the CCORE that tells the parent process when the CCORE execution has finished. [22]

If the designer is not satisfied with the latency and area report estimates given by Catapult, they can be changed with the *DATUM_OVERRIDE* directive. This directive should be used with care, CCOREs with negative slack estimates cannot be used afterwards so it might be useful to set it to zero for further usage since the original report is just an estimate. Final reports can be achieved with an RTL synthesis tool. [22]

5 CODING PRINCIPLES

In order to achieve high QoR with HLS, it is important to follow some guidelines and have an idea of what kind of hardware is generated from the code that is written. This chapter presents some typical code structures and examples demonstrating how to achieve optimal results in them. The examples are generalized and simplified versions of the problems and solutions found in the case study.

5.1 Loops

Loop handling is one of the key features in HLS to tune up the performance of the design. As mentioned in 2.3.6, loops can be unrolled or pipelined to exploit parallelism in C++ code. These can be done either directly in the source code or the HLS tool. For a loop to be pipelined or unrolled, however, certain rules need to be followed to achieve good, expected results.

The first thing Catapult defines in a loop is the iteration count. This should be clearly defined in the source code with strict boundaries since it cannot be changed in Catapult. Dynamic loop bounds should be avoided in HLS code if possible, but an early break from a fixed bound loop is allowed. If it is uncertain if the loop is always going to exit in an early break or not, the iteration count can be minimized for example with an appropriately sized ac_int-type loop iterator. This kind of structure should be avoided as well, if possible.

Figure 9 shows an example code of a simple shift register class. It has *DATA_TYPE* and unsigned integer type *N* as template parameters in line 5. *DATA_TYPE* is the type of data that the shift register contains and *N* the depth of the shift register. The actual shift register variable is declared in line 8 as an *N*-element data struct. The source code of the data struct is presented in figure 10.

In the constructor line 12, the shift register elements are initialized with the *init_array* helper function to zero values. The same functionality could be achieved with a loop that assigns zeros to each element.

In the interface function of the class in line 18, there is a *SHIFT*-loop with decreasing iterator *i* that starts from value *N*-1 and ends at value 1. Naming the loops is not necessary but it is a good practise, especially if there are multiple or nested loops in the code, and it helps to identify the loops in Catapult GUI as they are handled there.

In the loop body, the previous element of the array is shifted to the next one and the last element is overridden or "shifted out". The starting index *N*-1 points to the last element and the index 1 points to the second element of the array since indexing starts from zero and there are *N* elements. When the loop is executed, the input data *din* is saved to the first element of the array and the whole array is written to output data *dout*.

In line 15, directly above the loop, *#pragma hls_unroll* makes the loop iterations execute in parallel. This decreases the latency significantly; if the loop is left rolled each iteration takes at least one clock cycle even though there was slack time for more executions. Thus, the latency of a rolled loop in clock cycles is at least the number of the loop iterations. Unrolling the loop decreases the latency to the latency of the longest iteration in the loop. In this case, where the loop simply moves data from one register to another, the latency on an unrolled loop is one clock cycle. Usually, unrolling a loop would increase the amount of generated logic hardware since there would be fewer opportunities for resource sharing. This case is again exceptional due to the simple nature of the shift register; there are no actual resources to be shared in the first place, so the area of the unrolled loop is expected to be similar, or even smaller, compared to the rolled one's.

The loop in line 18 has *N*-1 iterations. Since *N* is a template parameter, it is seen as constant inside the class and thus, *N*-1 is constant. Therefore, Catapult can define the absolute number of loop iterations and the hardware needed for the unrolled loop design. Since the *N* or *DATA_TYPE* might change in other instances of the class each one of the instances with different template parameters will have different hardware, so they must be synthesized with Catapult separately.

```
1    #ifndef _SHIFTREG_
2    #define _SHIFTREG_
3    #include <ac_fixed.h>
4    #include "data_struct.h"
5    template<class DATA_TYPE, unsigned N>
6    class shiftreg_c{
7      private:
8        data_struct<DATA_TYPE,N> reg;
9      public:
10       // constructor
11         shiftreg_c(){
12           ac::init_array<AC_VAL_0>(&reg.data[0],N);
13         }
14         // interface function
15         #pragma hls_design interface
16         void run(DATA_TYPE din, data_struct<DATA_TYPE,N> &dout){
17           #pragma hls_unroll
18           SHIFT:for(int i=N-1;i>0;i--){
19                reg.data[i] = reg.data[i-1];
20           }
21           reg.data[0] = din;
22           dout = reg;
23         }
24   };
25   #endif
```

Figure 9. Example code of a shift register template class.

```
template<class T, unsigned N>
struct data_struct{
      T data[N];
};
```

Figure 10. Source code of the data struct used in example codes. This struct can be included in a source code with *#include "data_struct.h"* at the beginning of the file.

One important thing when writing a loop that is planned to be unrolled is that data dependencies between loop iterations must be avoided. This may seem obvious, but some dependencies might be difficult to identify in the source code. Fortunately, Catapult provides the tools to identify data dependencies and locate them in the source code.

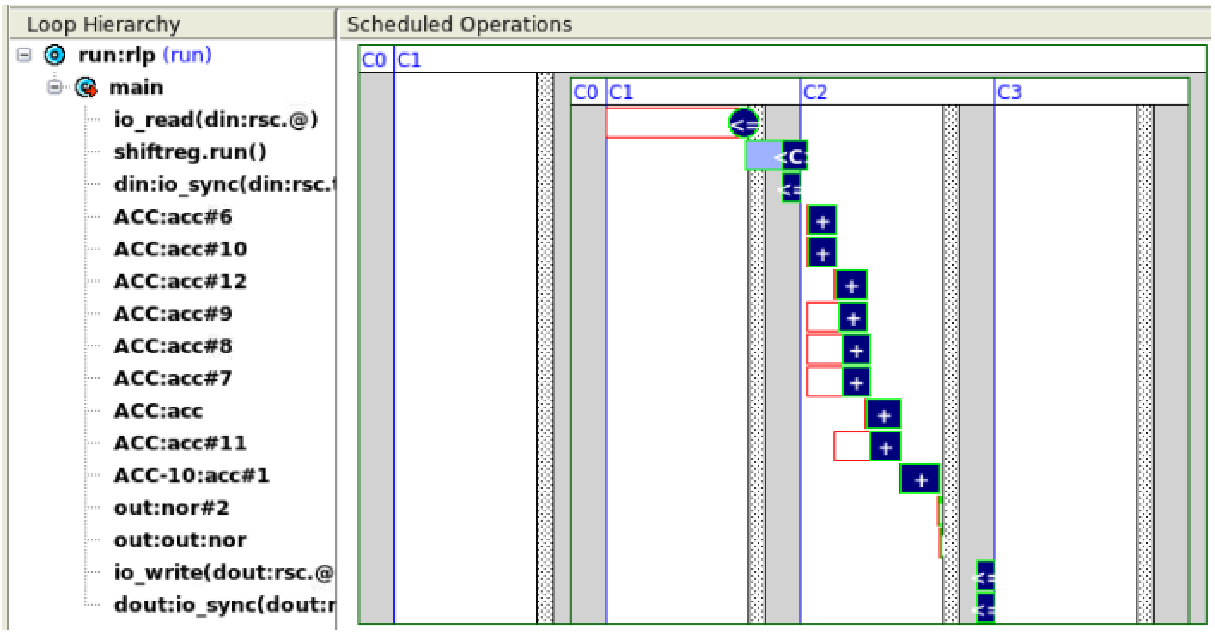
Figure 11 shows an example code of a simple accumulator. The shift register and data struct source codes from figures 6 and 7 are included. Lines from 5 to 7 cover the AC type definitions to be used in the example, *data_t* is the 5-bit signed integer input and *acc_t* is the 9-bit signed integer output data type. *acc_t_RND_SAT* is not used in the following example so it can be ignored for now. Line 9 shows the shift register declaration with template parameters *data_t* and 10. The shift register contains and returns 10 *data_t* type elements.

In the interface function line 14, an appropriately typed variable *arr* is declared for the output of the shift register. This does not need initialization values since the array in the shift register is initialized to zeros and returned. Line 15 shows the accumulator type variable *acc* declaration and initialization. This variable is initialized to zero for every function call and then accumulated with all the data in the shift register. The return value is the sum and the *ACC*-loop is fully unrolled with the pragma.

```
1   #ifndef _ACC_
2   #define _ACC_
3   #include "shiftreg.h"
4   #include "data_struct.h"
5   typedef ac_fixed<5,5,true>              data_t;
6   typedef ac_fixed<9,9,true>              acc_t;
7   typedef ac_fixed<7,7,true,AC_RND,AC_SAT> acc_t_RND_SAT;
8   class acc_c{
9       shiftreg_c<data_t,10> shiftreg;
10    public:
11      acc_c(){}
12      #pragma hls_design interface
13      void run(data_t din, acc_t &dout){
14        data_struct<data_t,10> arr;
15        acc_t acc = 0;
16        shiftreg.run(din, arr);
17        #pragma hls_unroll
18        ACC:for (int i=0; i<10; i++){
19          acc += arr.data[i];
20        }
21        dout = acc;
22      }
23  };
24  #endif
```

Figure 11. Example source code of an accumulator. Shift register source code from figure 6 is included and used in this example.

Since the data types used in this example use the default rounding and saturation modes that clip the out-of-bounds bits away, no data dependencies are created between the loop iterations and everything can be executed within one clock cycle. This can be seen from the schedule of the design in figure 12. Starting from the top left in the *main*-loop, at the end of the C1 step there is the IO-access <=, the shift register CCORE operation <C> and then all the accumulation operations within C2. Finally, the output IO-access is at the end of the C2 step. The adders do not appear to be in parallel in the schedule since Catapult allocates the slowest and smallest resources it can to fit the operations within one clock cycle.


Loop Hierarchy

- run:rlp (run)
  - main
    - io_read(din:rsc.@)
    - shiftreg.run()
    - din:io_sync(din:rsc.
    - ACC:acc#6
    - ACC:acc#10
    - ACC:acc#12
    - ACC:acc#9
    - ACC:acc#8
    - ACC:acc#7
    - ACC:acc
    - ACC:acc#11
    - ACC-10:acc#1
    - out:nor#2
    - out:out:nor
    - io_write(dout:rsc.@
    - dout:io_sync(dout:r

Figure 12. Schedule view of a source code in figure 11 in Catapult GUI. In this case, AC data types with default quantization and overflow modes were used.

The schematic of the design can be seen in figure 13. As expected, there are nine adders instantiated. The large purple rectangle represents the shift register CCORE. There are *ccs_in* and *ccs_out* ports for input and output. The blue rectangles on the right are the output registers, *dout* in the source code, and the blue rectangle on the bottom left represents the FSM logic automatically generated by Catapult. In conclusion, this is a compact RTL design with no unexpected hardware generated by the HLS tool.

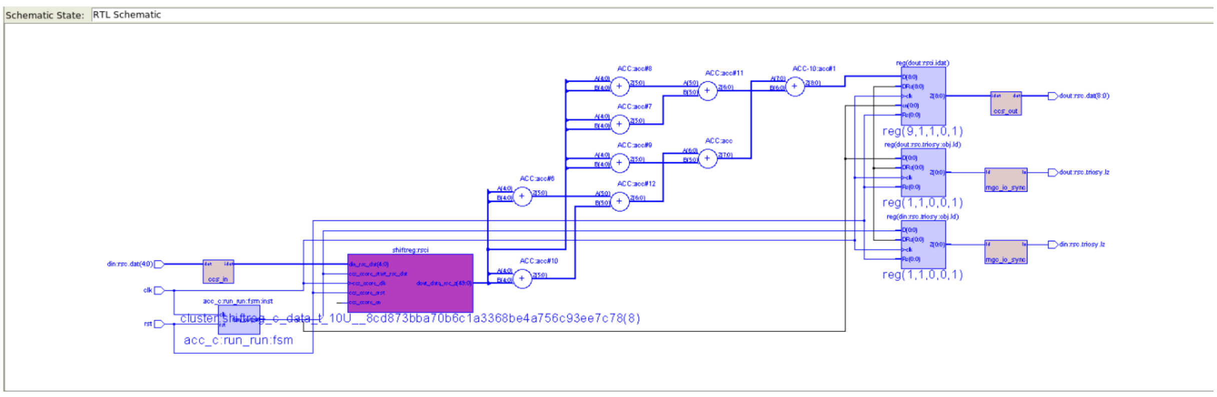


Figure 13. The RTL schematic of the design in figure 8 generated by Catapult.

Now a designer wants to create a similar design but with rounding and saturation modes. For demonstration, here is an example of a bad way to do it.

Let's use the figure 11 source code except for using the *acc_t_RND_SAT* data type for the variable *acc*. Source code is compiled and executed with a testbench to verify that it works correctly. The generation of RTL with Catapult, however, does not result in satisfactory extraction. Figure 14 presents a schedule view of this example.

Comparing this to the previous example, the functionality looks similar, but the accumulation operations are not executed in parallel despite the loop unrolling, and the latency is increased by two clock cycles. By clicking, for example, the adder in the *ACC-5* iteration, the data dependencies can be seen. Each of the operations requires the previous operation to be finished in order to perform the computation. The reason for this is the fact that the data type of *acc* performs rounding and saturation in each iteration.

It can be seen from the schedule that Catapult tries to unroll the *ACC* loop. As mentioned before, if a loop is left rolled each iteration will take at least one clock cycle. In this case, the loop would take ten cycles to finish whereas now it takes three. Since one adder doesn't take the whole clock cycle Catapult fits as many of them to one as possible.

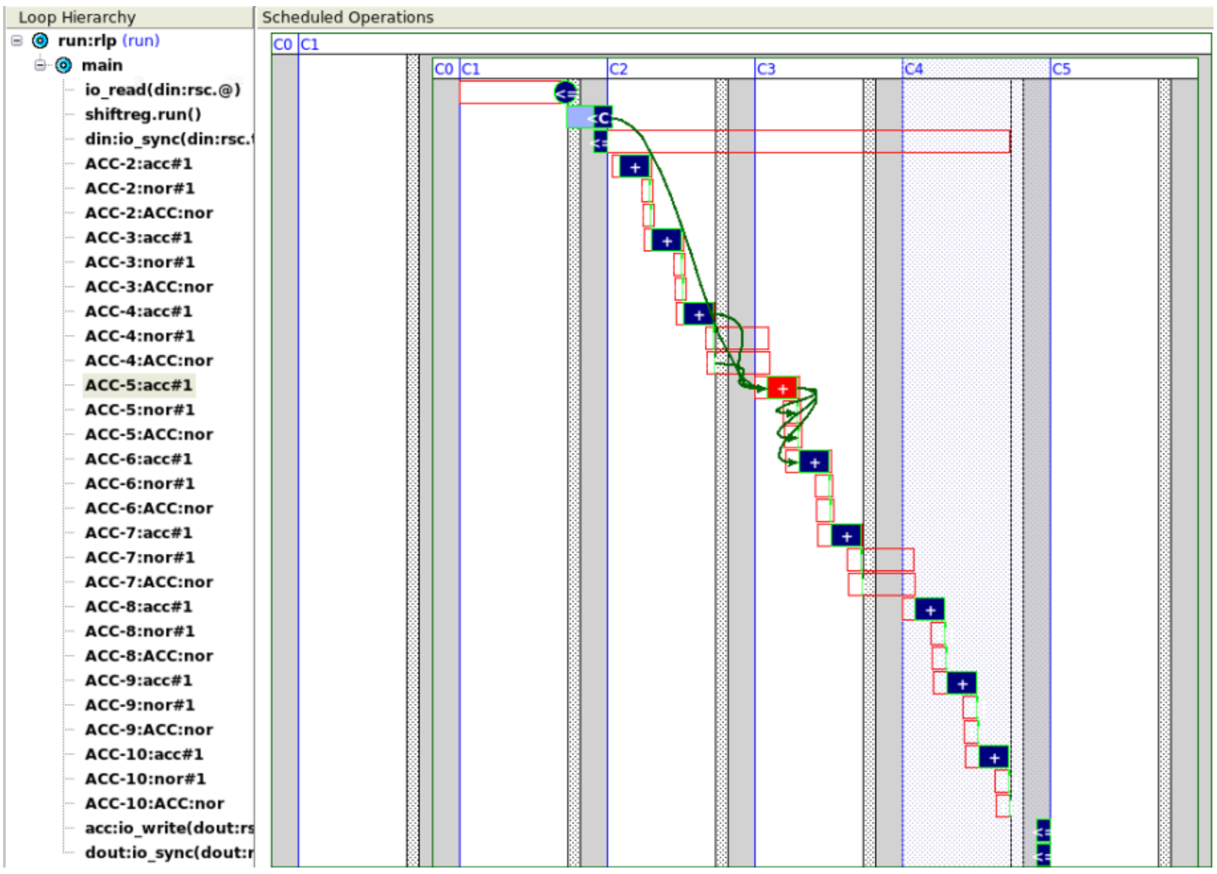


Figure 14. Schedule of the source code in figure 11. In this example, the *acc_t_RND_SAT* data type is used for the variable *acc*.

The RTL schematic of this example can be seen in figure 15. It is far more complex than the one in the previous example since the saturation logic is generated for each adder. However, since the loop is completed within three clock cycles instead of one, resource sharing opportunities are opened, and Catapult automatically generates optimal or as good as possible RTL from the non-optimal source code. Only three adders can be found from this schematic. Nevertheless, this is not an optimal design nor the expected hardware in this case. Non-default quantization and overflow modes should never be used within unrolled loops to avoid unexpected data dependencies. Note that rounding logic is not generated even though it is defined for the variable since all the data types are integers and therefore no rounding can occur in this case and Catapult can optimize it away.

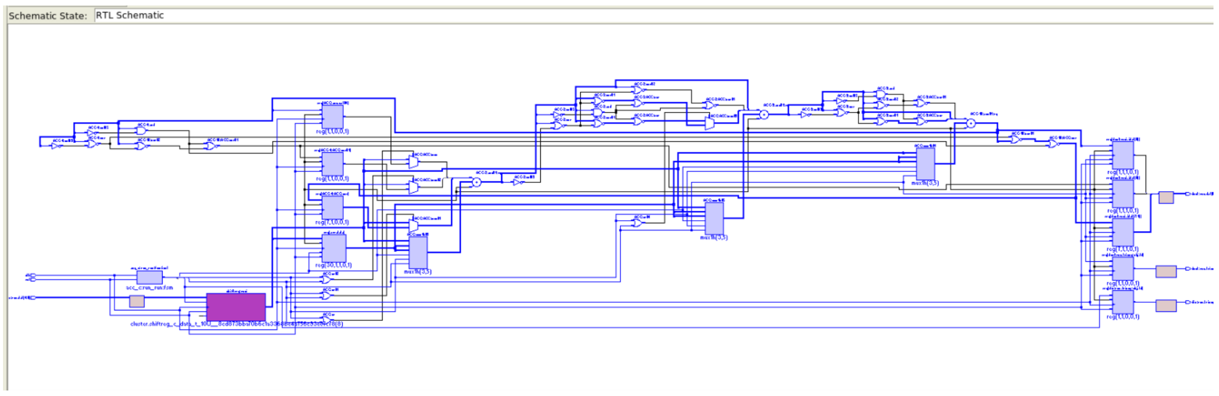


Figure 15. The schematic view of the accumulation example with rounding and saturation within the loop.

Figure 16 presents a better way to perform rounding and saturation in this kind of case. Now the default quantization and overflow modes are used within the loop and it is made sure that there is enough bit-width for not losing any data during accumulation. A temporary variable is declared after the loop with the desired bit-width, quantization, and overflow modes. The accumulated value is then assigned to this variable to perform rounding and saturation. This way, the quantization and overflow logic would be generated only once and no data dependencies between loop iterations would occur.

```
#include "shiftreg.h"
#include "data_struct.h"
typedef ac_fixed<5,5,true>              data_t;
typedef ac_fixed<9,9,true>              acc_t;
typedef ac_fixed<7,7,true,AC_RND,AC_SAT> acc_t_RND_SAT;
class acc_c{
    shiftreg_c<data_t,10> shiftreg;
  public:
    acc_c(){}
    #pragma hls_design interface
    void run(data_t din, ac_int<7,false> &dout){
      data_struct<data_t,10> arr;
      acc_t acc = 0;
      shiftreg.run(din, arr);
      #pragma hls_unroll
      ACC:for (int i=0; i<10; i++){
        acc += arr.data[i];
      }
      acc_t_RND_SAT tmp = acc;
      dout = ac_int<7,false>(tmp);
    }
};
```

Figure 16. An example with good coding in the accumulator with rounding and saturation.

As can be seen from the schedule in figure 17, the execution takes only one clock cycle and the schedule is nearly identical to the schedule in figure 12 since the saturation logic takes a very short time to execute. Figure 18 presents the schematic of the design, where nine adders can be identified as well as only one instance of the saturation logic, as desired.

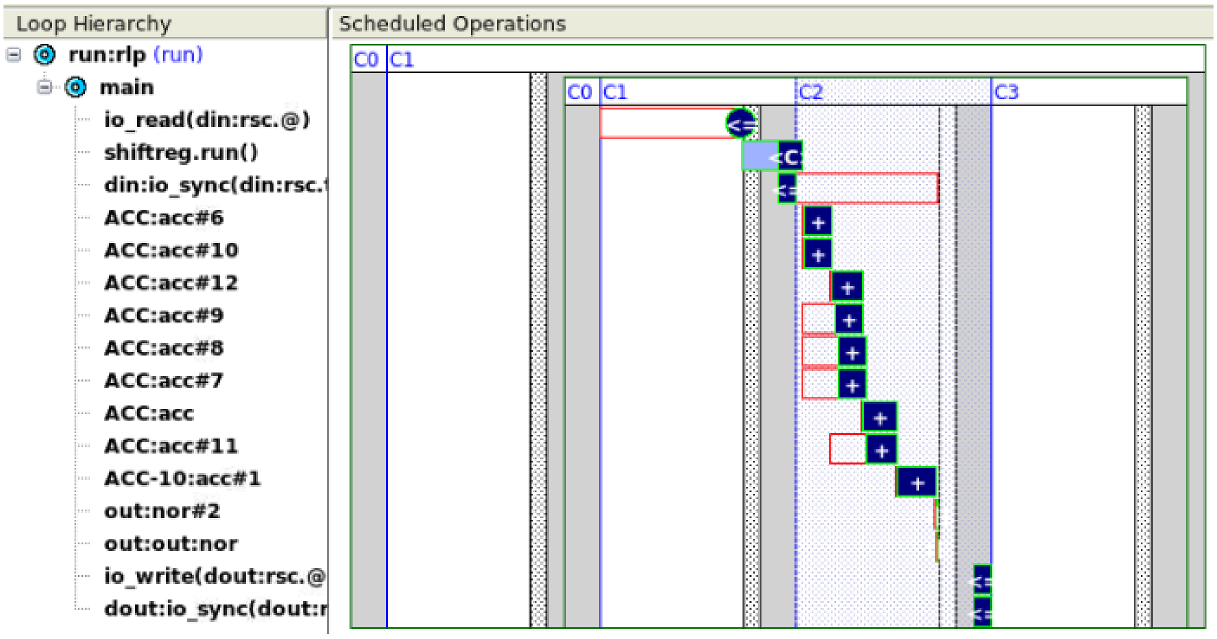


Figure 17. A schedule view from an accumulator with saturation and rounding logic with good coding.

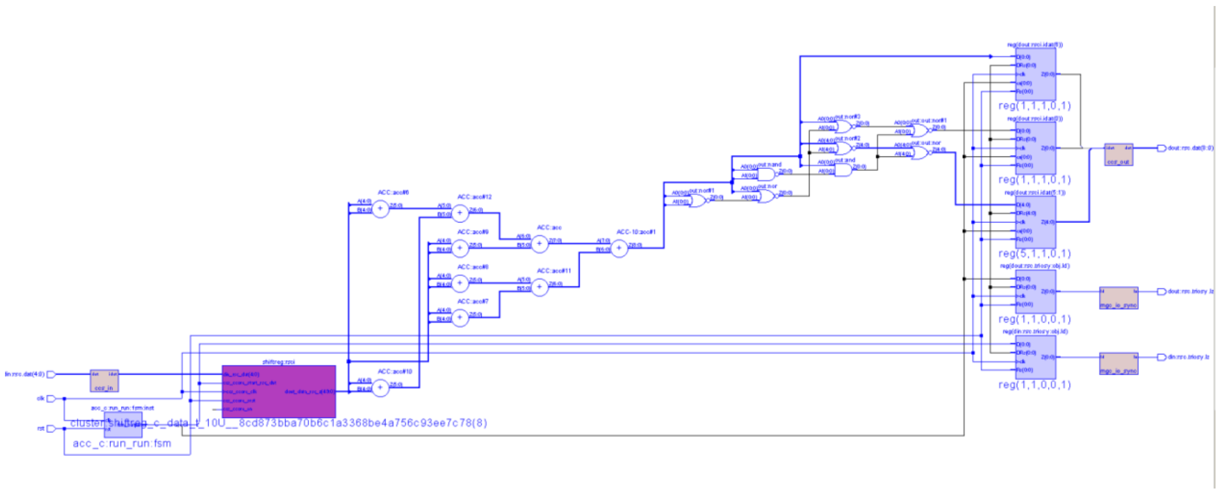
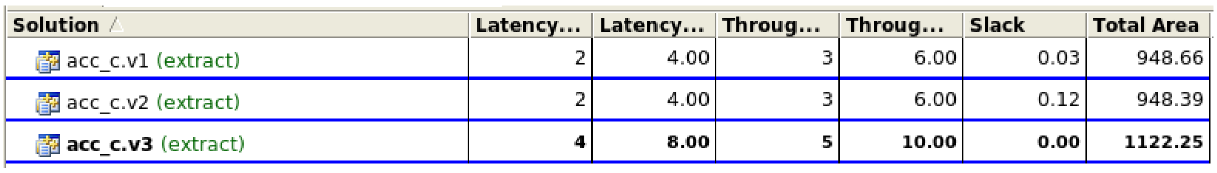


Figure 18. A schematic view from an accumulator with saturation and rounding logic with good coding.

Figure 19 compiles the area and latency estimates of the examples. *acc_c.v1* represents the latest example with an accumulation loop followed by rounding and saturation. *acc_c.v2* is the first example with no rounding nor saturation and *acc_c.v3* is has the rounding and saturation within the loop.



Solution	Latency...	Latency...	Throug...	Throug...	Slack	Total Area
acc_c.v1 (extract)	2	4.00	3	6.00	0.03	948.66
acc_c.v2 (extract)	2	4.00	3	6.00	0.12	948.39
acc_c.v3 (extract)	**4**	**8.00**	**5**	**10.00**	**0.00**	**1122.25**

Figure 19. Table view from Catapult GUI showing the area and latency estimates of different accumulator solutions.

As expected, *acc_c.v3* has two clock cycles longer latency than the other ones. It also has a remarkably higher area despite resource sharing. *acc_c.v1* and *acc_c.v2* have a very minimal difference in terms of timing and area.

The same technology library and clock settings were used for each solution in this demonstration.

5.2 Conditions

Typically, Catapult can share components in mutually exclusive branches of the code. Such mutual exclusivity can be caused by if-else or switch-case statements. In general, Catapult is good at finding mutual exclusivity in the source code, but poor coding practices result in poor hardware and sharing opportunities.

It may be easy to write complex nested conditions and see the mutual exclusivity by analysing that code, but the key to achieving high QoR is keeping the code simple so it is easy to analyse for the synthesis tool as well. If it is possible, resource sharing should be written in the source code with conditional input variables to a function call instead of a conditional function call. A simple example of this is presented in figures 20 and 21. The functionality in both interface functions is identical: there are two integer type inputs and a boolean control signal that chooses one of the inputs to be raised to the power of the two and assigned to the output. It is worth mentioning at this point that this example is so simple that both source codes may result in identical hardware, but the coding practices demonstrated should be applied to more complex situations where they make a difference.

```
1     class pow2{
2           int pow2(int x){
3                 return x*x;
4           }
5           public:
6                 pow2(){}
7                 #pragma hls_design interface
8                 run(int din1, int din2, bool control, int &dout){
9                       int tmp;
10                      if(control){
11                            tmp = pow2(din1);
12                      } else {
13                            tmp = pow2(din2);
14                      }
15                      dout = tmp;
16                }
17    };
```

Figure 20. Example source code of bad coding in a mutually exclusive function call.

In the first example, there is an integer type temporary variable initialized to zero in line 9. If the input control is ‘1’, *din1* is raised to the power of two and the return value is assigned to *tmp*. If the control signal is ‘0’, the same thing is performed to *din2*, but never to both. The operations are mutually exclusive. Now that there are two separate conditional function calls the designer is relying on Catapult’s ability to prove mutual exclusivity in the code and generate only one instance of the multiplier logic. This example of mutual exclusivity is as simple as it gets so Catapult could probably handle it.

Figure 21 presents a better alternative for the interface function in figure 20. The rest of the class is assumed identical to the class in figure 20, only the interface function is replaced with a new one.

```
#pragma hls_design interface
run(int din1, int din2, bool control, int &dout){
      int tmp;
      if (control){
            tmp = din1;
      } else {
            tmp = din2;
      }
      dout = pow2(tmp);
}
```

Figure 21. A better coding interface function example for the class in figure 20.

Since the designer knows that the function calls with different inputs are mutually exclusive, the mutual exclusivity can be written in the source code so the resulting hardware’s QoR is not depending on Catapult’s ability to prove mutual exclusivity. In figure 21, only the value stored in *tmp* depends on the control signal, and the *pow2()*-function is called only once with the *tmp* as input, so it is certain that the synthesis tool only generates one instance of the multiplication logic.

What is demonstrated above with a function call, applies to return statements. Multiple conditional return statements are allowed by both C++ and HLS in general, but they are not recommended. If a conditional return statement is mapped to an IO-component and Catapult cannot prove mutual exclusivity, the main loop can't be pipelined with II=1 since only one IO access per clock cycle is allowed. A general rule of thumb is that each function should only have one return statement and the number of function calls in conditional programs should be minimized.

5.3 IO and memory accesses

IO and memory access tend to be the bottleneck in the design quite frequently since they are limited in bandwidth. Therefore, the designer must be careful when programming these interfaces and know exactly what is desired, and what kind of hardware is expected from the written source code.

5.3.1 Pass by value

Inputs can be passed by values to functions. What this means in hardware is that the inputs are read into registers inside the hardware module at the beginning of the function call [10]. The values outside the function might change outside the function but that does not affect the function itself. When the function is called again, new values are read into input registers and the function executes with these values.

5.3.2 Pass by reference

Inputs passed by reference are not read at the beginning of the function call. They are not stored in registers inside the hardware module by default, but they are read from the interface when they are used inside the function. This requires that the inputs are kept stable during the function execution to ensure correct functionality. [10]

Arrays in interfaces are references, so they are not read inside the hardware module at the beginning of the function call, but when they are used instead. When dealing with arrays in the interface it is important to make sure with a proper coding style that multiple unmergeable array accesses are not generated within the same clock cycle. It is possible to manually read an array from the interface at the beginning of the function if it is not clear that the array values are held stable during the function call. This way the array accesses are merged through the interface and there are no restrictions in accessing the array within the function anymore. This generates another array to the hardware inside the function which might not be optimal especially if the array is very large.

The outputs of a function should always be passed by reference, so the function can manipulate the external variable. Also, ac_channels are always passed by reference in the interface and accessed with desired *read()* and *write()* functions. ac_channels themselves are declared at the top level to connect different sub-blocks.

5.3.3 Memory access

The designer must be decently careful while when writing memory accesses in the source code, but when properly written, Catapult offers a lot of flexibility with different directives. Figure 22 shows a coding example of memory access, where both memories *memA* and *memB* are accessed twice in one clock cycle (indices 0+*i* and 32+*i*). If it is desired to pipeline the design with II=1 this obviously cannot be scheduled with the default settings since only one memory access per clock cycle is allowed.

```
class memoryExample_c{
  public:
 memoryExample_c(){}
  #pragma hls_design interface
  void run(uint8 memA[64], uint8 memB[64]){
    MEM_LOOP:for(int i=0;i<32;i++){
      memB[i]    = memA[i];
      memB[i+32] = memA[i+32];
    }
  }
};
```

Figure 22. Memory access example with multiple memory accesses per clock cycle.

To make the scheduling possible, the memories can be split into a total of four memories with the *BLOCK_SIZE* directive set to 32. If consecutive indices were desired to be accessed from the memories at the same clock cycle the *WORD_WIDTH* directive could be set to 16 to merge the two memory accesses and generate two 32*16bits memories or *INTERLEAVE* directive could be set to 2 to generate four 32*8bits.

In more complex situations all these directives can be mixed to find the optimal solution.

5.4 Multidimensional array access example

A specific but often used and interesting example of HLS generated RTL working exactly like the original source code syntax is when accessing a multidimensional array that has a non-power of two dimensions in C++.

In C++, if an index in a multidimensional array goes out of bounds of the inner dimension, it points to the next index of the outer dimension. An example is shown in figure 23. A two-dimensional array called *array* has 10*10 elements. Let's assume some data is stored in this array and is then assigned to variables *data0* and *data1*. *data0* accesses element *[0][10]* of the array, which is out of bounds by one since the indexes of a 10-element array go from 0 to 9. In C++ this is completely legal, and the return value assigned to *data0* is the first element of the next dimension.

```
int array[10][10];
int data0 = array[0][10];  // indexes go from 0 to 9, 10 is out of bounds
int data1 = array[1][0];   // the same element of an array as [0][10]
```

Figure 23. Example presenting how multidimensional array indexing works in C++.

Catapult HLS generates RTL that works exactly like the source code syntax, meaning that this feature is also inherited in C++ HLS. If the array dimensions are factoring of two, this is not an issue since the index variables pointing to the array have a fixed number of bits and if sized correctly, out of bounds of the array is also out of bounds of the variable range.

If the array dimensions are not factoring of two like in the example above, this feature can cause problems to an ignorant designer. For indexing a 10-element array, one needs a 4-bit unsigned variable that covers values from 0 to 15. Values from 10 to 15 in the inner dimension would not be used, but according to C++ syntax, they would be accessing specific elements causing undesired multiplexing in the generated hardware. In the outer dimension, the effect would be even worse since indexing the array with the values from 10 to 15 would cause undefined behaviour and therefore, unexpected hardware.

Figure 24 presents an example of bad coding practice in both non-constant array indexing and multidimensional array accessing. The class contains a ten-by-ten array as a persistent variable and it functions as a delay line for ten data channels. The channel is chosen with the input *ix* and the input *din* is saved in the first element of the corresponding channel. Every function call, elements in channel *ix* are shifted forwards and the last element of *array[ix]* is returned to the output. In C++ this works perfectly if it's known that from outside this class *ix* can only have values between zero and nine.

However, Catapult does not know which values of *ix* are used, so it generates the RTL functionality for all the values since the range is not limited in any way. Another issue with this code is that *ix* is not a constant and therefore its value is checked individually in each loop iteration. This leads to data dependencies between loop iterations since Catapult does not understand how the shifting per channel works generating excessive logic and not being able to fully unroll the loop.

```
#include <ac_int.h>
class arrayExample_c{
	typedef ac_int<6,false> data_t;
	data_t array[10][10];
	public:
	arrayExample_c(){
		ac::init_array<AC_VAL_0>(&array[0][0], 100);
	}
	#pragma hls_design interface
	void run(data_t din, ac_int<4,false> ix, data_t &dout){
		#pragma hls_unroll
		for (int i=9;i>0;i--) array[ix][i] = array[ix][i-1];
		array[ix][0] = din;
		dout = array[ix][9];
	}
};
```

Figure 24. An example of a bad coding style for multidimensional array access.

Figure 25 presents the schedule in compact mode. It can be seen that the loop iterations are not executed in parallel but sequentially and there are a lot of multiplexers in each loop iteration. The clock frequency is set that low in this case, that the loop still fits within one C-step since there are data dependencies from later loop iterations to previous ones and Catapult was not able to schedule the design with higher clock frequency due to a too long feedback path.

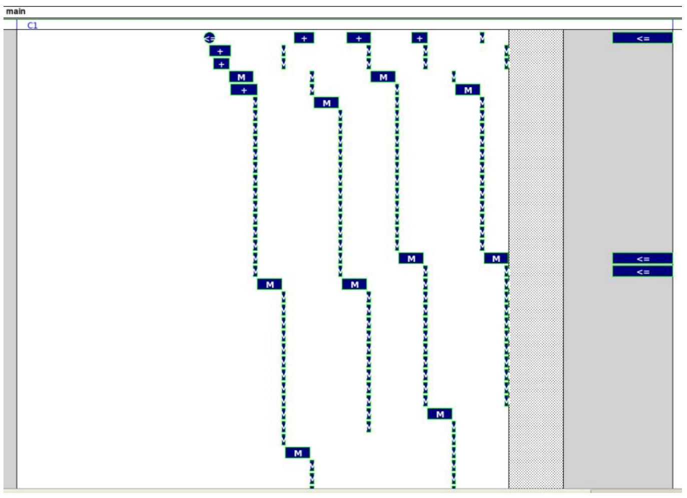


Figure 25. Schematic of a poorly coded multidimensional array access.

Taking a look at the schematic in figure 26, it is far more complex than desired knowing that a hundred registers and shifting logic are the goal.

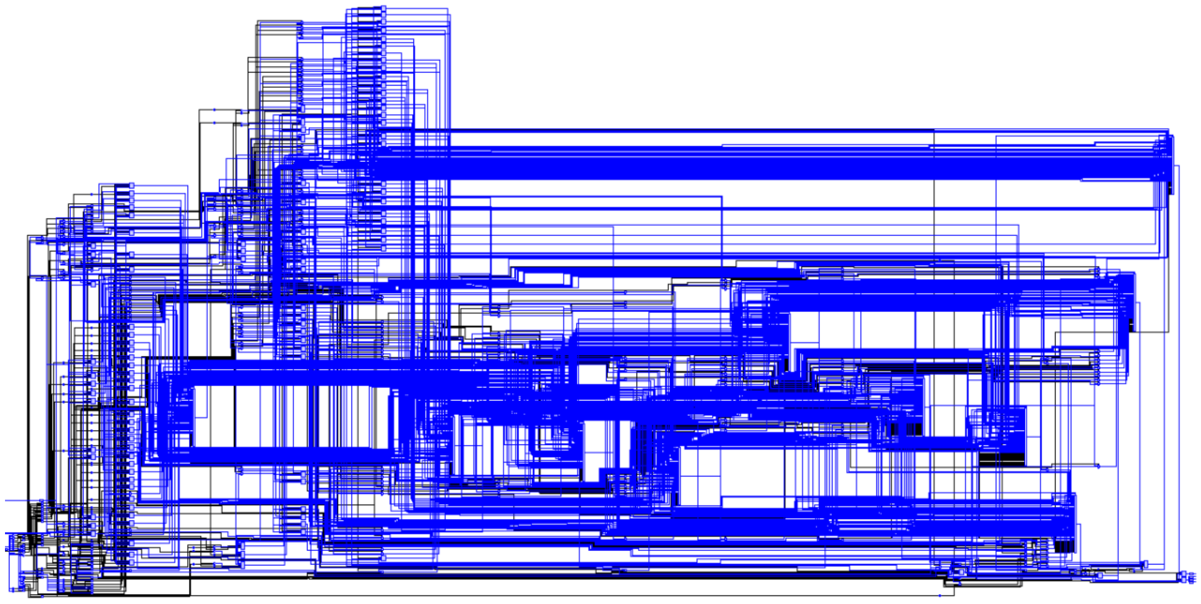


Figure 26. Schematic of a bad coding style example for multidimensional array access.

In figure 27 these issues are fixed by placing the array access loop within another unrolled loop and a condition. This way, the *ix* range is limited to 0-9 with the condition, and if larger values are inputted the behaviour can be defined which in this case is nothing since the code within the condition is never executed. Now all the array pointers are constants which allows Catapult constant propagation to optimize multiplexers away as well as the outer loop of the design.

```
#include <ac_int.h>
class arrayExample_c{
        typedef ac_int<6,false> data_t;
        data_t array[10][10];
        public:
        arrayExample_c(){
                ac::init_array<AC_VAL_0>(&array[0][0], 100);
        }
        #pragma hls_design interface
        void run(data_t din, ac_int<4,false> ix, data_t &dout){
                #pragma hls_unroll
                for (int j=0;j<10;j++){
                        if (ix == j){
                                #pragma hls_unroll
                                for (int i=9;i>0;i--) array[j][i]=array[j][i-1];
                                array[j][0] = din;
                                dout = array[j][9];
                        }
                }
        }
};
```

Figure 27. Better coding style of the same multidimensional array access example as in figure 24.

The corresponding figures for this design implicate much simpler and more efficient hardware. Figure 28 shows the schedule where there are no sequential operations, everything is executed in parallel despite the output write.

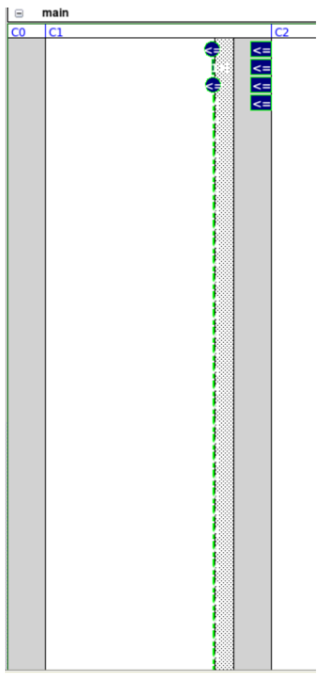


Figure 28. Schedule of a properly unrolled loop from a good coding style example.

In figure 29 the schematic of the fixed multidimensional array access is presented and compared to figure 26 it is much simpler and looking more like what is expected. The relative area numbers for good and bad coding styles for this example are 9448,97 and 3629,84 with the same technology library and clock frequency so with a simple example like this one it is possible to generate over 2,5 times larger design compared to the optimal one with poor coding style.

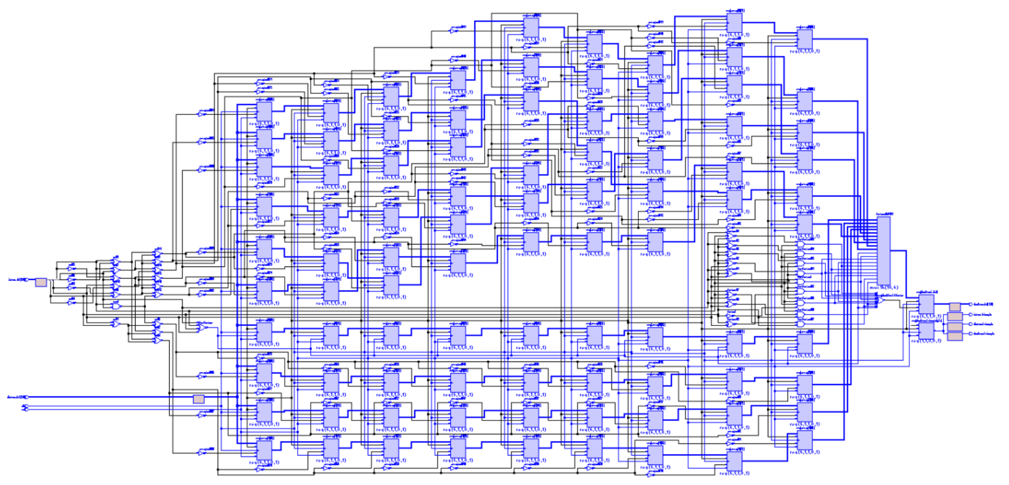


Figure 29. Schematic of a good coding style in multidimensional array access example.

5.5 In-context variable declaration

Variables should be declared in the context they are used, or as close to it as possible. This helps the HLS tool to analyse the dependencies and lifetime of the variable better.

The worst coding practice in these terms is to declare all signals within a block as member variables of a class. In HLS, these are called persistent variables since their data will be saved from one function call to another causing data registers in hardware. This might occur to variables declared within functions as well. The goal of in-context variables is to minimize the usage of registers and optimize as many variables to wires as possible.

Coding, where a variable is declared and used, followed by lines of code, and then using the variable again should be avoided if possible. The reason for this is the same as above: lines of code in between might take more than one clock cycle to finish leading to registering the original variable to save the value for later use. A better way would be to bring the usages of the variable closer together or to declare a new variable containing the same information later, if possible. The unintuitive thing about the in-context variable declaration is that using the smallest number of variables does not always result in the smallest hardware.

References to variables outside context should be minimized as well to avoid unexpected data dependencies. This means, for example, that persistent variable arrays should preferably not be accessed within unrolled loops if they cannot be indexed with the loop iterator. A better practise is to declare temporary variables to save the data in the loops, then create another loop where it is possible to index the persistent array with the loop iterator. Loop iterator is constant in unrolled loops and often optimized away generating only wires instead of multiplexers in the case of bad coding examples. Temporary variables should also be optimized away by the HLS tool, but this must be confirmed by the designer.

A bad coding style like this can also lead to unexpected data dependencies and bad scheduling. If indexing an array with something else than the loop iterator within an unrolled loop, the HLS tool might not be able to prove if the same element of an array is accessed in multiple loop iterations. In this case, the loop cannot be fully unrolled leading to longer latency and excessive multiplexing and area.

5.6 Summary of coding principles

As a summary for the HLS specific coding principles it can be said that the more parallelism there are in the target architecture, the more careful the designer must be to achieve optimal results. If the target architecture is a single pipe, where the operations are performed to one input signal in sequential order and there are no unrolled loops, the HLS tool is likely to extract good quality RTL. If there is a lot of parallel operations in the target architecture, it must be written to the source code and care must be taken so no undesired data dependencies between the desirably parallel operations are generated.

With the target architecture being known, the data flow and the amount of the resources should be known as well. This means that the resource sharing opportunities should be clear for the designer and the resource sharing should be written to the source code. Even though Catapult can prove mutual exclusivity in a lot of the cases in the source code, it does not understand the high-level data flow within the design. Therefore, the designer should leave as little as possible room for incorrect interpretation of the source code by the HLS tool.

The functionality should be perfectly described in the source code. There must be no room for undefined behaviour for example in form of uninitialized memory reads. Different compilers might interpret the undefined behaviour differently resulting in terrible quality of results.

6 CASE STUDY

This chapter of the thesis includes a case study of a digital downconverter and decimator IP block designed in HLS methodology using Catapult HLS tool and C++ as input language. The algorithm and architecture of the design are designed in a Nokia SoC organization and it is originally implemented in RTL level of abstraction. The aim is to implement the same design with HLS using Catapult and compare results in terms of area, latency, power, and designer productivity. Due to confidentiality policies, C++ source codes and the details about the design cannot be presented.

First, the design was partitioned into reasonably sized units after which they were described with C++ and finally translated to RTL with Catapult. C++ test benches were created as well to verify the correct functionality of the source code. Thorough RTL verification of the design is not a scope of this thesis and therefore not completed within the time frame. The C++ source code was verified properly and SCverify was used with some of the test cases to verify correct timing and the equivalent functionality of original C++ and the synthesized RTL with the given test cases.

As mentioned, the design consists of a digital downconverter (DDC) and decimator (DEC). The input for the design is a composite signal from which different carriers are separated and mixed down to the baseband. After this, the decimator drops the sample rate of the signal to a minimum level without compromising the data. The downmixing is done with a numerically controlled oscillator (NCO) followed by a chain of half-band filters (HBF) where each filter drops the sample rate of the signal to half. Before NCO, there are polyphase sub-band decimation (PSD) filters that separate the composite signal into sub-bands and drop the sample rate by four or two depending on the configuration. This allows the NCO to run at lower sample rates and reduces power consumption. The design can manage a maximum of 16 composite input and 64 output signals concurrently.

The design consists of three large design blocks placed sequentially one after another. Initially, each design block was programmed separately and tested with a separate test bench. After a basic functionality was verified with single carrier test cases, a top-level of the design was created to integrate design blocks as well as a top-level testbench for larger-scale verification. This enabled multi-carrier testing with multiple different configurations.

Carrier-specific configurations were given to the design from a software register bank implemented as direct input signals in the HLS design.

6.1 Requirements

Since it is desired to compare the HLS design to handwritten RTL programmed by other designers, some requirements need to be set for the HLS design to achieve.

The target architecture is the same as in the original RTL design. As mentioned about the capacity, 16 composite input signals arrive in four different input ports equalling a max of four composite signals per port. Polyphase sub-band decimation filters separate the composite signals to max 64 carriers in a total of four output ports. The number of filters in the design is pre-defined as well as the signal routing and timing for different filters for optimal usage.

The target clock frequency is 983.04 MHz. The design must be able to receive a data sample to all four input ports every clock cycle and produce an output sample to all four output ports every clock cycle as well. For HLS this means that the main function must be pipelined with II=1 and the design must not stall under any circumstances if there is data available. The input sample rates per antenna vary from 122.88 MSps (Mega samples per second) to 3932.16 MSps and the output sample rate per carrier from 7.68 to 491.52MSps. Input sample rates that are greater than the clock frequency mean that more than one input port is used to transfer the signal. Many different configurations need to be supported as well as runtime reconfiguration of the design.

The technology library used for the design is the same as used in the RTL design and it is provided by an external tool vendor.

Area- and latency-wise the goal was to match or go under the reference RTL written by hand after Design Compiler gate-level synthesis while meeting the timing requirement (throughput 1 and no negative slacks). Power was not an optimization target in this work, but the power data is also gathered for reference.

6.2 Catapult usage

During the design work, there was close support from Siemens EDA to help with Catapult. This was critical to find best practices and avoid mistakes in the design process. A lot of C++ coding style suggestions and advice came from Siemens EDA to achieve optimal QoR and to help understanding how the tool interprets C++, how to interpret the error messages, and how to fix the source code to be Catapult-friendly.

After a decent learning period Catapult usage became intuitive. Applying the basic directives to guide the RTL generation process and analysing their effect on the design was easy, but when run into difficulties the support was critical and it was there. The support also suggested special directives under development for some special optimization cases that would probably have left unnoticed otherwise even though found in the documentation.

The directives for the design were quite similar throughout different blocks. Common directives file was set up containing the default settings for the project. All the needed tool flows were enabled here, default settings applied as well as the technology library definition and clock and reset declarations. This file was then called at beginning of each design block's directives file.

The single most altered directive was clock overhead. This defines how big of a percentage of a clock cycle is left for the backend design. For example, if the clock cycle was 10ns and the clock overhead 20, Catapult would only use 8ns of the clock cycle for logical operations and leave the 2ns unused. Iterating this directive with Design Compiler, optimal timing solutions were found. If Design Compiler reported negative slack, clock overhead was increased and if the timing requirements were achieved easily, clock overhead could be reduced. Clock overhead can be defined locally for different blocks to optimize the design.

As described earlier, all the loops within the design were unrolled and the interface functions of sub-blocks were pipelined to achieve throughput 1. This caused some trouble with the scheduling of the design at some parts of the process, but all the problems were eventually solved with good coding practices described in chapter 5. Due to the throughput 1 requirement, back pressure could be removed from the design by changing the interface types. Since all the blocks had to be able to receive data every clock cycle, there was no need to use valid-ready type interface handshaking, but only valid could be used to tell the receiving block when to process the input data. Depending on if the receiving block had to function if there were no valid data available or if the block could wait for a valid sample, blocking or non-blocking read was used.

The directives were also used to give the same timing information to Catapult as were given to the Design Compiler so nothing unexpected would happen there. Before this information was aligned, Catapult would fill all clock cycles with logic, then Design Compiler would have additional information for example about external input delay on input ports causing negative slack times on these ports or usage of excessively large logic components to meet the timing requirements.

Once all the necessary information was in place to generate the desired hardware, some optimization was done with Catapult by analysing the schedule and adjusting directives to fill the possible gaps. More of this optimization could have been done to further reduce design latency and area but since the results were already quite satisfactory and the process of iterating with Catapult and Design Compiler is time-consuming, further optimization was left for future projects.

When creating and synthesizing the initial design, a lot of small entities were mapped into CCOREs, but in later optimization phases a lot of these small entities were inlined by removing the interface pragma from the class. This way the function logic of the class is handled as a part of the higher level in the hierarchy, where the function is called instead of synthesizing the class separately and using it as a library component. This may increase synthesis runtime as each function call is handled separately by Catapult, especially if there are multiple instances of the class. The upside of this is that the schedule and the resource usage of the block are not fixed to what is defined for the CCORE, but logic is handled as a part of a larger logical block. This potentially enables Catapult to push more operations within each clock cycle, which decreases latency and register usage. Greater resource-sharing opportunities may be introduced as well since the different instances of the same class use the same resources but not necessarily at the same time.

In the final optimization phases, using Catapult was critical since the last pieces of area optimization were done as well as all negative slack times needed to get rid of. At this point the process was iterative and the iterations took hours since changes needed to be done in Catapult and run DC synthesis after that to see and analyse the results to make any adjustments if needed. The changes were also very minimal since the negative slack times were small as well as the number of violating paths.

6.3 Results

The original goal of this work was to create a complete HLS implementation of the DDC DEC IP that was originally written by hand in VHDL language. This way perfect one-to-one comparison could be made between the two designs to analyse the QoR and design effort between the two methodologies. However, due to the schedule of this work, a few things are lacking from the HLS design which of course are excluded from the QoR comparison as well as possible.

Lacking features in the HLS design are some debugging features in the original RTL. There are also some bugs in the HLS design, like minor data mismatch occurring from data rounding or slightly mismatched data types. One bigger bug was left in the design that caused a major data mismatch in some test cases with very specific carrier combinations. This is probably a data alignment issue, where to samples get routed to the same place at the same time causing the loss of another sample and corrupting the following data for the same channel. These bugs are not estimated to remarkably affect the design area, but they should be kept in mind when comparing the results.

Dynamic reconfiguration was left undone as well. The features were programmed to the design as well as possible, but the verification effort for these features was too large in the given time window so it cannot be stated that dynamic reconfiguration works in the design. Since different carrier configurations have been verified and the dynamic reconfiguration simply changes the configuration in run-time, these features mostly require proper data alignment from the design. This functionality is not expected to increase the design area remarkably.

Design effort cannot be compared one-to-one in this case. In the original hand-written RTL design, there were two hardware designers with different experiences and a separate verification team working, all working on the same IP. While these people's working hours on the project could be count, in theory, the number would not be realistic since there is also other work than only the design included such as documentation, research, and decision making about architectural choices that were not included in the HLS design.

The HLS design on the other hand included adoption and learning of the methodology, the tool, and the source code language. The whole process of design and verification was done by one person with the support of another from Siemens EDA, who did not even have access to the source code files all the time during the process.

The area scores of the HLS and hand-written RTL design are presented in Table 3 as well as the relation between the two. If the percentage number is less than one hundred it means that the HLS design was smaller compared to hand-written RTL, and vice versa. To achieve one-to-one comparison, the differences between HLS and RTL design have been minimized as well as possible by for example excluding the missing debug features from the reference's numbers.

As can be seen from Table 3, the HLS design was in total approximately twenty per cent smaller than the hand-written RTL design, and there is only a single block that was bigger in HLS.

After the initial functional design was programmed but before the source code optimization, the area numbers were compared against the numbers in the RTL design. Depending on the nature of the block the area numbers varied from roughly the same as RTL to approximately ten times the corresponding RTL areas. It could be generalized that the blocks that were sequential by nature, for example, NCO had good results initially and those that exploited high parallelism had the worst results and there required the most optimization effort. After analysing the results and applying the source code optimization according to chapter 5, the results in table 3 were achieved.

The latency numbers in Table 3 do not include the group delays of the FIR-filters in the design. Latency reported here is the latency reported by Catapult, which means the number of clock cycles it takes from the first valid input to the first valid output sample. In the case of this design where multiple different channels are handled concurrently, achieving these latency numbers in a realistic test would require valid data coming in for the same channel every clock cycle in the input of each of the blocks reported. Since the nature of the design is to drop the sample rate of the channels, this will never happen, but the latency numbers illustrate the theoretical latencies of each block. The group delay of the filters is easily calculated from the number of taps in the filters.

To achieve the realistic latency numbers for comparison between the two designs, the same test case with theoretically longest latency was run and the latency was observed from the simulation waves. The latency number for RTL design was 2655 clock cycles and for HLS design, 2629.

Table 3. The area scores of synthesized blocks for both HLS and hand-written RTL design.

Block name	Area	Area_ref	Area %	Latency
alignmentbuffer	2233,1779	2596,6966	86	4
psd7_2	5878,0634	7554,4517	78	3
psd7_4	44861,8089	55303,8486	81	6
bandselector	41180,7533	37753,1335	109	5

hm_ddc	100183,5645	109732,396	91	15
NCO	1876,5403	2040,7824	92	5
HBF1	2714,4266	3953,7525	69	3
partial_deca	7812,0461	8385,457	93	9
hm_dec_a	64975,9506	67083,656	97	9
HBF2	4726,9051	7936,7392	60	2
HBF3	8768,1686	15116,3178	58	2
HBF4	17698,3904	22967,5434	77	5
six_to_two	879,282	1767,839	50	2
packetizer	6163,6608	8992,1376	69	1
cascaded_hbfs	69745,2223	98568,1478	71	17
hm_dec_b	139557,0936	197136,2956	71	17
Total	304716,6087	373952,3476	81	41

Gate and register counts of the two designs are presented in table 4. The differences between the two designs are not considered here, so these are absolute total counts for both designs, and they are not equivalent to each other. These numbers should not be looked at too closely as it's known that there is more hardware included in the RTL design compared to the HLS design. That is also implied by the total number of gates and registers.

One thing to note in this table is that for decimator A (hm_dec_a), there are more logic gates but approximately 30 % fewer registers in the HLS design. This would imply that Catapult can push more operations within each clock cycle, using the resources more effectively while saving total area and latency as can be seen from Table 3.

In other design blocks, the numbers seem to be in line with Table 3 as the HLS gate and register counts are smaller as expected. Keep in mind that the numbers in Table 4 are not one-to-one comparable.

Table 4. Logic gate and register count numbers for HLS and hand-written RTL designs.

	Gate count		Register count	
	HLS	RTL	HLS	RTL
hm_ddc	526333	614184	145660	177669
hm_dec_a	468189	457110	70905	106421
hm_dec_b	557025	918950	250233	291098
Total	1551547	1990244	466798	575188

The power consumption of the two designs is presented in table 5. The power numbers are acquired from the RTL level for both designs, not from more accurate gate-level models like the area numbers above. The same test case has been used to get these power numbers by generating a switching activity file with an RTL simulator using the same time window in the power simulations. The power analysis tool used for both designs was Ansys PowerArtist. Since the power simulations are performed in RTL and not gate level, the power numbers are only rough estimates as well, since RTL level power analysis inaccuracy might even be higher than 10 %. The differences between the two designs are again not considered since both designs must be fully functional for this kind of power analysis. Therefore, the missing features cannot be excluded from the RTL design, but these features' power consumption should be marginally small since they are not used in these test cases and therefore there is no switching activity in these parts of the design.

The UL test case is theoretical a maximum power test case, where data is fed into the design through all inputs with maximum input rate. The output rate is maximum as well since all four outputs feed out valid data every clock cycle.

In the DL test case, zero-data is fed into the design in the time window with some carriers enabled in the configuration by direct inputs. The zero data travels through the design as well since its purpose is to flush the pipeline of all valid data if there was some.

The purpose of the power consumption comparison is not to compare which methodology produces more power-efficient hardware. Since there are differences in the designs compared in table 5 and the RTL-level power analysis is so inaccurate, the purpose of this comparison is simply to prove that Catapult HLS generated RTL is not particularly bad in terms of power consumption, but the power consumption numbers are in the same order of magnitude as hand-written RTL. Again, there are many variables in this comparison and the results are not accurate, but the order of magnitude is the same. Note, that Catapult Ultra power optimization steps were run for the solution to achieve these results. This is not a part of the basic Catapult HLS package, but it requires a separate license.

Table 5. Power consumption in mW of HLS and RTL designs with the same test case.

	Downlink (DL)		Uplink (UL)	
	HLS	Ref	HLS	Ref
hm_ddc	76,8	89,0	332,2	324,5
hm_dec_a	101,1	138,8	220,3	237,7
hm_dec_b	131,9	151,2	234,9	392,1

Lines of code comparison have been done for the two designs. Active lines have been counted with a script that excludes blank and commented lines from the files in a given file list. C++ block comments are not excluded from the comparison, but their effect on the total line count is very minimal. The lines of code comparison itself is very rough that should not be looked at too closely but gives some estimation on the scale of coding effort. It should be noted as well that the directives files are not included for the comparison, so it is assumed that the synthesis to final RTL is done manually in Catapult GUI that takes time and effort as well.

The file list given to the script contains all synthesizable C++ source code files and the files that are required to compile the design for the HLS. This means that for example the testbench is not included in the count. For RTL design, the target for the file list is to include files that would share the same functionality with the HLS design. This means that the missing features from the HLS design are also excluded from the RTL designs file list. The generated VHDL lines of code have been included as well, for reference. This file list only contains the concatenated VHDL file from the Catapult. In this file, the entire synthesized design is in a single file.

The lines of code for the designs are presented in table 6. As can be seen, there are over 80 % fewer source code lines for HLS design compared to the hand-written RTL. Comparing the VHDL lines from the output of Catapult compared to hand-written VHDL, hand-written one has over 96 % fewer lines compared to the synthesized one. This, however, is not very relevant since the outputted files are not meant to be debugged or modified. Ideally, the RTL should not be touched since the functionality is supposed to be equivalent to the source code.

Table 6. Active source code line counts for HLS and hand-written RTL design.

Language	HLS	RTL
C++	2461	
VHDL	361305	13390

6.4 Analysis

Compared to hand-written RTL, HLS works better on resource allocation and scheduling the design. It was also noticed how big of an advantage design space exploration is in HLS. Even though the target architecture was known in the initial phase of the design work and it was written to the source code, DSE allowed to quickly explore different design options such as mapping block to top-down or bottom-up CCOREs or inlining the interface function, setting input or output registers to CCOREs and change clock overhead to push more or less logic within the clock cycles. All the solutions gave a different set of area, latency, and slack times for the blocks, the optimal ones were chosen for gate-level synthesis, and finally, if there were no problems and the results were satisfactory, there was no need for further iteration.

Especially big differences can be seen in the half-band filters. This can be explained by the fact that there are two different instances of FIR-filters in the RTL design, one for real and another for the imaginary part of the complex data. These two instances were created to the C++ source code as well, targeting the same architecture, but instead of leaving them as two separate instances in the final design, DSE allowed to inline the filter instances on the HLS design giving better results in terms of area and latency. Assumably this is because both instances had their own adder trees that could be combined and optimized more effectively as part of the higher-level hierarchy.

The overall difference in terms of the area can be explained by Catapult's ability to push the maximum number of operations within each clock cycle and therefore optimise the register usage. This was observed in practise while optimizing the design to get rid of the negative slack times; if any negative slack times were left on some of the design paths, there were immediately thousands of violating paths of similar negative slacks. By slightly increasing clock overhead or some other method, the timing violations were fixed but that just tells how effective the scheduling in Catapult is. The latency numbers acquired from the simulation waves also refer to the fact that Catapult optimized the pipeline better.

It is also easy to analyse the schedule of the design in Catapult and adjust directives in case there are for example loosely filled clock cycles. In traditional RTL-flow, these are not possible, and the final fine-tuning of the solution is a lot more difficult and time-consuming, and often left undone.

The gate and register counts of the two designs seem to be roughly in line with the area numbers as well as the ratios between the two. The only noticeable thing is that there are more logic gates in the HLS design for hm_dec_a and remarkably fewer registers compared to hand-written RTL considering the overall three per cent area difference between the two. This would suggest that Catapult can push more operations within each clock cycle optimizing register usage and decreasing latency of the design. Otherwise, gate and register counts seem as expected.

The power consumption comparison was done to demonstrate similar order of magnitude in the acquired power numbers. From this point of view, the comparison was successful but any other conclusions can be made due to the inaccuracy in the numbers.

The lines of code comparison is done to roughly evaluate the design effort of the designer. Designer's personal preferences in the writing process, for example, have an impact on these numbers so they only roughly give the scale of the source code amounts. While being a rather silly way to measure it, lines of code in a specific language give some idea about the design complexity and workload related to the projects. Of course, in the case of HLS writing the source code is not the entire work, but the HLS tool must be used as well to achieve the RTL that is written directly in the other flow.

The VHDL lines for the Catapult HLS design are added simply for reference and it is not a relevant number by any means. The VHDL files contain some information that the designer might need to check but these files are not supposed to be modified and there should not be a need to debug these as the functionality is equivalent to the original C++ source code files.

7 DISCUSSION

The idea of HLS first seems like an easy methodology to create high-quality hardware, but the truth is that to get acceptable results from HLS generated RTL, the designer must know about SW, HW design, and the tool itself. The designer must be able to write flexible high-quality C++ or other high-level code and have an idea of the target architecture to achieve the optimal QoR. If, and when, the tool does not understand the programmer's intent and extracts unexpected hardware the designer must be able to figure out what the tool is doing and fix the source code or design constraints.

This thesis aimed to find the best C++ coding practices to generate high-quality RTL with an HLS tool. Theoretically, the same RTL architecture should be achievable with HLS and hand-written RTL. According to the results of the case study, good practices were found to generate the desired hardware.

The goal of the case study in chapter 6 was to generate RTL with equivalent QoR as in hand-written RTL, with a five per cent margin acceptable in the area. As it turned out, some extra optimization opportunities were found with HLS thanks to DSE compared to the hand-written design resulting in a 19 % smaller total area with the same throughput and power consumption. These results are better than in previous case studies [3],[4],[5],[6], and the goals of the thesis work were achieved, but it cannot be generalized that HLS generates better hardware compared to hand-writing the RTL. Equivalent hardware can be created with both methodologies. It should be noted that in HLS, DSE allows easy exploration to find the optimal solutions for each sub-block potentially resulting in better QoR.

The differences and difficulties in writing HLS friendly C++ that produces high-quality hardware compared to traditional software code are related to the parallel nature of the target architecture in HLS. Traditional software code targets a different kind of technology that executes the operations in the source code sequentially. In HLS this changes since a lot of the operations in the source code are desired to be executed in parallel. Instead of writing sequential code, the designer must understand the parallelism in the written source code. The parallel operations and the order of execution might not be dependent on the order of the operations in the source code but the data dependencies between the operations instead.

To generate high-quality hardware, the designer must understand how the tool interprets the source code, which might even be slightly tool-dependent. Learning how Catapult analyses the variable lifetimes and data dependencies between operations helps also to understand if the source code is good or not, and how to make it better even before the synthesis. The target architecture should be known before even writing the code so if the outcome is not what is desired that should be visible to the designer.

It was found that the claim in [2] about HLS tools behaving like black boxes is not entirely true. It certainly is easy to generate sub-optimal hardware, but while there were some difficulties analysing the unexpectedly generated hardware in some cases, the behaviour of the synthesis tool certainly got clearer with more experience with it. This might have changed in the past few years but the analysability of the generated RTL is rather simple in different analysis views in the current Catapult version and they even include the Design Analyser tool to help the analysis and link the source code to the resulting RTL.

Learning the HLS friendly C++ coding and understanding the synthesis tool comes with experience in HLS. In the beginning trial, error and analysis are common and good ways to learn but since some things in HLS may seem unintuitive, not all cases can be reverse engineered, and in those cases, support is needed to improve.

For an HW designer to adopt HLS friendly coding principles, they would require training in the input language and the HLS tool. The training should include some hands-on example exercises with the tool to demonstrate the effects of the changes in the source code. For an SW engineer, training on the target technology would be required as well as the synthesis tool to demonstrate the parallelism in the source code and the difference to traditional SW coding. In both cases, an open mind should be kept since good HLS coding principles might not comply with the previously learned ones.

Memory components were not used in the case study so the coding principles with those were not experimented with too carefully. HLS might introduce some challenges or optimization opportunities with them, so that could be studied more in the future.

For the tool developers, maybe the tool could trigger a warning if assumingly unexpected RTL is about to be generated, for example, if a loop that is supposed to be unrolled, cannot be fully unrolled. This would simplify the analysis process of the generated hardware since now it is the designer's responsibility to notice these things the analysis views after the synthesis. This kind of unexpected structure often is larger and has longer latency than the desired one.

In conclusion, Catapult HLS with C++ is a viable option for traditional RTL flow in data-driven applications such as this one. The case study shows that HLS can potentially produce higher quality RTL compared to handwriting it with less effort. The design effort was only compared in terms of lines of source code in this thesis but other studies [4],[5][6],[9] suggest fewer working hours for HLS as well.

8 SUMMARY

The purpose of this thesis was to find good coding practices for C++ targeting high-level synthesis. The thesis includes a case study where a digital downconverter and decimator is implemented with high-level synthesis using the found coding practices and the quality of results is compared against the same design hand-written in VHDL by other designers.

Chapters 2, 3, and 4 of this thesis contain an introduction to HLS as well as the used input language C++ with the HLS additions and restrictions such as the algorithmic C data types. Siemens EDA Catapult HLS tool is introduced with the functionality, features, and design steps within the tool.

Chapter 5 generalizes some of the coding principles to achieve optimal QoR in the resulting RTL. Different examples are presented with good and bad coding in some specific cases as well as the effect of different coding styles on the QoR.

Chapter 6 presents the case study of designing an uplink digital downconverter and decimator with HLS. First, the specifications and the requirements of the design are discussed, followed by personal user experience with Catapult and HLS in general. Finally, the results of the case study are presented in terms of area, latency power, and lines of active source code, and these results are compared against a close to equivalent hand-written RTL design written by different designers. The results show that the HLS design is almost 20 % smaller with slightly lower latency. The power numbers are not accurate, but the order of magnitude is found to be similar in both designs.

As an overall conclusion, it can be stated that Catapult HLS generates high-quality RTL if the source code is written properly according to good coding principles. It is easy to generate poor-quality RTL without paying attention or not analysing the result with the provided tools within Catapult. Minor differences to the source code have a huge impact on the resulting hardware. According to this thesis, Catapult HLS is a viable option for traditional RTL flow DFE ASIC development.

9 REFERENCES

[1] G. Martin and G. Smith (2009) High-Level Synthesis: Past, Present, and Future. In: IEEE Design & KPN Test of Computers, August 21, vol. 26, no. 4, pp. 18-25

[2] Zelei Sun et al. (2016) Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis. In: 21st Asia and South Pacific Design Automation Conference (ASP-DAC), March 10, Macao, China, pp. 218-225.

[3] Joentakanen T. (2017) Evaluation of HLS modules for ASIC backend, Master of science thesis, Master's Degree Programme in Electrical Engineering, Tampere University of Technology, Tampere, 71 p.

[4] Torppa E. (2015) High-level synthesis in IP based SoC development. University of Oulu, Department of Electrical Engineering, Degree Programme in Electrical Engineering. Master’s thesis, 69 p. I. Kivimäki

[5] Ollikainen P. (2016) SoC subsystem design using SystemC based high-level synthesis. University of Oulu, Degree Programme in Electrical Engineering. Master’s Thesis, 48 p.

[6] Kivimäki I. (2016) High-Level Synthesis Design Flow in FPGA Design. Degree Programme in Electrical Engineering, University of Oulu, Oulu, Finland. Master’s thesis, 60 p.

[7] P. Coussy, D. D. Gajski, M. Meredith & A. Takach. (2009) An Introduction to High-Level Synthesis. In: IEEE Design & Test of Computers, August 21, vol. 26, no. 4, pp. 8-17.

[8] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer (2018) A modular digital VLSI flow for high-productivity SoC design. In: Proceedings of the ACM/IEEE Design Automation Conference, June 24-28, San Francisco, CA, USA, pp. 1-6.

[9] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, (2019) Are We There Yet? A Study on the State of High-Level Synthesis. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, May 8, vol. 38, no. 5, pp. 898-911.

[10] Mentor Catapult HLS tool Datasheet (accessed 14.1.2020). URL: https://s3.amazonaws.com/s3.mentor.com/public_documents/datasheet/hls-lp/catapult-high-level-synthesis.pdf

[11] High-level Synthesis Bluebook v10.6 (2020), Mentor Graphics Corporation, Oregon, 251 p.

[12] B. C. Schafer and Z. Wang, (2020) High-Level Synthesis Design Space Exploration: Past, Present, and Future. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, September 24, vol. 39, no. 10, pp. 2628-2639

[13] G. Schewior, C. Zahl, H. Blume, S. Wonneberger and J. Effertz, (2014) HLS-based FPGA implementation of a predictive block-based motion estimation algorithm — A field report. In: Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing, October 8-10, Madrid, Spain, pp. 1-8.

[14] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski (2012) Impact of FPGA architecture on resource sharing in high-level synthesis. In: Proceedings of the ACM/SIGDA international symposium on

Field Programmable Gate Arrays (FPGA '12), February 22, New York, NY, USA, 111–114.

[15] S. A. Edwards. (2006) The Challenges of Synthesizing Hardware from C-Like Languages. In: IEEE Design & Test of Computers, September 25, vol. 23, no. 5, pp. 375-386.

[16] J. Santiago da Silva, F. Boyer & J. M. P. Langlois. (2019) Module-Per-Object: A Human-Driven Methodology for C++-Based High-Level Synthesis Design. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), April 28-May 1, San Diego, CA, USA, pp. 218-226.

[17] C++ reference, Information, A brief description page (accessed 29.12.2020). URL: http://www.cplusplus.com/info/description

[18] Stroustrup B. (1985) The C++ Programming Language. Addison-Wesley, Reading, MA, 326p.

[19] C++ reference, Information, History of C++ page (accessed 28.12.2020). URL: http://www.cplusplus.com/info/history

[20] Cook S. (1989) Introducing object-oriented systems. In: IEE Colloquium on Applications of Object-Oriented Programming, November 16, London, UK, pp. 1-1

[21] J. R. Garcia Ordaz & D. Koch. (2017) On the HLS Design of Bit-Level Operations and Custom Data Types. In: FSP 2017; Fourth International Workshop on FPGAs for Software Programmers, September 7, Ghent, Belgium, pp. 1-8.

[22] Catapult Synthesis User and Reference Manual (2020), Mentor Graphics Corporation, Oregon, 1236p.

[23] HLS LIBS Homepage (accessed 29.12.2020). URL: https://hlslibs.org/

[24] Kahn, G. (1974). The semantics for simple language for parallel programming. In: Information Processing 74, Proceedings of IFIP Congress 74, August 5-10, Stockholm, Sweden.

[25] IEEE Standard for Standard SystemC Language Reference Manual. (2012). In: IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), January 9, pp. 1-638.

[26] F. Sijstermans, J. Li (accessed 14.1.2020) Working smarter, Not harder: NVIDIA Closes Design Complexity Gap with High-Level Synthesis, Mentor paper, URL: https://resources.sw.siemens.com/en-US/white-paper-working-smarter-not-harder-nvidia-closes-design-complexity-gap-with-hls