TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TERO JOENTAKANEN
EVALUATION OF HLS MODULES FOR ASIC BACKEND

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 17th August 2016

ABSTRACT

TERO JOENTAKANEN: Evaluation of HLS modules for ASIC backend
Tampere University of Technology
Master of Science thesis, 71 pages
May 2017
Master's Degree Programme in Electrical Engineering
Major: Embedded systems
Examiner: Prof. Timo D. Hämäläinen
Keywords: High-level synthesis, System-on-Chip, ASIC backend design

Digital systems continue growing in complexity, but the design and verification productivity has not been able to improve in the same manner, which has led to a productivity gap. Raising the abstraction level of the design with high-level synthesis (HLS) has been proposed to increase productivity. However, at the higher abstraction level, the designer has less control on the generated register-transfer level (RTL) code, which might cause problems later in the design flow. Moreover, certain design steps might be impractical to carry out with HLS.

This thesis work investigates if HLS is compliant with an existing ASIC implementation flow. The research is conducted by creating an IP (intellectual property) block with a modern HLS tool and passing the generated RTL code through the various steps in the flow. The quality of results and design effort are also compared to the manually coded RTL implementation of the same IP.

The HLS tool and the generated RTL code are found mostly compliant with the existing flow, but a few problems are identified in the ECOs (engineering change orders) and technology-specific component instantiation. The HLS design has almost equal physical area with the hand-written RTL design, and it meets the given timing constraints. Design effort with HLS is estimated 20–50% smaller compared to traditional RTL design, and the C++ code contains 60% fewer lines of code than the manually written VHDL code.

TIIVISTELMÄ

TERO JOENTAKANEN: HLS-lohkojen evaluointi ASIC-piirien toteutusvuossa
Tampereen teknillinen yliopisto
Diplomityö, 71 sivua
Toukokuu 2017
Sähkötekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Prof. Timo D. Hämäläinen
Avainsanat: Korkean tason synteesi, System-on-Chip, ASIC-piirien toteutus

Digitaalijärjestelmät kasvavat yhä monimutkaisemmiksi. Suunnittelun ja varmennuksen tuottavuus ei ole kuitenkaan pysynyt tämän kehityksen perässä, mikä on ajan myötä johtanut tuottavuusvajeeseen. Eräs ratkaisu tuottavuuden parantamiseksi on nostaa suunnittelun abstraktiotasoa käyttämällä korkean tason synteesiä (high-level synthesis, HLS). Korkeampi abstraktio rajoittaa kuitenkin suunnittelijan mahdollisuuksia vaikuttaa tuotettuun rekisterisiirtotason (register-transfer level, RTL) kuvaukseen ja saattaa myös vaikeuttaa suunnittelun muita vaiheita.

Tässä diplomityössä tutkitaan HLS:n soveltuvuutta ASIC-piirien toteutukseen. Tutkimusta varten luodaan IP-lohko (Intellectual Property) käyttäen HLS-työkalua, jonka tuottama RTL-koodi viedään suunnitteluvuon eri vaiheiden läpi. Myös työmäärää ja tulosten laatua verrataan käsinkirjoitettuun RTL-kuvaukseen, joka samasta IP:stä on saatavilla.

Tässä työssä käytetty HLS-työkalu ja sen tuottama RTL-koodi osoittautuvat soveltuvan olemassa olevaan suunnitteluvuohon, mutta myös muutamia ongelmia nousee esille ECO-muutoksissa (Engineering Change Orders) ja teknologiakomponenttien käytössä. HLS-lohkon pinta-ala on lähes sama kuin käsinkirjoitetulla, ja sen ajoitus pysyy vaadituissa rajoissa. Työmäärä HLS:llä on arviolta 20–50% pienempi verrattuna perinteiseen RTL-suunnitteluun, ja C++-koodi sisältää 60% vähemmän koodirivejä kuin käsinkirjoitettu VHDL-kuvaus.

PREFACE

This thesis work was done in the SoC organization of Nokia during fall 2016 and early 2017.

I would like to thank all of my colleagues for the great working environment, and also for the support and guidance they have provided for this thesis work. Especially I want to thank my supervisors Dr. Erno Salminen and Prof. Timo D. Hämäläinen for their excellent and valuable feedback. I also want to thank my line manager Jyrki Hyrsylä for the opportunity to do this thesis at Nokia, and Dr. Ari Kulmala for providing an interesting topic to the thesis. Many thanks also to Rich Toone and Richard Langridge at Mentor Graphics for their excellent support in tool-related matters.

Finally, I want to thank my family and friends for supporting me during this thesis work and the many years of studies.

Tampere, 4.5.2017

Tero Joentakanen

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

AC	Algorithmic C
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
BIST	Built-In Self Test
CDC	Clock Domain Crossing
CDFG	Control and Data Flow Graph
CIC	Cascaded Integrator-Comb filter
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
DFG	Data Flow Graph
DFT	Design For Testability
DMA	Direct Memory Access
DSP	Digital Signal Processing
ECO	Engineering Change Order
FIFO	First-In First-Out
FIR	Finite Impulse Response filter
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
IP	Intellectual Property
IQ	In-phase and Quadrature
LEC	Logical Equivalence Checking
LTE	Long-Term Evolution
LUT	Look-Up Table
NRE	Non-Recurring Engineering
PCB	Printed Circuit Board
RAM	Random Access Memory
RTL	Register-Transfer Level
SAIF	Signal Activity Interchange Format
SC	SystemC

SoC	System-on-Chip
TCL	Tool Command Language
TLM	Transaction-Level Modeling
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language

1. INTRODUCTION

Digital systems have grown extremely complex over the years, and the amount of functionality integrated in these systems will only keep increasing in the future. Designing and manufacturing an integrated circuit (IC) (Figure 1.1) may require years worth of design effort from hundreds of engineers. At the same time, the demand for shorter time-to-market and reduced costs limit the possibility to increase the product development time or the size of the engineering teams. Hence, the productivity per engineer has to improve to enable the development of the increasingly complex systems.

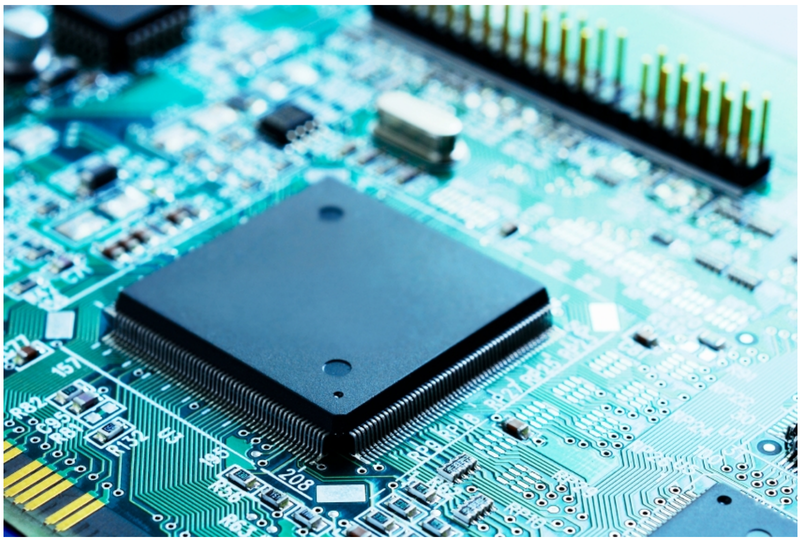


Figure 1.1 *Integrated circuit on a printed circuit board [51].*

Over the past two decades, the design work has been mainly carried out at register-transfer level (RTL) using hardware description languages (HDL), such as VHDL or Verilog. At this abstraction level, the designer describes the digital logic on a cycle-accurate basis, implying every register in their code. However, as the digital systems continue growing in complexity, designing them at RTL will eventually become impractical. Therefore, designers are nowadays looking to raise the abstraction level such that instead of hand-coding the architecture at RTL, they would describe the algorithmic behavior, which would then be synthesized to RTL using a high-level synthesis (HLS) tool.

The demand for a higher abstraction level has led to the development of a large number of HLS tools, both academic and commercial. Many research papers have addressed this abundance by providing evaluation methods for choosing a suitable tool [27, 23]. Previous studies have also compared HLS to the traditional RTL design, and the current generation of tools have shown promising results in terms of design productivity and quality of results [53, 54, 17]. However, as these papers focus on the frontend design (i.e. generating the RTL description out of the given algorithm), only few of them consider the subsequent design steps and checks that the RTL code goes through before it is implemented as a physical circuit on silicon. Nevertheless, it is possible that the tool-generated RTL code contains structures that are not feasible to implement or otherwise cause problems in the later stages of the design flow.

This thesis work evaluates the suitability of HLS-generated RTL code to the backend design flow of ASICs (Application-Specific Integrated Circuit) and develops means to integrate HLS methodology into an existing RTL design flow. The purpose of this thesis is also to provide HLS tool vendors with feedback on how to improve the tools in the future to make them more suitable for ASIC design. This work is continuation to previous theses [45, 18, 30, 15] that have evaluated the frontend design flow with several HLS tools.

This study is carried out by creating a HLS design and passing the generated RTL code through the various steps of the backend implementation flow. Although the main focus of this thesis is on the backend flow, the HLS design is also briefly compared to the existing hand-coded RTL design in terms of the quality of results and design effort, and the problems are assessed that are still present in the current tools.

The structure of this thesis is following. Chapter 2 covers the design process of ASICs and introduces the abstraction levels of the design flow. The fundamentals of HLS are described in Chapter 3, as well as the advantages and problem areas of the current generation of the HLS tools. Chapter 4 presents the design that is used as a test case and the HLS tool that is evaluated in this thesis work. The research questions are covered in Chapter 5, which is followed by the results in Chapter 6. Chapter 7 provides recommendations for both the designers looking to use HLS and the tool vendors to further develop the HLS tools. Finally, Chapter 8 concludes the results of this thesis work.

2. ASIC DESIGN

Digital systems are extremely complex nowadays. A single system might contain several processors, memories, and peripherals, which together may comprise billions of transistors [37]. Therefore, the only practical way to realize these systems is to use integrated circuits (IC) where a large amount of transistors are integrated on a single silicon die. In the past, the systems were built of several chips so that each component, such as processor or memory, had its own respective IC. However, as the level of integration has increased, a single chip has been able to contain more functionality, and nowadays whole systems can be included on a single chip. This kind of system is commonly known as a System-on-Chip (SoC) [38]. An example of a modern SoC is shown in Figure 2.1 that represents a layout of Qualcomm's Snapdragon 810 mobile chip used in smartphones and tablets. As can be seen, the chip contains several functional blocks, such as processing units, LTE (Long-Term Evolution) modem, and positioning systems.

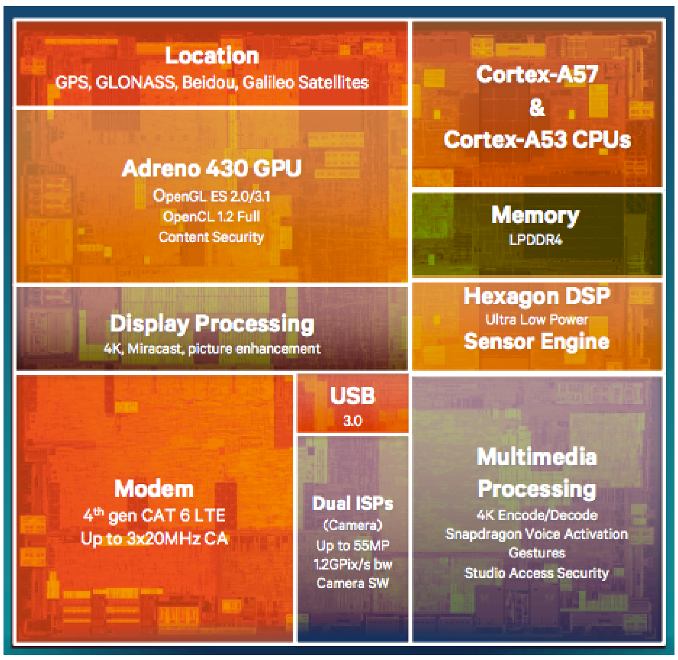


Figure 2.1 *Layout of Qualcomm's Snapdragon 810 SoC [10].*

For many applications, general-purpose processors and microcontrollers are suitable for implementing the desired functionality. However, these general-purpose components are typically designed for a wide range of applications, and thus they often contain excessive functionality. This overhead means wasted area and increased power consumption which might be impractical for certain applications. Furthermore, the performance of these components might not be sufficient for applications that require heavy computation and low latency [11]. When more specialized components are needed, there might be readily available ICs that are designed for a specific functionality. These application-specific standard products (ASSP) are typically available for applications that are needed often and by many manufacturers [38]. A typical example of an ASSP would be a bridge component for connecting a bus using USB (Universal Serial Bus) protocol to a UART (Universal Asynchronous Receiver/Transmitter) bus.

If there is no suitable component readily available, the manufacturer might have to design a customized logic circuit. In some cases, it might also be cheaper to produce own circuit instead of buying them from another manufacturer when a large amount of components is needed. In general, there are two options for implementing customized digital logic: one option is to use a programmable logic device, such as a FPGA (Field-Programmable Gate Array) or a CPLD (Complex Programmable Logic Device); the other option is to design an ASIC that is a customized IC where the logic is fixed and cannot be altered after fabrication.

There are naturally trade-offs for both options. Since the programmable devices are designed to be used in many applications, they will contain extra logic which degrades their performance and power efficiency [19]. On the other hand, designing them contains fewer risks since they allow fixing bugs afterwards by reconfiguring the logic. In contrast, a defective ASIC would have to be completely remanufactured in the worst case.

The development time and cost are also significant factors in the choice between an ASIC and a programmable device. The development time for FPGAs can be in the order of months whereas an ASIC design cycle may take more than a year to complete [19]. Moreover, the NRE (non-recurring engineering) costs of an ASIC are tremendously higher: creating the masks and fabricating the first sample might cost millions of dollars [19]. For FPGA designs, the NRE costs can be an order of magnitude lower. However, the unit price of an ASIC is generally less compared to a programmable device. Therefore, the more chips are needed, the more cost-effective option the ASIC will be.

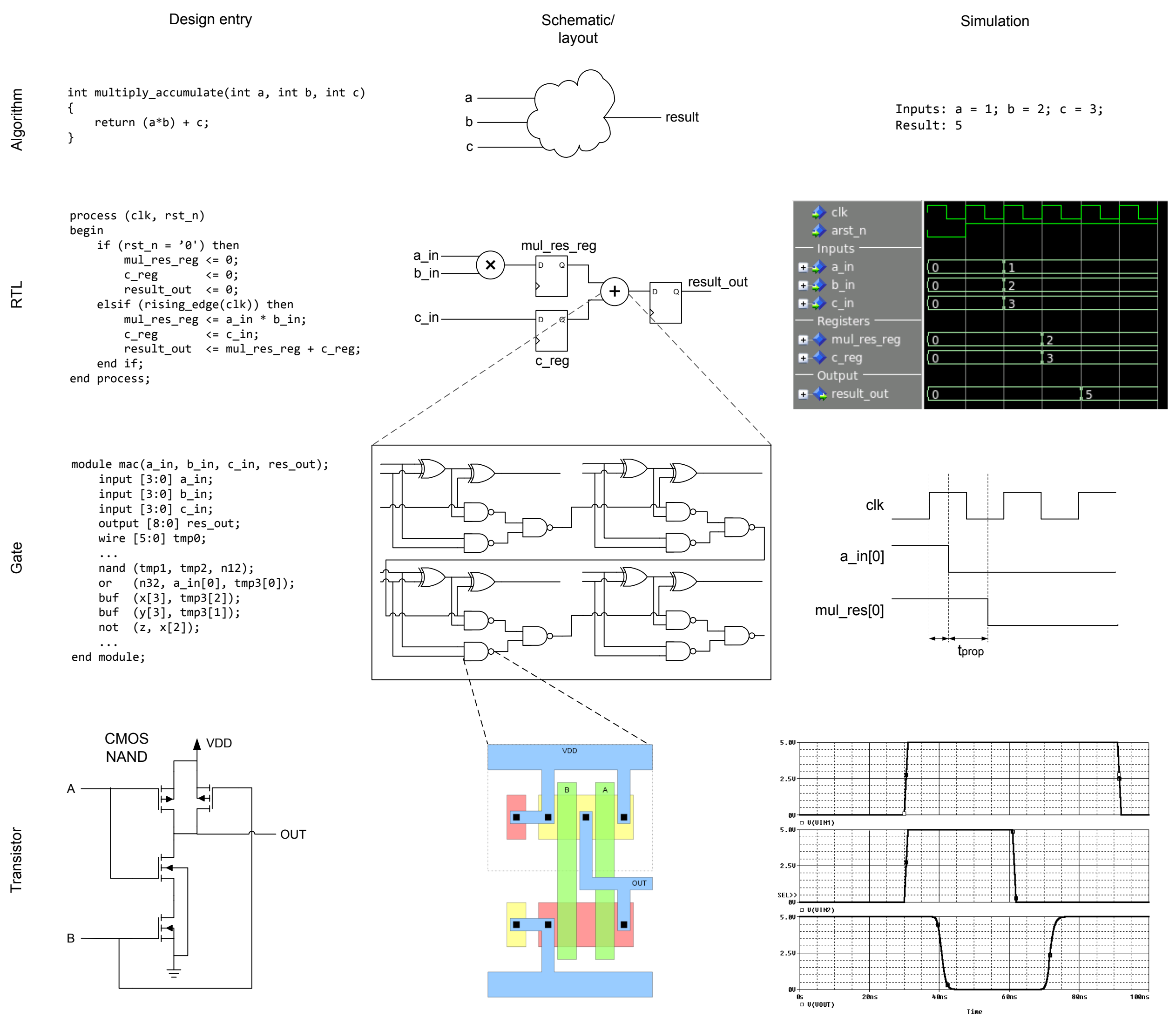
2.1 Abstraction levels

Digital logic is modeled at several abstraction levels to manage the design complexity [6]. The most often considered levels are algorithm level, register-transfer level, gate level, and transistor level. This is not an exhaustive list, however, as one might also include intermediate levels between these. For example, transaction-level modeling (TLM) can be used between algorithm level and RTL, so that the system is modeled at the level of transactions, such as sending a data packet to another module [9]. Moreover, all abstraction levels can be divided into more fine-grained levels. For instance, algorithm-level model may be timed or untimed, or it might use bit-accurate data types instead of floating point numbers. Nevertheless, these four abstraction levels are the main ones that are considered in most designs.

Figure 2.2 illustrates the abstraction levels in three different domains. Each row represents an abstraction level such that the uppermost row shows the highest abstraction level – which is the algorithm level – and below are the RTL, gate level, and transistor level in a descending order. The figures on the left show a typical input format at that particular abstraction. For example, in the case of RTL, the input is a VHDL process that implements the RTL architecture. The figures in the middle show a schematic or layout that represents the logic, and the rightmost figures show a typical output format that a simulator might provide at that abstraction level. At algorithm and RT levels, the figure shows an implementation of a multiply-accumulate circuit that calculates the result of $ab + c$. For the other two abstraction levels, only parts of the circuit are shown due to increasing complexity of the presentation.

Algorithm level is used to describe the behavior of the logic. In other words, it describes the function that relates its inputs to outputs, and it has no concept of timing. Algorithm level is often used in behavioral modeling, where the algorithm designer explores different options for the algorithm and investigates which of them fulfills the given specifications. Once a suitable algorithm is found, the model is often refined to a more accurate representation of the hardware [9]. For example, numbers can be presented with formats that are more suitable for hardware implementation, which also allows using the model to define the minimum data width that does not cause too much quantization noise in the results. A bit-accurate model has also the advantage that it can generate reference data for the functional verification that is carried out at lower abstraction levels.

A RTL description introduces timing to the model at the level of clock cycles [6]. Events happen only at the clock edges, and all the operations related to that event



```
int multiply_accumulate(int a, int b, int c)
{
    return (a*b) + c;
}
```

```
Inputs: a = 1; b = 2; c = 3;
Result: 5
```

```
process (clk, rst_n)
begin
    if (rst_n = '0') then
        mul_res_reg <= 0;
        c_reg       <= 0;
        result_out  <= 0;
    elsif (rising_edge(clk)) then
        mul_res_reg <= a_in * b_in;
        c_reg       <= c_in;
        result_out  <= mul_res_reg + c_reg;
    end if;
end process;
```

```
module mac(a_in, b_in, c_in, res_out);
    input [3:0] a_in;
    input [3:0] b_in;
    input [3:0] c_in;
    output [8:0] res_out;
    wire [5:0] tmp0;
    ...
    nand (tmp1, tmp2, n12);
    or   (n32, a_in[0], tmp3[0]);
    buf  (x[3], tmp3[2]);
    buf  (y[3], tmp3[1]);
    not  (z, x[2]);
    ...
end module;
```

Figure 2.2 *Abstraction levels in digital design. From top to bottom: algorithm level, RTL, gate level, and transistor level [49].*

are executed instantaneously. RTL describes the data flow between registers, and the operations between registers are defined only at high level. For example, the RTL code might describe that the outputs of two registers are summed and stored to another register, but it does not care about the implementation details of the adder. RTL is nowadays the main abstraction level for designing digital logic.

At gate level, the logic is implemented with gates that are primitive logic elements such as NOT, AND, and OR [6]. Gate-level description is already close to the actual physical implementation. If the physical characteristics of the gates are known, it is possible to get rather accurate estimates of the design area and power, and it also allows analyzing the timing of the circuit. In a typical design flow, gate-level netlist is automatically generated from the hand-written RTL description using a

RTL synthesis tool.

The transistor level is considered mainly in the final layout design before the masks used in the silicon fabrication are created. Since the gate-level abstraction is already a rather accurate representation of the physical circuit, most of the physical design flow that contains, for example, the placing and routing of the components is done at the gate level. The transistor level is, nonetheless, important for the engineers developing new technologies and creating standard cell libraries. However, at transistor level, the design is no longer digital, but involves also the analog characteristics of the transistors [6].

2.2 Design flow

The design flow for digital systems can be roughly divided into three phases: specification, design, and verification. Each of these phases are carried out at several abstraction levels. For example, the specification phase consists of tasks such as specifying the algorithm, RTL architecture, and the target technology of the physical circuit. Hence, these three phases typically proceed in parallel in the design flow. This has been demonstrated in Figure 2.3 that shows how the design phases might progress in a typical ASIC project. It is also common that the design flow contains iteration loops between the phases. For instance, if a bug is found in the functional verification of the RTL, or the synthesized gate-level implementation does not meet the timing constraints, the RTL designer will fix the issue by modifying the RTL code according to the given feedback.

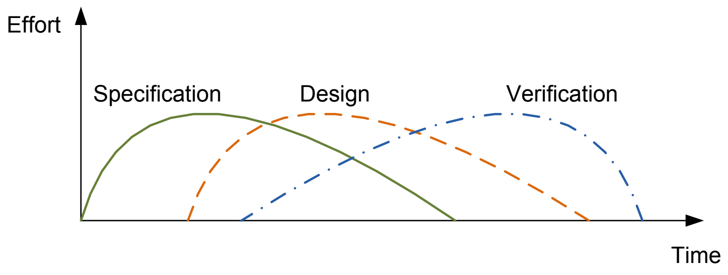


***Figure* 2.3** *Design phases and their effort in the ASIC project timeline.*

The design flow starts with the high-level specification of the functionality of the system. This specification is then partitioned and refined into a more detailed description, and eventually to an algorithm or behavioral model. In SoCs, this phase also includes decisions such as how the functionality is split between software and hardware [50]. In ASIC design, the target technology should also be specified at an early stage to allow evaluating the feasibility of the physical implementation of the system.

The design phase consists of the implementation of the system, which can be generally divided into frontend and backend design [26]. The frontend design consists of implementing the RTL architecure based on the behavioral model provided by the algorithm designer. The backend design comprises the physical implementation steps, such as mapping the RTL structures to standard cells of the target technology (i.e. RTL synthesis), placing and routing the cells on the physical layout, and checking that the physical netlist meets all of the requirements related to the physical parameters, such as area, timing, and power.

Functional verification ensures that the designed logic behaves as described in the specification [9]. At algorithm level, the behavioral model is verified by simulating the model and examining the performance metrics or other measures depending on the application. As the high-level model is transferred to lower abstraction levels, the verification mainly checks that the functionality is equivalent with the higher level model. This can be accomplished by simulating both models with same stimulus and comparing their outputs, or the equivalence can be shown formally through a mathematical proof. In ASIC design, the role of functional verification is extremely important since even a small bug can make the chip unusable, which might lead to enormous costs.

2.3 Productivity

The exponential growth of digital systems increases the design and verification effort constantly. Increasing the engineering team sizes or extending the design cycle in the same vein is impractical, and hence the amount of logic designed per engineer has to increase. In other words, design productivity has to improve. There have been many technology advancements in the past that have improved productivity. Table 2.1 lists some of them and provides also estimates for their associated design productivity improvement [13]. Two methods that are especially relevant to this thesis work are design reuse and higher abstraction level.

Design reuse has been realized in the form of IP (intellectual property) blocks [47]. These IP blocks are design entities that implement certain functionality and can be instantiated multiple times in the system as well as reused in several chips. IP blocks can be relatively simple single-function designs such as FIFO (first-in first-out) buffers, but they can also comprise whole processor subsystems that are built of several smaller IPs. IPs typically use standardized bus interfaces such that they

Table 2.1 *Design technology improvements and their impact on productivity [13].*

DT Improvement	*Year*	*Productivity Delta*	*Productivity (Gates/Year/Designer)*	*Cost Component Affected*	*Description of Improvement*
None	1990		4K HW		
In-House Place and Route	1993	38.90 %	5.55K HW	PD Integration	Automated block placement and routing, transferred from the semiconductor house to the design team
Tall-Thin Engineer	1995	63.60 %	9.09K HW	Chip/circuit/PD verification	Engineer capable of pursuing all required tasks to complete a design block, from RTL to GDSII
Reuse—Small Blocks	1997	340 %	40K HW	Circuit/PD verification	Blocks from 2,500–74,999 gates
Reuse—Large Blocks	1999	38.90 %	56K HW	Chip/circuit/PD integration verification	Blocks from 75,000–1M gates
IC Implementation Suite	2001	63.60 %	91K HW, 87K SW	Chip/circuit/PD integration EDA support	Tightly integrated tool set that goes from RTL synthesis to GDSII through IC place and route
RTL Functional Verification Tool Suite	2003	37.50 %	125K HW, 87K SW	SW development verification	Tightly integrated RTL verification tool suite including all simulators and formal tools needed to complete the verification process
Transactional Modeling	2005	60 %	200K HW, 250K SW	SW development verification	Level above RTL, including both HW and SW design and consisting of behavioral (where the system function has not been partitioned) and architectural (where HW and SW are identified and handed off to design teams) levels
Very Large Block Reuse	2007	200 %	600K HW, 323K SW	Chip/circuit/PD verification	Blocks >1M gates; intellectual-property cores
Homogeneous Parallel Processing	2009	100% HW, 100% SW	1200K HW, 646K SW	Chip/circuit/PD design and verification	Many identical cores provide specialized processing around a main processor, enabling performance, power efficiency, and high reuse
Software Virtual Prototype	2011	300% SW	1200K HW, 2584K SW	SW development	Virtualization tools used to allow development prior to completed silicon
Intelligent Testbench	2012	37.5% HW	1650K HW, 2584K SW	System design and verification	Like RTL verification tool suite, but also with automation of the verification partitioning step
Reusable Platform Blocks	2013	200% HW, 100% SW	4949K HW, 5168K SW	Chip/circuit/PD verification	Fully functional platforms used as a block in larger platform design (e.g., ARM in OMAP)
Silicon Virtual Prototype	2015	100% HW	9897K HW, 5168K SW	System design and verification	A hardware virtualization platform that enables an RTL handoff of a SOC
Heterogeneous (AMP) Parallel Processing	2017	100% HW, 100% SW	19794K HW, 10336K SW	SW development verification	Many specialized cores provide processing around a main processor, which allows for performance, power efficiency, and high reuse
Many-Core SW Development Tools	2019	60% SW	19794K HW, 16537K SW	SW development	Enables compilation and SW development in highly parallel processing SOCs
Concurrent Memory	2021	100% SW	19794K HW, 33074K SW	SW development	Memories capable of on-chip memory management
System-Level Design Automation (SDA)	2023	60% HW, 37.5% SW	31671K HW, 45476K SW	System design and verification	Automates true system design on- and off-chip for the first time, including electronic, mechanical and other heterogeneous technologies
Executable Specification	2025	200% HW, 200% SW	95013K HW, 136429K SW	System design and verification	Describes the system specification in a manner that allows automated design and validation
Total		7920% HW, 21119% SW			

can be easily used in many different environments. Figure 2.4 represents an artistic view of the concept of building a chip out of several IP blocks.

It is a common practice nowadays to buy ready-made IPs from IP core vendors, and thus reduce the total design effort (see e.g. [8]). There are also open-source IPs available, especially if the functionality of the IP is needed frequently in many designs (see e.g. [31]). The IPs can be delivered in the form of *soft*, *firm* or *hard* IPs. The soft IPs are delivered as a RTL description that the user can modify to fit in their needs and synthesize to the target technology. The firm IPs are already

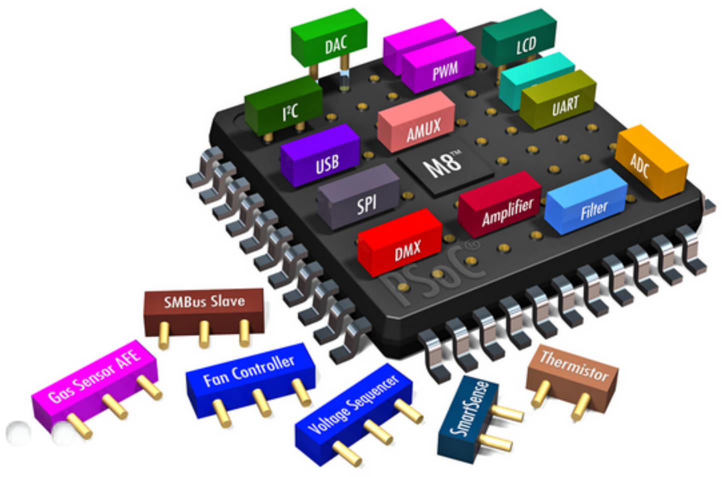


Figure 2.4 *Artistic view of IP-based design [3].*

synthesized to a gate-level netlist, but still allow small modifications. Hard IPs, on the other hand, are already implemented as a physical layout and cannot thus be altered. Hard IPs are typically memories and analog components, such as analog-to-digital converters or phase-locked loops that the silicon foundry provides to be used in a chip that they fabricate.

IP reuse has been a subject to extensive research (see e.g. [14]), and many methodologies have been developed based on the studies. One practical example is the IP-XACT standard [1] that defines a common XML (Extensible Markup Language) format for hardware component descriptions, which, for instance, eases the integration of IP blocks from multiple companies. The benefit of reuse can also be seen in Table 2.1 where it has been one of the most significant contributors to the design productivity improvement.

Another way to improve productivity is to design the logic at a higher abstraction level. In practice, this means that the functionality is described at the algorithm level, and automatically synthesized to a RTL description. This process is called high-level synthesis (HLS), which is the main topic of this thesis and will be described in more detail in the next chapter.

3. HIGH-LEVEL SYNTHESIS (HLS)

High-level synthesis, also known as behavioral or algorithm synthesis, is the process of converting a higher level algorithm description to a RTL architecture. Although HLS is only now becoming more widely used in the industry, its history dates back to the 1970s. Martin and Smith [21] divide the history of HLS tools into three generations. The first generation (1980s – early 1990s) was mostly used in research but was still significant for the future development of the tools. The second generation (mid-1990s – early 2000s) already found some real applications, but it was a commercial failure nonetheless. The main reasons for the lack of success were the poor quality of results and input languages that did not fit well into high-level modeling, and neither algorithm nor hardware designers considered them worthwhile to learn.

The current, third generation (starting from early 2000s) has demonstrated better results, and the tools have already been used in numerous real-world applications. Most of the present tools use some common algorithm modeling language, such as C/C++ or MATLAB, as their input language. This allows designers to use the untimed algorithm model as a starting point for their hardware design. Ideally, HLS would allow designers to use the algorithm model as such, but the tools still require guidance from the designer to produce the desired hardware architecture. Therefore, the designer has to also have a solid understanding of hardware design to achieve good results with the HLS tools.

Table 3.1 lists a few examples of modern high-level synthesis tools and some of their features, such as the target platform and input language. More detailed information about the tools can be found on the websites of the vendors that are referenced in the table. In addition to these, there are many other tools available. More thorough list of HLS tools can be found in references [27] and [23] that also include comparisons of the tools.

Table 3.1 Examples of modern high-level synthesis tools.

Tool	Vendor	Ref	Target platform	Input language
Catapult HLS	Mentor Graphics	[24]	ASIC/FPGA	C/C++/SystemC
CyberWorkBench	NEC	[28]	ASIC/FPGA	C
HDL Coder	Mathworks	[22]	ASIC/FPGA	MATLAB/Simulink
LegUp	Univ. of Toronto	[29]	FPGA (Altera)	C
Stratus HLS	Cadence	[5]	ASIC/FPGA	C/C++/SystemC
Synphony C Compiler	Synopsys	[41]	ASIC/FPGA	C/C++
Vivado HLS	Xilinx	[52]	FPGA (Xilinx)	C/C++/SystemC

3.1 Fundamentals

This section covers the fundamentals of HLS by going through the steps of a typical HLS process that is illustrated in Figure 3.1. This is a rather generic description of the flow, as in practice these steps are different for each tool, and their order of execution varies. Nevertheless, most of the tools go through these steps in some way or another [7].

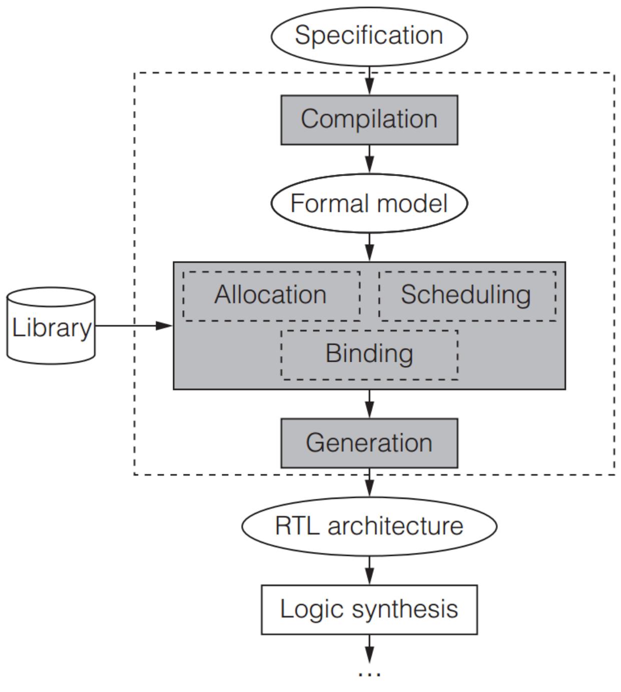


Figure 3.1 Flow diagram of the high-level synthesis process [7].

3.1.1 Compilation and optimization

The high-level code is first compiled to the internal format of the HLS tool. During compilation, the tool extracts information that is needed in the further processing of the design. For example, it might evaluate the dependencies between variables, the amount of loop iterations, and possible parallel structures in the code. Moreover, the tool checks that the code contains no structures that would be infeasible for hardware implementation, such as recursion, dynamic memory allocation and file operations.

At this point, the tool also optimizes the design by taking the hardware implementation into account. The optimizations might include removing parts of the code that are never executed, limiting the widths of the signals, or determining the minimum depth for buffers.

The compilation output is typically a data flow graph (DFG) that represents the data operations and their dependencies [7]. This is demonstrated with an example shown in Program 3.1. It implements a for-loop that gets three integer arrays – a, b, and c – as an input, and calculates the result of $a * b + c$ for each of the array elements. The extracted DFG is shown in Figure 3.2. The nodes in the graph represent data operations, and the edges are input, output or intermediate data.

```
for (int i = 0; i < 3; i++) {
     d[i] = a[i]*b[i] + c[i];
}
```

Program 3.1 *Example for-loop that gets three arrays – a, b, and c – with three integers, and calculates the result d for each of them.*

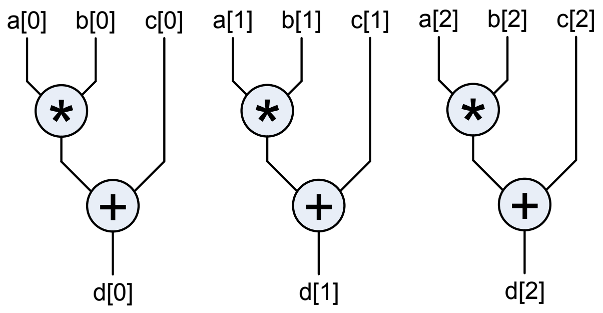


Figure 3.2 *Data flow graph of the example for-loop.*

Control dependencies are often also included to the graph, resulting in a control and data flow graph (CDFG). The CDFG consists of basic blocks that contain the data dependencies, and the connections between basic blocks depict the control flow.

3.1.2 Constraints

After the design has been compiled, user sets constraints to the design. The amount of control the user has depends on the tool. Some tools let the user specify only high-level targets in terms of area and latency, whereas others allow the user to affect the architecture in more detail.

The constraints often involve making trade-offs between different quantities, such as area, power, latency, and throughput. There is typically no single solution that would minimize all of these quantities simultaneously. Instead, there is a range of *Pareto optimal* solutions where one quantity can be improved only by degrading another [4]. Figure 3.3 demonstrates this for area and latency. The Pareto optimal solutions form a curve that is also called Pareto front. At any point of the Pareto front, the latency cannot be decreased without increasing the area, and vice versa. The grey area beyond the Pareto front contains solutions that are infeasible, and on the other side of the curve are the actual implementations. At both ends of the Pareto front, there is a point after which any increase in area (or latency) will not improve the latency (or area).

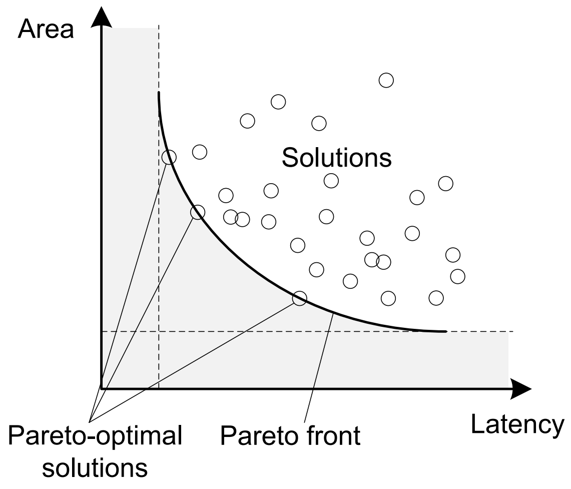
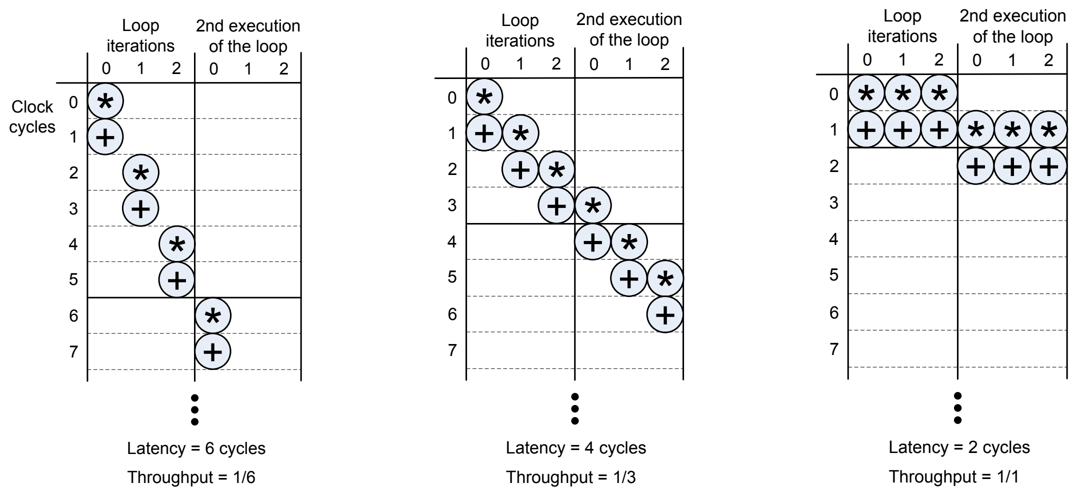


Figure 3.3 *Pareto front for area and latency.*

A typical user-defined constraint is the way loop structures are implemented. Loops can be pipelined or unrolled to optimize the latency and throughput of the design. The previously discussed for-loop example (Program 3.1) is used to demonstrate these loop optimizations. This example had no dependencies between the loop iterations, and hence there is a wide range of possible schedules. Figure 3.4 shows



(a) Sequential loop (b) Pipelined loop (c) Unrolled loop

Figure 3.4 *Three possible schedules for the example for-loop: (a) Sequential loop that corresponds most to the C code. (b) Pipelined loop where the multiplier and the adder can operate on every clock cycle. (c) Unrolled loop where all 3 array elements are processed simultaneously.*

3 different schedules. Figure 3.4(a) corresponds most to the original sequential for-loop. It calculates the product and the sum on separate clock cycles, and this is repeated for each set of integers to get the final results. Therefore, this implementation has latency of 6 clock cycles and can take new inputs every 6th cycle. In Figure 3.4(b), the loop is pipelined such that both the multiplier and the adder can operate on every clock cycle. Hence, this implementation can take new inputs every 3rd clock cycle, and the overall latency is 4 clock cycles. Figure 3.4(c) shows an unrolled loop, where the result is calculated simultaneously for each set of integers. This implementation requires two additional adders and multipliers, but can take new inputs every clock cycle, and the latency is only 2 clock cycles. This also demonstrates the Pareto optimality since the latency is decreased with the cost of additional operators (i.e. increased area).

Another main constraint is the target technology which limits the amount of operations that can be executed within one clock cycle and also the amount of components that can fit into a certain area. Therefore, most HLS tools require information about the technology library to be able to find the optimal hardware architecture within the given area and latency constraints. For ASIC designs, this information is usually provided by the silicon vendor in the form of a Liberty file which contains the timing, area and power characteristics of each standard cell, such as NAND gate or flip-flop. Some tools use the Liberty file as such during synthesis, whereas other tools create their own component library and synthesize each component with a 3rd party RTL synthesis tool to observe the correlation between area and timing.

3.1.3 Resource allocation, scheduling and binding

After the design has been constrained, the tool determines the resources that are needed to implement the functionality. At this point, the the tool concentrates mainly on the functional units, such as adders and multiplexers, but might also include registers, memories and connectivity components to the allocated resources. However, the amount of registers cannot be completely defined until the design is scheduled and the length of each register pipeline is known.

Next, the functional operations are scheduled into the states of the control logic, also known as control steps. Each control step takes at least one clock cycle but might also contain several cycles in case of a nested control logic, where the control step contains another control sequence. If the required operations cannot fit into a single control step, the tool divides them into several steps. The tool might also optimize area by using slower but smaller components divided into several control steps. However, each additional control step increases latency, and thus the tool considers also the latency constraints while optimizing the design.

The tool takes also the dependencies between variables into account during scheduling. For example, if some variable in a loop depends on the result of the previous loop iteration, and the user has pipelined the loop such that it should start a new iteration every clock cycle, all operations of the iteration must be scheduled within one clock cycle. If this is not possible with the target technology, the tool will inform the user that the design cannot be scheduled, and the user might have to modify either the source code or relax the loop constraints to solve the issue.

In binding phase, the tool examines the allocated resources and scheduling, and checks if some resources could be shared. For example, if the same adder type is used exclusively in two different operations, the adder could be shared between them. Therefore, these two operations can be bound to same physical resource. In addition to functional units, the storage elements, such as registers, might also be shared in case of variables having separate lifetimes. As the final result of binding, each operation, variable and connectivity element is bound to a physical resource.

Resource allocation, scheduling and binding are interdependent phases, and the tool would ideally consider these phases simultaneously or iterate between them [7]. However, this would lead to a very complex problem, and therefore the phases are typically executed in a sequence. Their ordering might vary depending on the optimization target. For example, when optimizing latency within a given area constraint, it would be preferable to allocate the resources before scheduling the design.

3.1.4 RTL generation

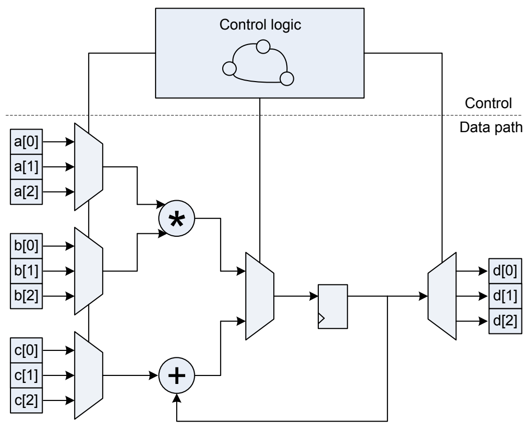
Finally, the tool generates the RTL architecture by using the results of resource allocation, scheduling and binding. The generated RTL code is typically divided into data path and control logic. In the data path, the tool instantiates the functional units and storage elements that were bound to the operations and variables. The control logic is realized with finite-state machines (FSM) that comprise the control steps derived during scheduling.

Continuing with the for-loop example (Program 3.1), Figure 3.5 represents possible RTL architectures for the 3 scheduling results shown in Figure 3.4. In Figure 3.5(a), the multiplier and the adder are shared for all loop iterations. Here the register is also shared such that it stores both the sum and multiplication results on separate clock cycles. The pipelined version in Figure 3.5(b) can also share the arithmetic units and registers between the loop iterations, but the registers cannot be shared within one loop iteration anymore since they have to be able to store the intermediate results every clock cycle. In Figure 3.5(c), there is no resource sharing at all. Although the control logic is not drawn in the figure, in practice there is still a trivial FSM that will indicate when the result is valid.

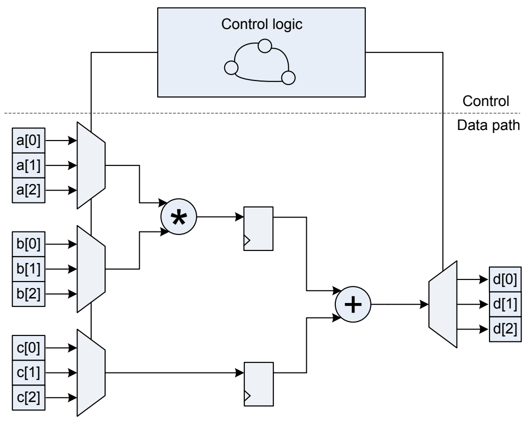
The final result is the synthesizable RTL code, typically in the format of VHDL or Verilog, depending on the tool. Generally, all of the code is generated into a single file, but it might also require compiling additional RTL libraries provided by the HLS tool vendor.

3.2 Advantages

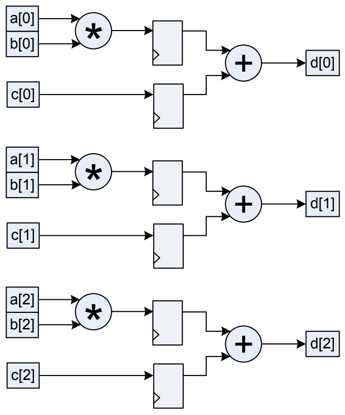
One of the main advantages in HLS is the increased productivity in frontend design, as the designer can focus on the functionality instead of implementation details. Table 3.2 lists results from previous case studies where HLS has been compared to traditional RTL design. Many of them have demonstrated 2–5 times higher design productivity compared to manually written RTL. In addition to effort estimations, the table shows the quality of results in terms of design area (or resource utilization for FPGAs) and maximum clock frequency f_{max} that can be achieved with the



(a) Sequential loop



(b) Pipelined loop



(c) Unrolled loop

Figure 3.5 *RTL architectures for the three schedules of the example for-loop. (a) Sequential loop that corresponds most to the C code. (b) Pipelined loop where the multiplier and the adder can operate on every clock cycle. (c) Unrolled loop where all 3 array elements are processed simultaneously.*

design. The percentages stand for the increase or decrease of the respective quantity in the HLS design compared to the manually coded RTL. Within these studies, the area difference varies from -38% to $+173\%$, and f_{max} difference ranges from -29% to $+10\%$.

The design effort is reduced also by the fact that the high-level description is at the same abstraction level with the algorithm model. Hence, it may be possible to reuse parts of the model in the HLS design, especially if the HLS tool uses same input language with the model. This is not possible with RTL code, as its representation differs greatly from the behavioral model.

Table 3.2 *Quality of results and design effort of HLS compared to hand-written RTL in several case studies.*

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

Another advantage of HLS is easier and faster design space exploration. Designers can easily try several alternatives for the architecture by using the same high-level code and modifying only the directives provided to the tool during the synthesis process, such as the loop pipelining and unrolling. This is significantly faster than trying different architectures at RTL, where in the worst case, most of the code would have to be rewritten.

The algorithm-level code is typically technology-independent, which allows using the same code to synthesize RTL architecture for different target technologies. Since the optimizations are done during the synthesis process based on the target technology, the resulting RTL code will always be optimized to that particular technology. Although it is also possible to use the same hand-written RTL code with several technologies, the resulting hardware would not be optimal without manually adjusting pipelining in the code. However, some HLS tools are vendor-specific such that they can generate optimal RTL only for the FPGAs of certain vendor. This was seen in Table 3.1 where Vivado HLS, for example, could target only Xilinx FPGAs.

The higher abstraction level accelerates also the functional verification since the algorithm-level simulation is considerably faster than RTL simulation. Moreover, the high-level model can be described with fewer lines of code, which not only consumes less time, but also decreases the chance of errors in the code. With verification being one of the largest bottlenecks in the ASIC design today [48], this is one of the most attractive features of HLS. Nevertheless, for the high-level verification to be sufficient, the equivalence of the algorithm description and generated RTL should be proven. Fortunately, many tools provide ways to do this either formally or through co-simulation of the high-level and RTL codes.

3.3 Problem areas

In spite of its many advantages, HLS imposes also some problems which are often related to the fact that the designer has less control on the final architecture. This might lead to a less optimal hardware, as the tool creates more complex logic than expected. Previous research has demonstrated large variation in the quality of results, which can be seen in the case studies listed in Table 3.2. For example, the amount of LUTs utilized in the signal correction block [18] was 173% larger compared to hand-written RTL, whereas the DMA controller [53] had 38% smaller area in the HLS design. This sort of variation makes it difficult to estimate the area for the floorplan of the ASIC. Moreover, it is possible that a small change in the high-level code has a huge impact on the RTL code and consequently on the physical area. If this happens late in the design flow, it might require changing the floorplan of the whole chip and delay the tapeout.

The limited control makes it also difficult to design logic that requires exact timing or complex control logic. This is complicated further by the fact that the tools use input languages that are executed sequentially. As the tool tries to create hardware that matches the sequential logic, there are cases where it is not possible to describe the desired functionality with the high-level language. Some tools support mixed-language design where part of the logic can be included in the design as a RTL description. However, this makes the simulation of the design more complex as RTL simulation differs greatly from the way the high-level code is simulated.

The input languages and synthesis processes for HLS are not currently standardized. Instead, every tool has its own way of processing the high-level code. Some tools require pragmas in the code that guide the tool in the synthesis, while others rely mostly on a GUI-based approach, where the instructions are given by the user during the synthesis process. This causes a problem where the high-level code can be processed only by the targeted HLS tool, and if the tool is later changed, the code would have to be refactored. This will consequently lead to vendor lock-in where chip manufacturer has to keep using the same tool, as the code modifications would likely consume a significant amount of time. In contrast, VHDL and (System)Verilog are standardized, vendor-independent languages.

The wide range of applications is also a common challenge for HLS tools. For example, control logic differs greatly from signal processing applications and requires different kind of algorithm description. Furthermore, the targeted platform affects the architectural choices, since ASIC and FPGA require a different kind of RTL description to achieve optimal hardware. Therefore, the HLS tools typically focus on a certain application domain. Thus, if the manufacturer wants to design products on both ASIC and FPGA, and has a broad range of applications, it might have to purchase licenses for several HLS tools, which may build up to excessive costs.

The RTL code generated by the HLS tools is generally not human-readable since it is intended to be parsed only by simulators and RTL synthesis tools. The generated RTL code may contain hundreds of thousands of lines in a single file, the signal and entity names are often very abstract, and the structure of the code is difficult to follow. Thus, synthesis and linting logs are difficult to read due to the long signal and entity names, and the code is difficult to debug if bugs are found in the RTL simulation. Fortunately, some tools allow cross-probing between the RTL and high-level codes to ease the debugging process.

4. CASE STUDY

The HLS flow and the quality of backend results are evaluated using an example design that resembles a real use case in IP-based ASIC design. The designed IP follows a typical structure of an IP block that is shown in Figure 4.1. It includes two streaming interfaces for data, a bus interface for software configuration, wrapper for memories, and a core with the main functionality of the IP. Only the IP core is created with HLS since the other components are either available as common modules or generated with in-house tools. The IP that was chosen as a basis for this work has an existing RTL implementation, which allows comparing the backend results and the design effort of the HLS-generated and manually coded RTL designs.

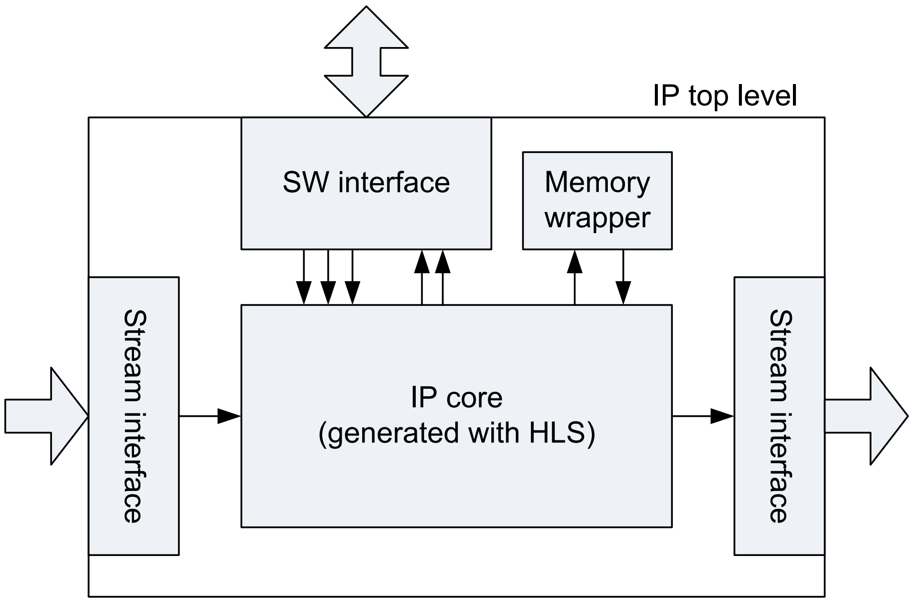


Figure 4.1 *Typical structure of an IP that contains data and configuration interfaces, memory wrapper, and the core with the main functionality of the IP.*

4.1 Decimator

The IP block used as a test case is a decimator, which is a commonly used component in digital signal processing (DSP) applications that operate with multiple sample rates. The main functionality of a decimator is to decrease the sample rate of the signal by a decimation factor M. This means that the decimator passes only every Mth input sample to the output. However, simply downsampling a signal would cause distortion since the frequency components above the Nyquist frequency of the decimated signal would be misinterpreted as low-frequency components. To avoid this *aliasing*, the signal is low-pass filtered[1] before downsampling [25], as shown in Figure 4.2. There are many ways to implement the anti-aliasing filter, the choice depending on the application. The filter types used in this design are cascaded integrator-comb (CIC) filter and polyphase decimation filter.

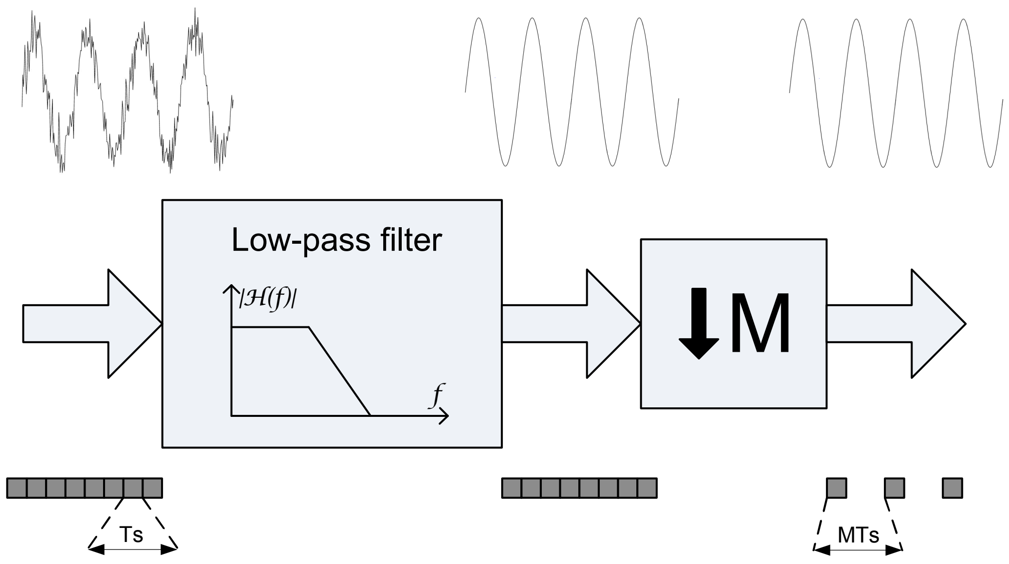


Figure 4.2 *Decimation is a combination of filtering and downsampling. T_s denotes the sampling interval of the signal.*

4.1.1 Cascaded integrator-comb filter (CIC)

CIC filter is a common filter type used in applications that require configurable decimation rate [12]. Figure 4.3 shows the typical structure of a CIC filter. It consists of a cascade of integrators followed by downsampling and an equally long cascade of combs that are essentially digital differentiators. The number of integrator and comb stages affects the attenuation of the aliasing spectral components such that increasing the number of stages improves the attenuation. As all the arithmetic operators in a CIC filter are either adders or subtractors, the hardware cost is small compared to regular FIR filters with multipliers. Another advantage of the CIC filter is that its frequency response depends on the decimation rate such that the

[1]Bandpass filters can also be used in decimation, in which case the aliasing is exploited to translate the bandpass signal to a lower frequency [2].

largest attenuation will always occur at the frequency components that would alias at zero frequency, which often is the frequency where the desired signal is located. Therefore, it lends itself to applications with configurable decimation rate since the only part that has to be configured is the downsampler.

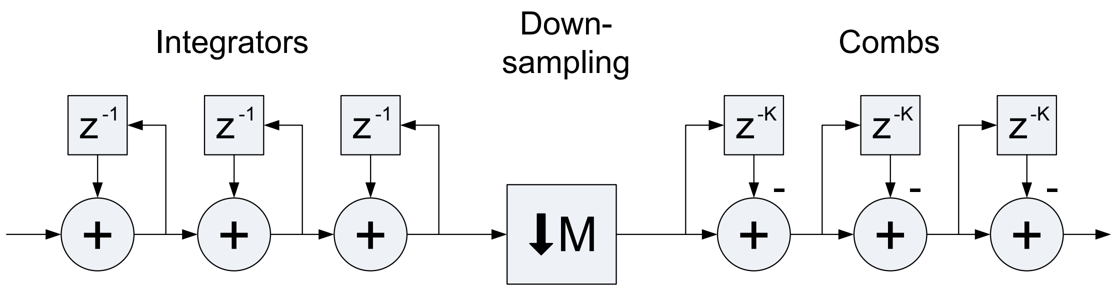


Figure 4.3 *Decimating CIC filter with three integrator and comb stages.*

The CIC filter is a convenient component for evaluating the optimization capabilities of HLS tools. As the cascade of adders or subtractors does not necessarily require any registers between them, the tool would have to figure out the amount of additions and subtractions that can be fitted within one clock cycle and decide the optimal pipelining. Therefore, the CIC filter is a good test case for evaluating the quality of the technology library characterization.

4.1.2 Polyphase decimator

As shown in Figure 4.2, decimation can be done simply with a low-pass FIR filter followed by a downsampler. However, this structure contains excess computation as the filter output is calculated also for samples that are eventually dropped. Therefore, decimating FIR filters are often implemented using a polyphase structure [25] that is shown in Figure 4.4. The name polyphase refers to the fact that the samples are divided into several branches, each being downsampled at different phases of the signal. In this example, the decimation rate is two, and thus the input samples are divided into two branches such that even samples go to the upper branch, and the odd samples to the lower. Filter $H_0(z)$ contains the even coefficients and $H_1(z)$ the odd coefficients of the FIR filter that would have been used in the basic decimator structure. Polyphase decimators have a fixed decimation rate, and their implementation requires multipliers, but it is easier to achieve the desired frequency response with them compared to a CIC filter [36].

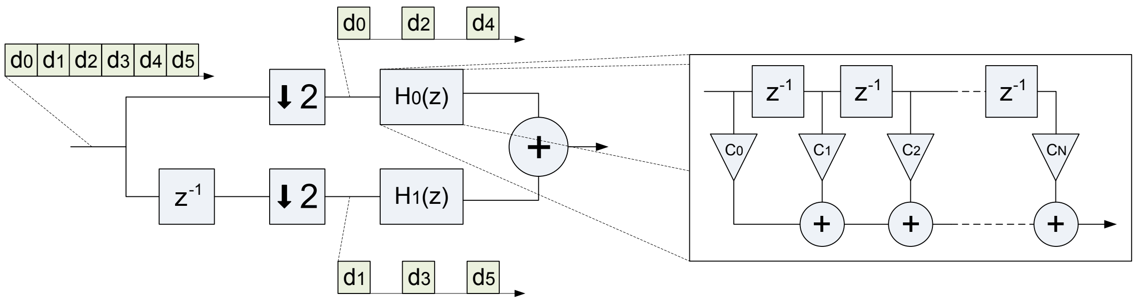


Figure 4.4 *Two-branch polyphase decimator with decimation factor of 2. Both $H_0(z)$ and $H_1(z)$ contain a FIR filter that is shown on right.*

4.1.3 Arbiter

The decimator differs from the generic IP shown in Figure 4.1 by having more than one data stream input. These are sample-based data streams that can transmit several channels of data by time-division multiplexing the samples of the different channels to a single data bus. In the input of the decimator, the data streams have high sample rates, and only few logical streams can fit in a single physical stream. As the data streams are decimated to a lower sample rate, streams can be combined to convey more channels. This is done with an arbiter that interleaves samples from two data streams as shown in Figure 4.5. This arbiter follows a priority scheme such that the samples in stream 0 are passed to the output by default, and the samples from stream 1 are passed only when there are no samples arriving from stream 0. To avoid losing the samples of stream 1, there is a FIFO buffer that stores the samples when the context of the arbiter is on the stream 0.

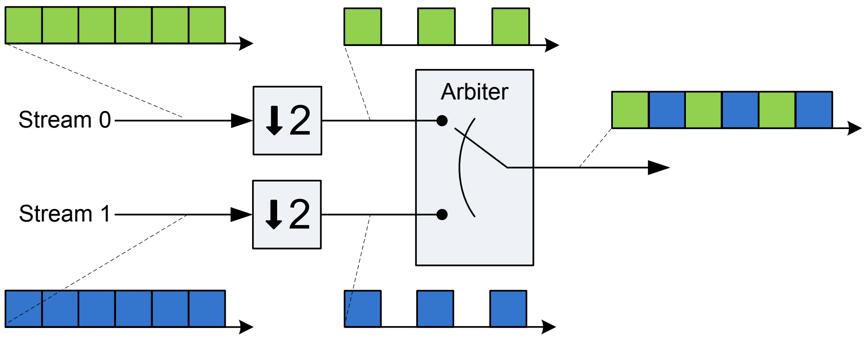


Figure 4.5 *Arbiter interleaves samples from two data streams into a single stream after decimation.*

4.1.4 Packager

Finally, as the data streams have been decimated, the samples have to be packed such that they follow the data stream protocol that is used to connect the decimator to the next IP. The protocol requires that the samples are sent in packets of 4 samples as a burst, and all samples in the packet must belong to the same channel. Due to decimation, however, the samples in the output of the final decimating subblock are in a seemingly random order, and hence the samples have to be organized to the packets of 4 samples by the help of a packager. The block diagram of the packager is shown in Figure 4.6 that also demonstrates the timing of the input and output samples. For each channel, there is a FIFO buffer where the incoming samples are stored. Once a buffer has 4 or more samples, they are sent to the output as a burst.

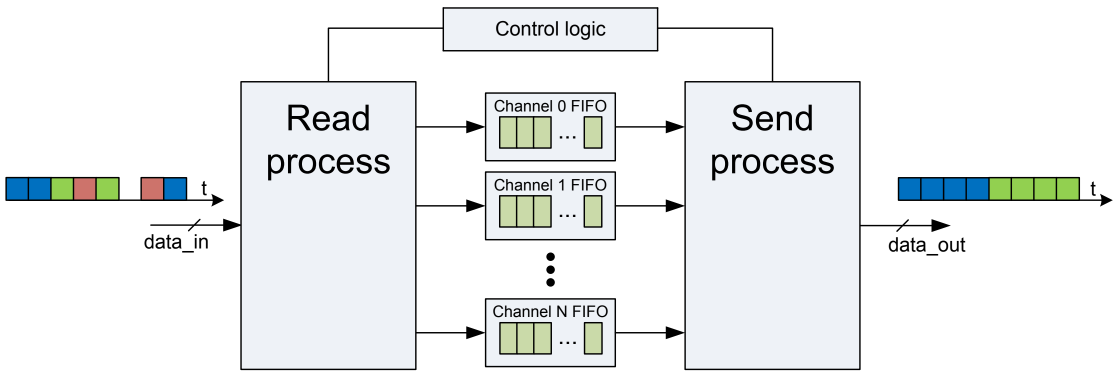


Figure 4.6 *Packager that organizes the samples into packets of four samples. Timing diagrams are also shown at the input and output where the colors represent different channels.*

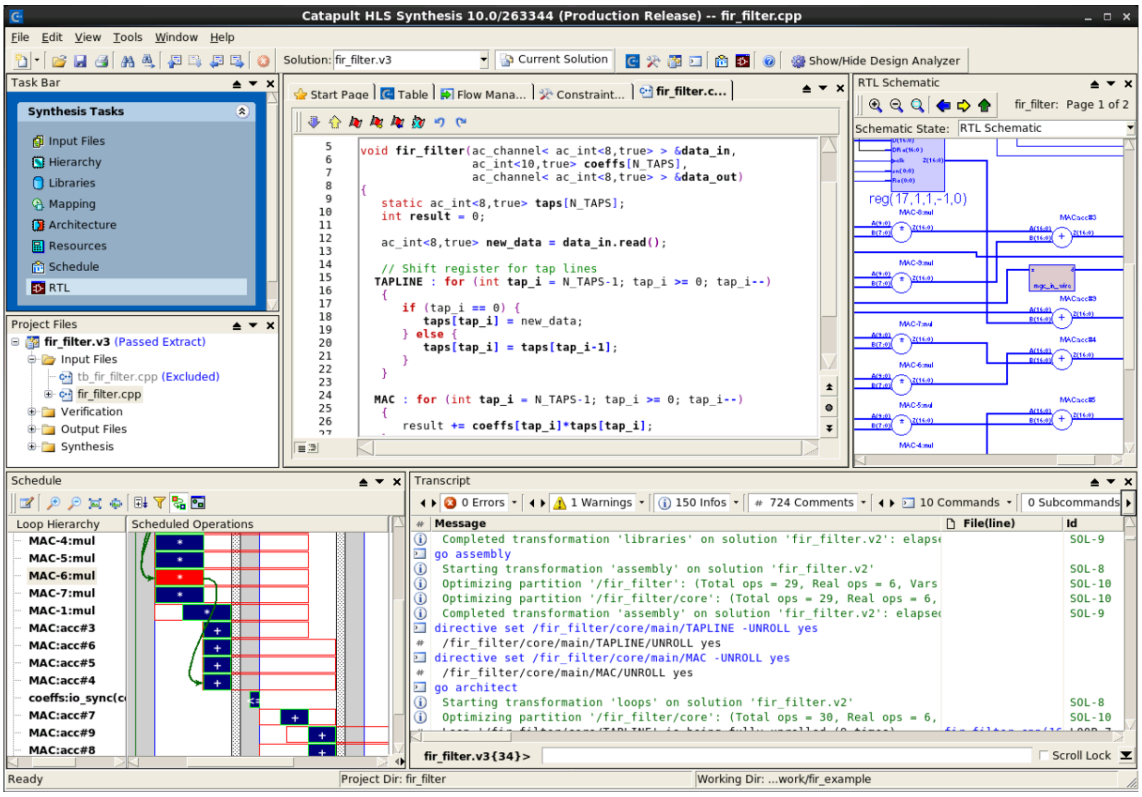
4.2 Catapult HLS

The HLS tool used in the evaluation is Catapult HLS by Mentor Graphics [24]. The tool is targeted for both ASIC and FPGA designs. It uses a large subset of ANSI C++ and SystemC as its input language, and the output language of the generated RTL can be selected as either VHDL or Verilog. The code written for the tool is generic C++ such that it can be compiled with most C++ compilers.

Catapult supports both SystemC (SC) and Algorithmic C (AC) data types for bit-accurate number representations. The AC data types, for example, contain integer (ac_int), fixed-point (ac_fixed), and complex number formats (ac_complex), which makes them convenient especially for DSP applications. They also handle rounding and saturation automatically when assigning to a variable with a different data format, and the user can choose the rounding and saturation behavior from a wide range of options. However, the downside of these data types is that they reduce the reusability of the code since other HLS tools might not support them.

4.2.1 User interface

The instructions and constraints are given to Catapult via a GUI (graphical user interface) that guides the user through the whole synthesis process (see Figure 4.7). Code pragmas can also be used to guide the tool, but it is often preferable to leave the source code untouched and give the instructions through the GUI. Furthermore, Catapult supports TCL (tool command language) scripting such that each setting given in the GUI has an equivalent TCL command. During the synthesis process, the tool stores each instruction into a TCL script, which allows the user to later repeat the whole process by simply executing the script.



```
void fir_filter(ac_channel< ac_int<8,true> > &data_in,
                ac_int<10,true> coeffs[N_TAPS],
                ac_channel< ac_int<8,true> > &data_out)
{
  static ac_int<8,true> taps[N_TAPS];
  int result = 0;

  ac_int<8,true> new_data = data_in.read();

  // Shift register for tap lines
 TAPLINE : for (int tap_i = N_TAPS-1; tap_i >= 0; tap_i--)
  {
     if (tap_i == 0) {
        taps[tap_i] = new_data;
     } else {
        taps[tap_i] = taps[tap_i-1];
     }
  }

 MAC : for (int tap_i = N_TAPS-1; tap_i >= 0; tap_i--)
  {
     result += coeffs[tap_i]*taps[tap_i];
```

```
#  Message                                                                       File(line)  Id
    Completed transformation 'libraries' on solution 'fir_filter.v2': elapse                 SOL-9
   go assembly
    Starting transformation 'assembly' on solution 'fir_filter.v2'                           SOL-8
    Optimizing partition '/fir_filter': (Total ops = 29, Real ops = 6, Vars                  SOL-10
    Optimizing partition '/fir_filter/core': (Total ops = 29, Real ops = 6,                  SOL-10
    Completed transformation 'assembly' on solution 'fir_filter.v2': elapsec                 SOL-9
   directive set /fir_filter/core/main/TAPLINE -UNROLL yes
#   /fir_filter/core/main/TAPLINE/UNROLL yes
   directive set /fir_filter/core/main/MAC -UNROLL yes
#   /fir_filter/core/main/MAC/UNROLL yes
   go architect
    Starting transformation 'loops' on solution 'fir_filter.v2'                              SOL-8
    Optimizing partition '/fir_filter/core': (Total ops = 30, Real ops = 6,                  SOL-10
fir_filter.v3{34}>
```

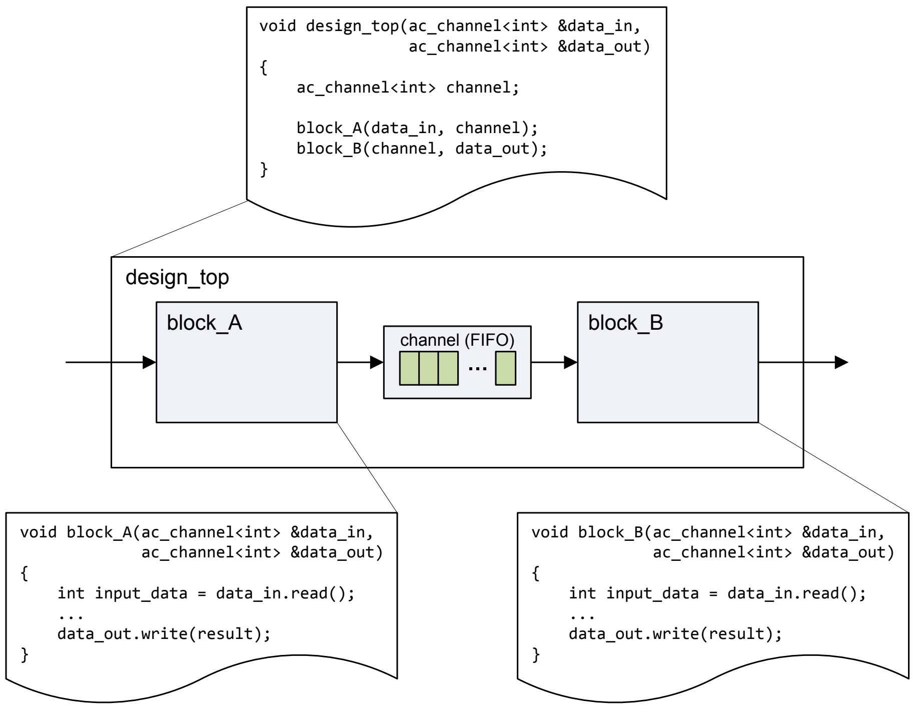
Figure 4.7 The user interface of Catapult HLS.

The user interface lets user to control a wide range of options. The most elementary options are clock frequency and polarity, and reset type (synchronous, asynchronous or both) and polarity. In addition, the user can define clock uncertainty and duty cycle. It is also possible to set multiple clocks to different hierarchical blocks, in which case the tool will replace the ordinary FIFO buffer between the blocks with a clock domain crossing (CDC) FIFO.

The architectural constraints let the user choose if the loops are pipelined or unrolled, and whether the data arrays are implemented as flip-flops or memories. In addition to these, there are plenty of other options, some of them rather detailed. The user can, for example, adjust the scheduling afterwards by moving the operations from control step to another.

4.2.2 Hardware interfaces

The C++ function parameters and return values are synthesized to hardware interfaces that can be realized as simple wires, or as streaming or memory-mapped interfaces. The streaming interfaces are created with ac_channels (included in AC data types) that can be mapped to a simple handshaking interface with ready/valid signaling. This provides easy connectivity to other IPs and also allows creating hierarchical designs where two subblocks are connected via ac_channel. The channel between the subblocks can contain a FIFO for buffering data samples. This has been demonstrated in Figure 4.8 where subblocks *block_A* and *block_B* are connected in the top-level function via an ac_channel.



```
void design_top(ac_channel<int> &data_in,
                ac_channel<int> &data_out)
{
    ac_channel<int> channel;

    block_A(data_in, channel);
    block_B(channel, data_out);
}
```

```
void block_A(ac_channel<int> &data_in,
             ac_channel<int> &data_out)
{
    int input_data = data_in.read();
    ...
    data_out.write(result);
}
```

```
void block_B(ac_channel<int> &data_in,
             ac_channel<int> &data_out)
{
    int input_data = data_in.read();
    ...
    data_out.write(result);
}
```

Figure 4.8 *Example of the use of ac_ channels in hierarchical design.*

The memory-mapped interfaces are created by using C arrays with fixed length. If the array is a parameter of the top-level function, Catapult will synthesize a memory interface with data and address buses, and read/write enable signals. Two subblocks can also use the same C array for communication, in which case Catapult instantiates a shared memory between them. The user can define, for example, the way the C array indices are mapped to memory addresses and the amount of data samples in a single memory address.

There are also other interfaces options. For example, control data can be provided as a direct input that has no handshaking or synchronization. This is typically used only for static control values because a change in direct input produces a different outcome in the untimed model and the synthesized RTL code, which would lead to failures in the co-simulation of these two models.

4.2.3 Verification

Catapult has an integrated verification flow where it automatically generates compilation scripts for running the tests for both the high-level model and the generated RTL. The user has to only write the testbench logic in C++ (or SystemC), which is rather straightforward as the testbench and the design are both written in the same language and at the same abstraction level.

In the beginning of the synthesis process, the C++ testbench is used to verify the functionality of the design. This is much faster compared to a typical RTL simulation and allows quick iterations in debugging of the high-level code. During the synthesis process, Catapult automatically generates a RTL testbench that uses the C++ testbench as a reference. This RTL testbench is mainly used to ensure that the generated RTL corresponds to the high-level functionality by co-simulating it with the C++ testbench. The verification aspect of high-level synthesis has been covered in more detail in Tulla's thesis work [46].

5. RESEARCH QUESTIONS

The purpose of this thesis is to evaluate the suitability of HLS-generated RTL code for several tools and flows related to backend design. Following topics are covered:

- Technology library characterization
- Technology library abstraction
- Design-for-testability (DFT) structures in RTL code
- Engineering change order (ECO)
- Static code analysis
- Logical equivalence checking (LEC)
- Area and timing
- Power efficiency

This chapter introduces these topics and provides evaluation criteria for each of them. The results are discussed in the next chapter.

5.1 Technology library characterization

The HLS process in Catapult uses its own component library that contains basic building elements, such as logic gates, registers, multiplexers, and adders. The tool requires area and timing information for all of these components to schedule the design and allocate resources properly. This information is provided to the tool by characterizing the target technology, for which Catapult provices a library builder tool that will be evaluated in terms of the required time and effort.

The characterization process might have to be repeated several times if the area and timing characteristics of the component library do not correlate with the RTL synthesis results. For example, if the characterized component delays are too optimistic, the physical implementation of the HLS-generated logic might contain timing violations. Moreover, technology libraries might get updated during the design process, which requires repeating the characterization process. Hence, the flow should be automated and repeatable with little manual effort.

5.2 Technology library abstraction

Although designs are generally technology-independent until RTL synthesis, there are certain technology components that have to be instantiated already in the RTL code. Typical examples of these sort of components are memories and components used in clock domain crossings (CDC), such as synchronizers.

It is a common practice in RTL design to abstract technology components by creating wrappers for them with generic interfaces. In this way, the designer will only have to instantiate these wrappers in the design, and the mapping to the technology is done inside the wrapper. If the design is later implemented using different technology, only the wrappers have to be updated, and the code of the actual IP can be used as such (assuming that the RTL behavior of the technology component has not changed). This is illustrated in Figure 5.1 that shows a typical use case for technology abstraction. In this example, the IP contains several memories that are instantiated using a wrapper with a generic interface. There are wrappers available for two different technologies, and depending on the technology which the IP is targeted to, one of them is selected for the compilation.

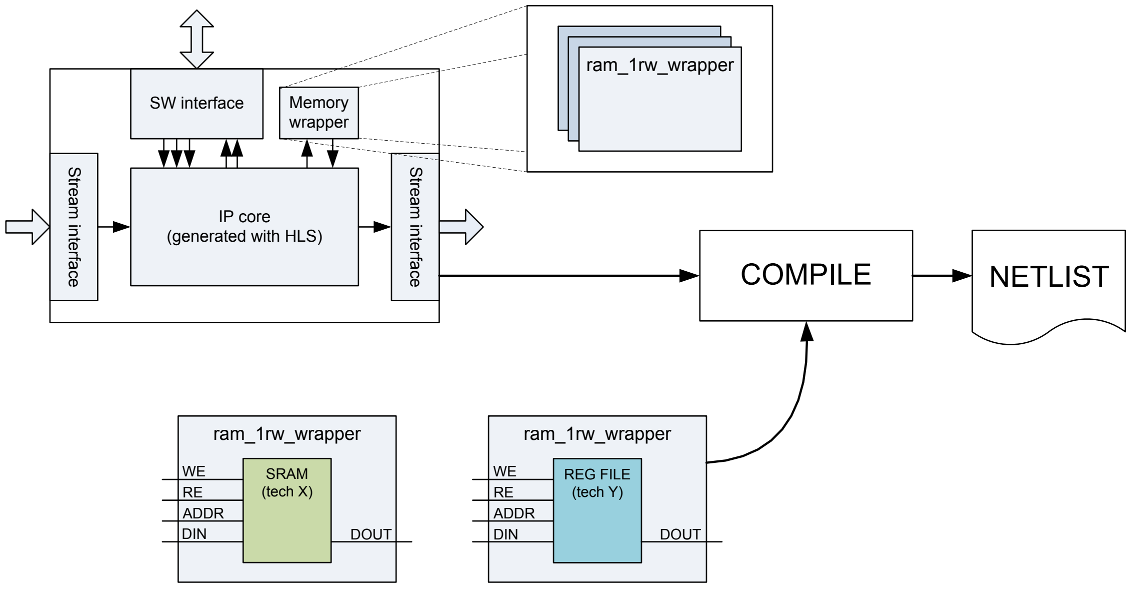


Figure 5.1 *Memory wrappers are used in the design to abstract the technology components. In this example, there are two implementations for the wrappers, and the choice depends on the target technology.*

Technology abstraction also allows using generic technology-independent components that model the behavior of the technology component and share the same interface with the corresponding wrapper. For example, a memory could be modeled simply as an array of flip-flops. These technology-independent components can be used in the first simulations before the technology components and their wrappers are available.

HLS has generally two ways to instantiate the technology components in the generated RTL. One way is to provide the tool with the information about the technology components, and it instantiates them as such to the RTL code. The other way is to import the wrappers to the tool, and the mapping to technology components is done during the compilation of the RTL code. The latter is more flexible way as it allows using the generic technology-independent components before the wrappers are available. However, when using the wrappers, the tool has no information about the timing and area characteristics of the technology components, which makes it more difficult to optimize the hardware around these components.

This thesis will investigate the possibilities for technology abstraction in Catapult, and best practices will be developed based on the trials with the example design. The flow will be evaluated based on the effort required to both import technology components to the tool and use them in the design.

5.3 Design-for-testability (DFT) structures in RTL code

Each chip that is fabricated has to be tested for faults that might occur due to variations in the manufacturing process. These faults include, for example, nodes stuck at 0 or 1, and shorted and open connections. Simply running the functional tests for each chip is slow, and thus the chips must have dedicated testing structures embedded to speed up the testing. Improving the testability in this way is called design for testability (DFT).

The most common DFT structure in digital circuits is a scan chain [48]. An example of a scan chain is shown in Figure 5.2 that represents a circuit of 4 flip-flops with some combinatorial logic between. The scan chain is used here to check the combinatorial logic for faulty gates. It is constructed by adding 2-input multiplexers in front of each flip-flop and connecting them in a chain, essentially forming a long shift register. Now, the flip-flops can be loaded with arbitrary values by enabling the scan chain (by setting scan enable to 1) and inserting the values as a bit stream through the scan data input. Once the desired values have been loaded, the output of the combinatorial logic is captured to the flip-flops by disabling the scan chain and sending a single pulse to the clock input. The output values can then be read from the scan chain output by enabling the scan chain again. Finally, the bit vector that was read is compared to a golden vector to determine if there were faults in the circuit.

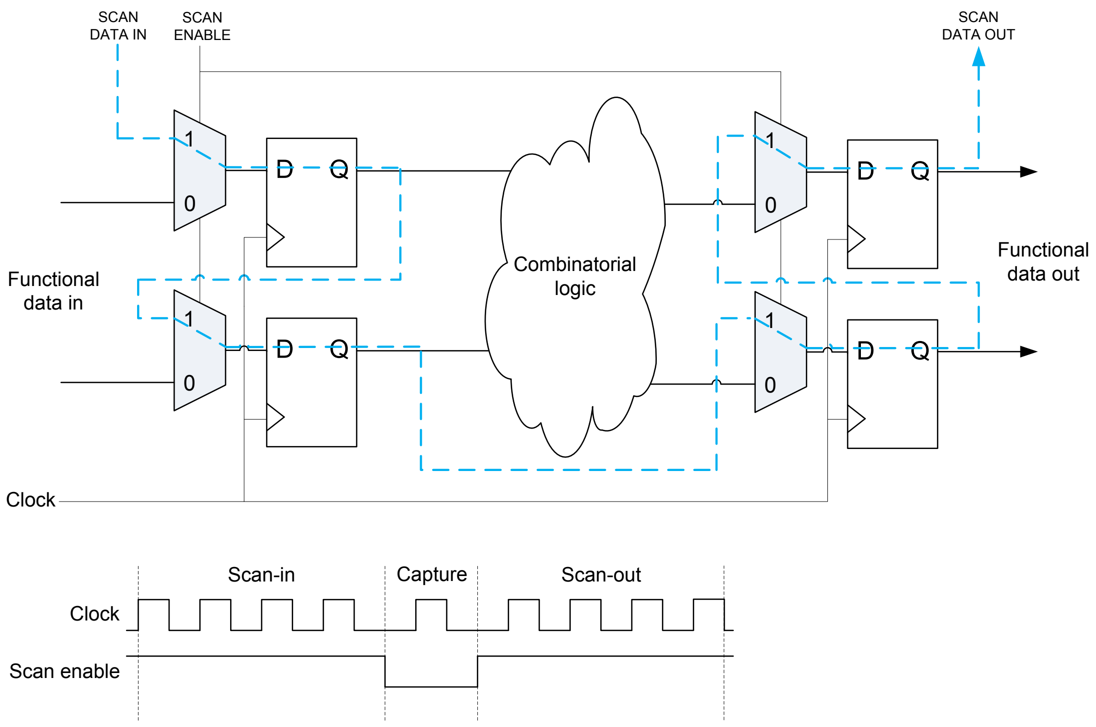


***Figure* 5.2** *Scan chain inserted to a circuit.*

The scan chain is often done automatically to the gate-level netlist, and thus the designer does not have to implement it in the RTL code. The scan insertion is done by replacing all flip-flops in the design with scan flip-flops that include also the multiplexer and the scan data and enable inputs in addition to the regular flip-flop [48]. The test synthesis tool takes also care of the chaining of the flip-flops.

Although the scan chain is implemented automatically, there are some cases where DFT has to be considered already in RTL design. For example, memories might have built-in self test (BIST) which requires routing the related control ports to the top level of the IP. Moreover, clock and reset manipulation (e.g. clock division and gating) requires attention regarding DFT. For instance, if a clock gate is controlled by a flip-flop, loading the data through the scan chain will occasionally disable the clock gate, preventing the scan data from flowing through the other, gated flip-flop. Hence, the clock gates must have an additional control for test mode that bypasses the clock gate.

This thesis will study the possiblity of inserting DFT structures in the RTL code generated by the HLS tool. The required effort will also be evaluated.

5.4 Engineering change order (ECO)

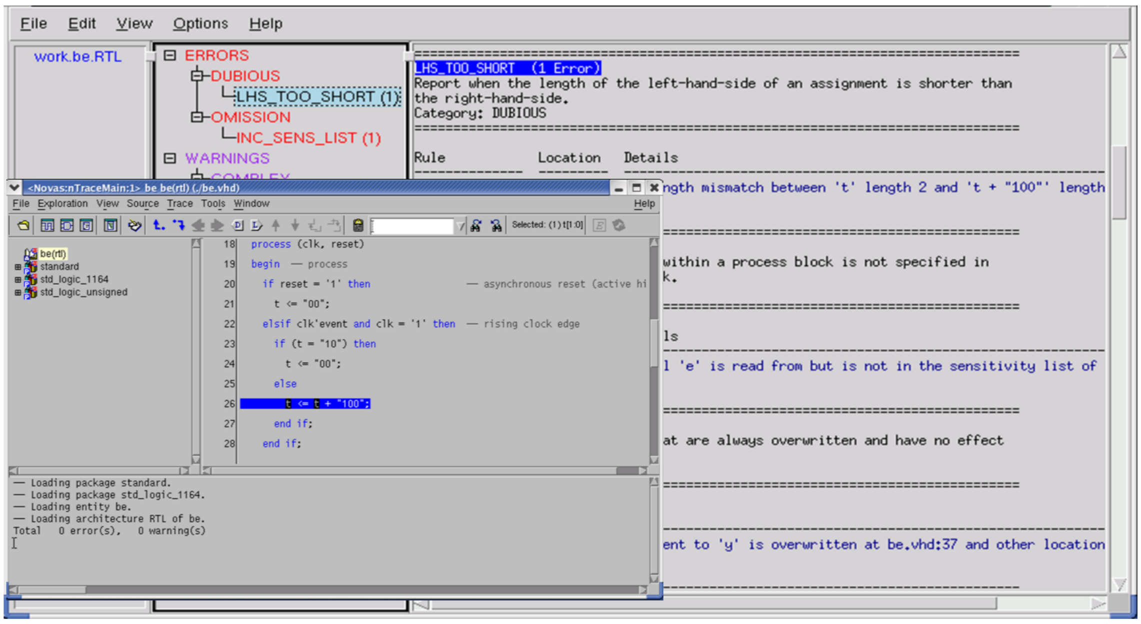
Since the physical implementation of the chip starts well before all verification is completed, it is possible that bugs are found after placing and routing the design. Moreover, specifications may change at a late stage of the design flow which requires modifying the RTL code and consequently the gate-level netlist and layout. Repeating the whole backend flow for these changes is costly and delays the tapeout of the chip, or requires a new tapeout if the masks have been already created. Hence, the changes are often done to the physical netlist as local modifications called engineering change orders (ECOs) [20]. ECOs can be done to the netlist manually or with a tool that observes the changes in RTL or gate-level netlist and automatically creates a patch for the existing physical netlist. After the change is done, the netlist is verified against the modified RTL description with a logical equivalence checking tool.

Unfortunately, HLS complicates the ECO flow due to the increased distance between high-level algorithm and its physical implementation. Even a small change in the algorithm code may cause massive changes in the gate-level netlist. Therefore, the ECO flow is one of the main concerns of HLS in ASIC design. Although it has been claimed that ECOs are rare with HLS due to the accelerated verification [42], it is still possible that specifications change. Hence, HLS tools targeting ASICs should provide some means to minimize the RTL changes.

Incremental high-level synthesis has been proposed to ease the ECOs in HLS flow [20]. This has been already taken into use in a few HLS tools, and Catapult has also implemented an incremental flow. This flow will be evaluated in this thesis by observing both the ease of use and the resulting changes in the RTL code. The scope of the changes will also be examined to see if the alterations are localized or spread all around the design.

5.5 Static code analysis

Static code analysis is used to check the RTL code for structures that might introduce bugs or other issues in the later verification and physical implementation phases [44]. The code analysis is carried out with a linting tool that reads the RTL code and ensures that it fulfills all the given rules. These rules detect issues such as unintentional latches, combinatorial loops, flip-flops without reset, and unsynthesizable structures. Examples of linting tools used in the industry are Ascent Lint by Real Intent [33] and SpyGlass Lint by Synopsys [40]. Figure 5.3 shows an example of a linting tool reporting a length mismatch in a signal assignment.



```
==========================================================================
LHS_TOO_SHORT   (1 Error)
Report when the length of the left-hand-side of an assignment is shorter than
the right-hand-side.
Category: DUBIOUS
==========================================================================

Rule             Location   Details
---------------  ---------  ----------------------------------------------
ngth mismatch between 't' length 2 and 't + "100"' length

==============================================

within a process block is not specified in
k.

==============================================

ls
----------------------------------------------
l 'e' is read from but is not in the sensitivity list of

==============================================

at are always overwritten and have no effect

==============================================


----------------------------------------------
ent to 'y' is overwritten at be.vhd:37 and other location

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

```
process (clk, reset)
begin  -- process
  if reset = '1' then                        -- asynchronous reset (active hi
    t <= "00";
  elsif clk'event and clk = '1' then  -- rising clock edge
    if (t = "10") then
      t <= "00";
    else
      t <= t + "100";
    end if;
  end if;
```

```
-- Loading package standard.
-- Loading package std_logic_1164.
-- Loading entity be.
-- Loading architecture RTL of be.
Total   0 error(s),   0 warning(s)
```

Figure 5.3 *Ascent Lint reporting a length mismatch in a signal assignment in RTL code [33].*

Static code analysis is typically used to check the RTL code before functional verification or RTL synthesis such that potential bugs are spotted before proceeding to these lengthy processes, and thus avoiding unnecessary iteration loops in the design flow. It is often the responsibility of the designer to run the RTL linting and check all the reported warnings. Projects often have a requirement that the linting logs have to be completely clean, and therefore the designer will either have to fix the issue that caused the warning or create a waiver with a good reasoning.

It is a good practice to run the static code analysis also for the tool-generated RTL code, although the designer has little influence on the coding style. In this way, the possible bugs in the HLS tools are spotted and can be reported to the tool vendor. However, if the linting tool gives warnings, it is difficult to fix the issue in the generated code, and hence the only practical way to have clean linting logs is by creating waivers. Since it is often a time-consuming task to create the waivers, it would be preferable to have the tool generate as clean RTL code as possible.

5.6 Logical equivalence checking (LEC)

As the design is processed further, for example through RTL synthesis, the behavior of the design has to stay the same. In other words, the two representations (e.g. RTL code and gate-level netlist) have to be logically equivalent. Therefore, the equivalence is often verified with a logical equivalence checking tool that shows the equivalence formally through a mathematical proof [48]. In RTL-to-gate checking, the tool typically creates a one-to-one mapping for all registers in the two models and checks the equivalence of the combinatorial network between these registers. Although it is also possible to verify the new representation by simulating it, with large designs it is often impractical to repeat all test cases many times. This is a problem especially in gate-level simulation that is considerably slower compared to RTL simulation.

The logical equivalence check is typically run after RTL synthesis to verify that the synthesized netlist has equivalent behavior with the RTL description. Although the synthesis tool should produce equivalent netlist, there is always a chance that the tool has a bug and produces erroneous logic, which would be costly to fix if not spotted early. In addition to RTL-to-gate checks, it is also common to check the equivalence of two gate-level netlists after modifications, such as scan insertion, place and route, and ECO.

In this thesis work, the formal verification is run to check the equivalence of the generated RTL and the synthesized gate-level netlist. The intention is to check if the generated RTL contains structures that are challenging to verify, which would be seen in the run time or failures in the verification.

5.7 Area and timing

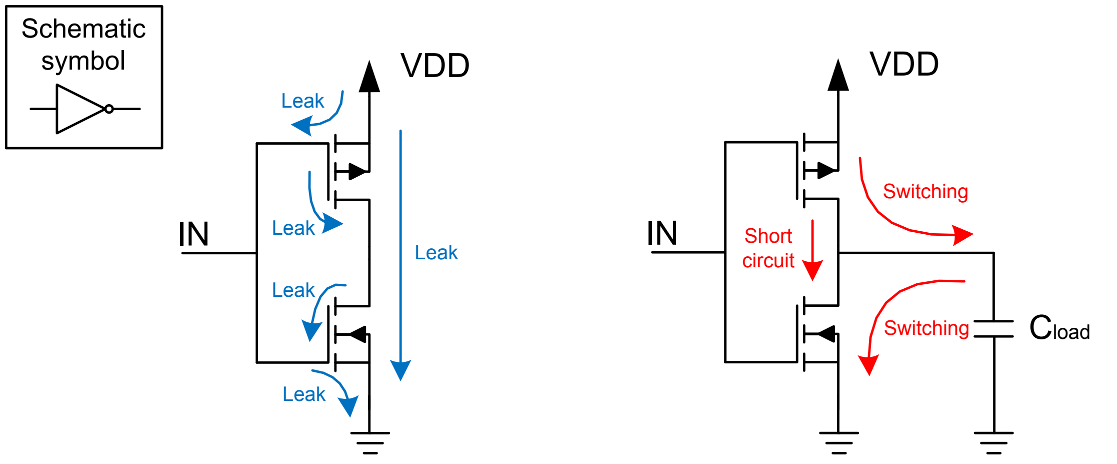
The physical area of the HLS-generated design will be compared against the hand-coded RTL design mainly to evaluate the quality and predictability of the results. Silicon area affects the unit price, and thus it should be kept at minimum. The predictability is important for being able to estimate the design area for initial floorplanning of the ASIC that is carried out at an early phase of the design flow. If the area varies largely, chip-level area estimation and partitioning becomes difficult. For this reason, the stability of the design area for several synthesis runs will also be examined.

The area and timing will also be compared to the estimates given by Catapult that are based on the internal component library of the tool. This will be the main feedback for defining the quality of the technology library characterization. If the area and timing results differ greatly from the estimates, the characterization process would have to be repeated with different settings to reach a more accurate approximation of the physical characteristics of the components.

5.8 Power efficiency

As the number of transistors in integrated circuits grows, power dissipation becomes more concentrated and power density increases. In addition, the trend of raising clock frequencies increases the power dissipation even further. As a consequence, the importance of power management has become more significant in logic design today. Whereas in the past, power consumption was mostly considered in mobile devices, nowadays all chip manufacturers have to steer towards low-power design to minimize the chance of malfunctions in the chips and the cost of cooling systems.

The power consumption in digital circuits can be generally divided into three components: leakage, switching, and short-circuit power [32]. These power components are demonstrated for a CMOS inverter in Figure 5.4. Leakage power is a static power component that is dissipated continuously as long as the circuit is connected to a voltage supply. Switching power and short-circuit power are dynamic power components. That is, they depend on the switching activity of the circuit. Switching power is the power dissipation that occurs in the transistors when they charge (or discharge) their load capacitance. Short-circuit power is internal power consumption in a CMOS cell that takes place in the short time interval during switching when both transistors are in a conductive state.



(a) Static power (b) Dynamic power

Figure 5.4 *Leakage, switching and short-circuit power components in a CMOS inverter.*

There are many ways to reduce the leakage power, for example by reducing the supply voltage or using transistors with low-leakage characteristics. However, at IP-level design, there are only few methods that can be applied to reduce leakage power.

Best way to minimize leakage is to minimize the number of transistors in the design. When the design area is optimized, the leakage power is simultaneously reduced. It is also possible to utilize power gating, which means shutting down the voltage supply to the parts of the design that are not needed at that moment. However, power gating generally requires more control logic, and it cannot be used frequently since shutting down and waking up the logic takes a relatively large amount of time. Power gating is usually done at system level, not within a single IP. Therefore, it is out of the scope of this thesis and not considered in the evaluation. Furthermore, power gating and supply voltage control is not visible to algorithm level, and thus it is rarely considered by HLS tools.

Dynamic power can be reduced by decreasing the supply voltage, minimizing the load capacitances, or lowering the amount of switching in the circuit. However, the supply voltage is typically decided at system-level, and the load capacitances cannot be affected until the RTL synthesis and physical design phases. The switching frequency can be reduced by decreasing the clock frequency – which again is a system-level decision – or by designing the logic such that unnecessary toggling of the gates is minimized. The latter can be controlled at RTL by avoiding the unnecessary switching of the states of the registers or the combinatorial logic in the input. In this way, the combinatorial network will consume only leakage power as its state is stable.

The most active net in a digital circuit is the clock tree network. Since it toggles its state twice every clock cycle, the combinatorial logic along the clock network causes significant dynamic power consumption. This combinatorial logic comprises clock buffers and the clock inputs of flip-flops that will switch even if the data input of the filp-flop remains stable. In addition, the wire capacitance of the clock network is relatively large which will increase the dynamic power further. Therefore, the clock signal should be propagated only to those flip-flops that will change their state at the next clock cycle. This act of isolating certain registers from the clock network is called clock gating.

Figure 5.5 demonstrates clock gating by showing two implementations of a circuit that contains a register that will change its output only when the enable input (en) is active. Figure 5.5(a) shows the classical implementation of the circuit that uses a multiplexer to preserve the previous state of the register. In Figure 5.5(b), the circuit is transformed such that the enable input controls the clock input of the register. When the register is disabled, its clock input is kept low, and the register retains its state. In this case, the dynamic power of the clock network after the clock gate is zero. For simplification, the clock gating element is drawn as an AND

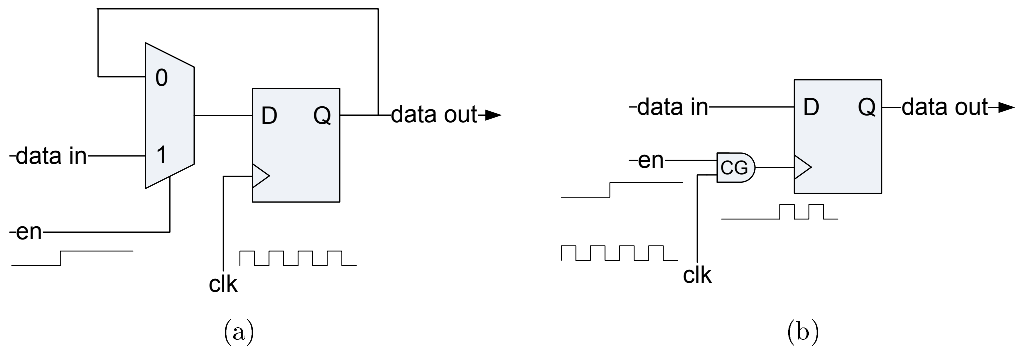


Figure 5.5 *(a) The state of the register is preserved with a multiplexer. (b) The multiplexer is transformed into a clock gating element. Both circuits have the same functionality.*

gate in the figure. In practice, however, the clock gating element often includes also a latch to avoid glitches in the output of the clock gate.

Clock gating can be added to the design at many abstraction levels. At system-level, IPs or even whole subsystems can be clock gated. Similarly at RTL, clock gates can be added to gate the clock signal for subblocks that are not needed. The designer can also do more detailed clock gating at RTL, but usually this is done automatically by the RTL synthesis tool. The tool can examine the logic and extract the conditions for updating a group of registers, and insert a clock gate to them. However, to be able to extract this information, the RTL code has to clearly imply the conditions for loading the registers with new values. Listing 5.1 shows an example of a VHDL code that will generate the clock-gated register that was shown in Figure 5.5. As the if-clause at line 6 is synthesized, the synthesis tool can deduce the condition for preserving the register state and automatically generate a clock gate.

```
1 process (clk, arst_n) is
  begin
3   if arst_n = '0' then
      data_out <= (others => '0')
5   elsif clk'event and clk = '1' then
        if (en = '1') then
7         data_out <= data_in;
        end if;
9     end if;
  end process;
```

Program 5.1 *Example of a sequential process that will imply an automatic clock gate in the RTL synthesis.*

To minimize dynamic power, the enable conditions should be as specific as possible such that the registers are updated only when necessary. For instance, the power savings in the reference decimator design are mostly achieved by clock gating data registers separately for each data channel. Hence, the clock input is toggled only for those registers that belong to the channel that is currently being processed. For example, in packager (Figure 4.6) only one of the data FIFOs is active at a time.

The simplest measure of the quality of automatic clock gating is the percentage of registers that have a clock gate. However, this is a static measure that does not take the switching activity into account. In practice, the efficiency of clock gating depends on the use case. For example, in the circuit shown in Figure 5.5, 100% of the registers are clock-gated, but if the register is always enabled, the clock-gating efficiency would be 0%.

To get a better measure of the power savings, clock gating efficiency is evaluated with the help of a switching activity file that is generated during a RTL or gate-level simulation. This activity file can either contain the full waveforms of each signal or statistical information, such as the amount of times the signal has toggled during the simulation. It is a common practice to generate the signal activity files for idle, typical, and maximum use cases. This gives a good power estimate for the whole range of use cases and reveals possible problems in the clock gating.

In this thesis, the power efficiency is mainly evaluated by examining the clock gating efficiency for three use cases: idle, typical, and maximum. These use cases are simulated with the RTL code, and the switching activity information is stored in a SAIF (Signal Activity Interchange Format) file. This file is given as an input to the RTL synthesis tool such that it can back-annotate the signal names of the gate-level netlist back to the RTL signals. Using this information, the power estimation is then run for the gate-level netlist of the design.

6. RESULTS

This chapter represents the results for the research questions introduced in the previous chapter. Before going to the results, however, there are a few comments about the experiences with the HLS design flow and a comparison of the effort required in HLS and traditional RTL design.

6.1 Design entry and effort

When starting to use Catapult, it took a couple of weeks to learn the new coding style in C++ that creates synthesizable logic. However, after getting used to the flow, creating the design was straightforward; especially the signal processing algorithms were easy to describe in the high-level code. Control logic was a bit more complicated to write, and the code often started to resemble RTL code. For example, implementing the downsampler with a proper decimation phase in the CIC filter required more lines of code than the actual filter core.

One feature that was difficult to implement was clearing data registers and counters with certain control input. The HLS block stalls when it has no data input, which also means that it does not read the control inputs (here defined as *direct inputs*). This problem was circumvented by inserting an extra sample to the HLS block when the control value changed. As this sample passes through all subblocks, it also triggers them to read the control input and clear the related registers. Although this extra sample is fake data, it causes no harmful side effects here because the data processing is disabled during the control input change.

Another, verification-related issue was encountered with the arbiter block. The arbiter used non-blocking read because it had to be able to check both data inputs every clock cycle. With blocking read, the arbiter would have stalled waiting for data if either of the data inputs had no valid data coming. However, when using non-blocking read, the outcome is different in the RTL and C++ simulations, as the input samples are interleaved in a different order. Hence, the equivalence of the two models could not be proven with the co-simulation. Nevertheless, in this case, the generated RTL could be verified with the RTL testbench of the hand-written version, which showed that the arbiter was functioning correctly.

The decimator was too complex design (in terms of logic size) to be generated at once, as it took 12 minutes to generate the RTL only for the polyphase decimator. Moreover, generating the whole IP caused the grid machine to often run out of memory. Therefore, the design was created with the bottom-up flow where the subblocks are generated in separate projects and included as hierarchical blocks in the top-level integration. This flow is convenient as it takes less than a minute to integrate the top level. If there is a need to modify one of the subblocks, only that subblock has to be regenerated, and the top level reintegrated. This saves time as the other subblocks do not require regeneration.

Implementing the decimator with Catapult took approximately 1 month, and optimizing the design another month. In comparison, the hand-written RTL design took 3–4 months. However, the comparison is not completely fair as the RTL design time also contained specification and documentation work. During the HLS design, all features were clear to the designer, and there was no need for studying the functionality of the decimator. Both designs included a learning period as the designer had no previous experience with HLS, and the hand-written version was the first RTL design task for the designer. Taking these aspects into account, the estimated **effort was 20–50% smaller in the HLS design**, depending on the optimization needs.

The effort was estimated also based on the code line count. Commented and empty lines were excluded from the calculation in both designs. The C++ code consisted of 1 040 lines, whereas the hand-written VHDL code contained 2 800 lines. Hence, the line count in the C++ code was approximately 60% smaller compared to the hand-written VHDL. The generated VHDL file, on the other hand, contained 117 000 lines, which means 40 times more lines to debug if bugs are found in the RTL simulation.

6.2 Technology library characterization

Technology libraries are characterized with a library builder tool in Catapult. It requires a Liberty file of the ASIC technology library as an input. Listing 6.1 represents the format in which each standard cell is defined in the Liberty file. This example shows a definition of an inverter. The first lines contain general information about the cell, such as area and leakage power. They are followed by pin definitions that include direction of the pin and other related information. Input pins include their capacitance, and output pins define their function, timing, and power. The timing and power characteristics are often given in a 2-dimensional table where the value depends on the input transition time (index_1) and load capacitance (index_2). The units of these quantities are typically defined in the beginning of the Liberty file. The level of detail in the Liberty files varies. For example, here the input capacitance has only one value, whereas some other library may provide separate values for rise and fall capacitance.

```
cell( INV ) {
  area : 100.0;
  cell_footprint : "inv";
  cell_leakage_power : 0.1;
  pin( A ) {
    direction : input;
    capacitance : 0.36;
  }
  pin( Out ) {
    direction : output;
    capacitance : 0;
    function : "!A";
    timing() {
      related_pin : "A";
      timing_sense : negative_unate;
      cell_rise(example_delay_table) {
        index_1 ("0.1, 0.2, 0.4, 0.8, 1.5");
        index_2 ("0.042, 0.085, 0.123, 0.160, 0.2");
        values ( \
          "0.239840, 0.348317, 0.513873, 1.01209, 1.83128", \
          "0.289023, 0.401293, 0.678347, 1.12348, 1.91349", \
          "0.379023, 0.588913, 0.793819, 1.18588, 2.01023", \
          "0.512209, 0.712387, 0.932173, 1.48137, 2.43983", \
          "0.812348, 0.943791, 1.123474, 1.83847, 2.84391");
      }
      cell_fall(example_delay_table) {
      ...
      }
      rise_transition(example_delay_table) {
      ...
      }
      fall_transition(example_delay_table) {
      ...
      }
    }
    internal_power() {
      related_pin : "A";
      ...
    }
  }
}
```

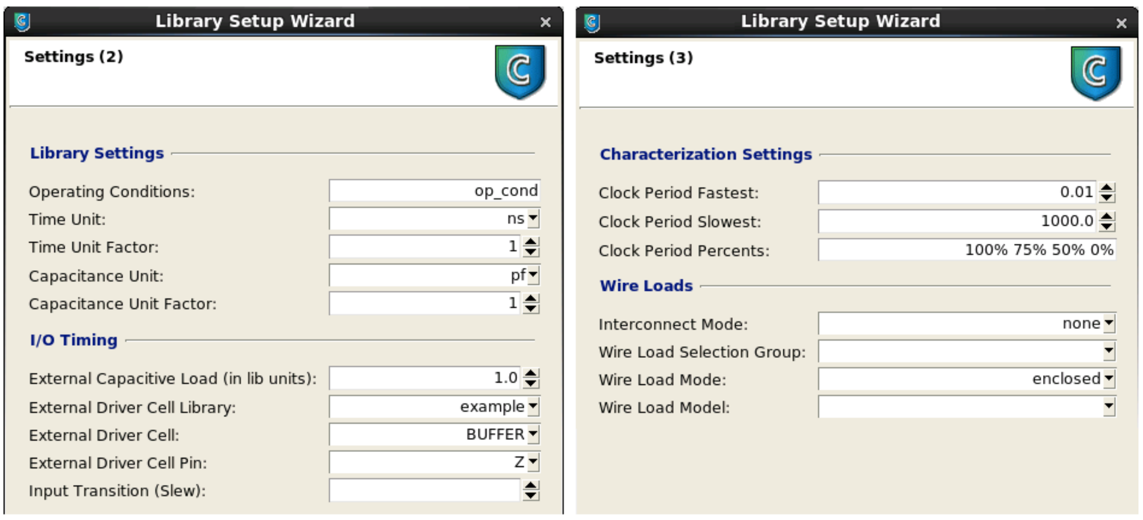
Program 6.1 *Standard cell definition of an inverter in an imaginary Liberty file.*

Based on the Liberty file, the library builder creates a collection of basic components, such as logic gates, adders, multipliers, and registers. Each of these components has several configurations. For example, adders are defined for different bit widths, and registers have several implementations that might differ by the reset behaviour or clock polarity.

The timing and area characteristics of each component are determined by synthesizing them with an external RTL synthesis tool. This process is repeated several times with different timing constraints for each configuration of the component. In the first synthesis run, the target clock period is set to an extremely small value to find the fastest implementation of the component. Similarly, the slowest and smallest implementation is found by synthesizing the component with an excessively long clock period. After finding the fastest and slowest implementations, the tool will also evaluate a couple of other measurement points between them. Based on all these measurements, the tool creates an estimation for the correlation between area and timing by interpolating between the measured values.

The shortest and longest clock periods are given by the user and might need some iteration to find suitable values. Therefore, it is a good practice to experiment with these values at first by characterizing a simple component, such as an inverter, and ensuring that with the shortest clock period even the fastest implementation cannot meet the timing, indicating that the synthesis tool has had to put effort in optimizing the delay of the implementation. Similarly, a slow component, for example a multiplier with a large bit width, should have some slack in the timing when synthesized with the longest clock period.

The library builder also needs the technology library in the format that is used by the backend synthesis tool. In addition, the tool requires a few user-defined parameters for the synthesis, as shown in Figure 6.1. The main ones are the driver cell and load capacitance that are used to build the synthesis setup shown in Figure 6.2, which also demonstrates the effect of the load capacitance on the area-delay correlation. These two parameters affect the characterization results most and might have to be adjusted later if the results are found to be inaccurate. The help section in the library builder gives some guidance for these settings, and the GUI also provides some suggestions. However, in this first experiment, the same driver cell was selected that is used in the RTL synthesis of the module. The input capacitance of this driver cell was set as the load capacitance.



Library Setup Wizard

Settings (2)

Library Settings

Operating Conditions:	op_cond
Time Unit:	ns
Time Unit Factor:	1
Capacitance Unit:	pf
Capacitance Unit Factor:	1

I/O Timing

External Capacitive Load (in lib units):	1.0
External Driver Cell Library:	example
External Driver Cell:	BUFFER
External Driver Cell Pin:	Z
Input Transition (Slew):	

Library Setup Wizard

Settings (3)

Characterization Settings

Clock Period Fastest:	0.01
Clock Period Slowest:	1000.0
Clock Period Percents:	100% 75% 50% 0%

Wire Loads

Interconnect Mode:	none
Wire Load Selection Group:	
Wire Load Mode:	enclosed
Wire Load Model:	

Figure 6.1 *User-defined parameters for the library characterization. These values are given automatically by the tool, and the user can change them if needed.*

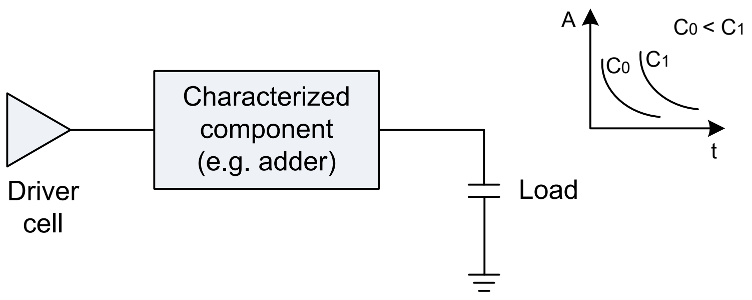


Figure 6.2 *Synthesis setup used for characterizing the library components, and the effect of the load capacitance on the area-delay correlation.*

One of the main concerns in this approach of characterizing the components is that it only considers the delay of the component itself. However, as the silicon technology nodes get more miniaturized, the delay of the interconnects between components becomes more significant compared to the gate delay [16]. The characterization process has no visibility to this delay as it has no information about the wire length between the components. Moreover, the interconnect delay depends on the placement and routing of the components which might vary significantly between designs. Hence, it is preferred to leave some safety margin in the characterization by using smaller driver cell or larger load capacitance such that the measured component delays are slightly pessimistic.

6.2.1 Flow evaluation

The flow was evaluated by characterizing two different technology libraries. With the first library, there were a few problems when parsing the Liberty file. These problems were related to the syntax used in the Liberty file that differed from the syntax that the parser expected. For this reason, a local copy of the Liberty file was created, and the problematic parts were either modified to follow the expected syntax or removed completely if they were not needed for the characterization. Finding the issue and modifying the Liberty file took 1–2 days. The parsing issue was fixed in a later release of the library builder.

After parsing the Liberty file, the synthesis flow was set up by providing the previously discussed parameters to the tool. Before the actual characterization, the user can select which components are characterized. By default, the tool characterizes all of the components that are extracted from the Liberty file. The default setting was used in this experiment, which resulted in a library of 1 800 components.

The characterization of the whole component library took 4 days to complete with one set of parameters. This is a rather long time, but as this process does not have to be repeated often, it is not seen as a significant problem. Moreover, only one person has to do the characterization as others can use the same library. However, if the characterization results are not sufficient, the iteration cycle is quite long. Therefore, it would be better to only include a subset of the component library for the iterations, and the whole library would be characterized only after finding suitable parameters. There is also a possibility to run the characterization on several computers in parallel, which would speed up the process. However, this also requires additional licenses for both library builder and the backend synthesis tool, which might be infeasible if there are only few licenses available.

The characterization of the second technology library was more straightforward as its syntax was more compliant with the parser. The characterization of the whole library took 2 days to complete even though it had an equal amount of components as the first library. The reason for the faster characterization was likely the smaller size of the standard cell library, as the file size of the Liberty file was approximately half compared to the first technology library. The difference might also be caused by the difference in the loading of the grid machines on which the library was characterized.

The characterization process can be later repeated by using a script that is created automatically during the first run. It is rather easy to adjust the necessary parameters in the script, execute it, and leave running for the duration of the characterization. The process is also simple to repeat with the GUI, as the stored library can be later opened and the settings modified via the user interface. With this approach, it is also possible to characterize individual components or a subset of the library if there is no need to update the whole library.

Overall, the flow was quite effortless, although it took a rather long time to run the whole characterization. However, as the characterization itself requires little manual work, it can be left running in the background. The flow can also be repeated afterwards with little effort if the results are inaccurate or the technology library gets updated.

6.3 Technology library abstraction

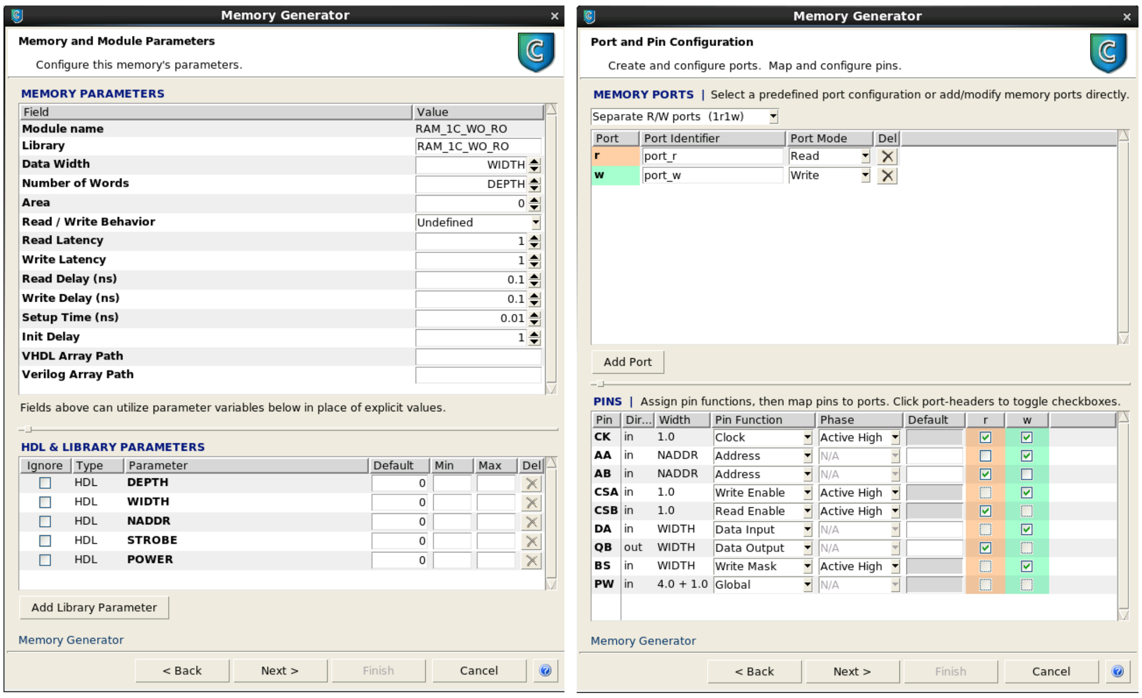
Catapult has two separate flows that can be used to abstract technology components. One is targeted only for memories, and the other is a general flow for importing customized RTL components to the tool. Since these two flows are rather different, they are discussed in separate subsections.

6.3.1 Memory libraries

The memory generator in Catapult can be used to import memories to the tool and generate a memory library. It parses the RTL description of the memory component and reads its interface. As it considers only the interface, the user can import a generic memory wrapper, which can be later switched to the technology-specific memory for RTL synthesis.

The memory generator requires also a few user-defined parameters to define the functional behavior of the memory component. These parameters include the memory type, delay and latency in terms of clock cycles for both read and write ports, and behavior of the memory in the case of simultaneous read and write operation to the same address. The number of read and write ports can be selected freely, and thus it supports the most typical memory types that are single- and dual-port memories, and memories with separate read and write ports. This process is carried out via a GUI (Figure 6.3) which also produces a TCL script that can be used to regenerate the memory library with different parameters in batch mode.

There are two ways to use the memory components in the design. The memories can implement either an external array that is used as an input or output for the design, or an internal array for storing values during data processing. When using the external arrays, the tool generates memory interfaces to the RTL top entity that



Memory and Module Parameters

Configure this memory's parameters.

MEMORY PARAMETERS

Field	Value
Module name	RAM_1C_WO_RO
Library	RAM_1C_WO_RO
Data Width	WIDTH
Number of Words	DEPTH
Area	0
Read / Write Behavior	Undefined
Read Latency	1
Write Latency	1
Read Delay (ns)	0.1
Write Delay (ns)	0.1
Setup Time (ns)	0.01
Init Delay	1
VHDL Array Path	
Verilog Array Path	

Fields above can utilize parameter variables below in place of explicit values.

HDL & LIBRARY PARAMETERS

Ignore	Type	Parameter	Default	Min	Max	Del
☐	HDL	DEPTH	0			✕
☐	HDL	WIDTH	0			✕
☐	HDL	NADDR	0			✕
☐	HDL	STROBE	0			✕
☐	HDL	POWER	0			✕

Memory Generator

Port and Pin Configuration

Create and configure ports. Map and configure pins.

MEMORY PORTS | Select a predefined port configuration or add/modify memory ports directly.

Separate R/W ports (1r1w)

Port	Port Identifier	Port Mode	Del
r	port_r	Read	✕
w	port_w	Write	✕

PINS | Assign pin functions, then map pins to ports. Click port-headers to toggle checkboxes.

Pin	Dir...	Width	Pin Function	Phase	Default	r	w
CK	in	1.0	Clock	Active High		✓	✓
AA	in	NADDR	Address	N/A		☐	✓
AB	in	NADDR	Address	N/A		✓	☐
CSA	in	1.0	Write Enable	Active High		☐	✓
CSB	in	1.0	Read Enable	Active High		✓	☐
DA	in	WIDTH	Data Input	N/A		☐	✓
QB	out	WIDTH	Data Output	N/A		✓	☐
BS	in	WIDTH	Write Mask	Active High		☐	✓
PW	in	4.0 + 1.0	Global	N/A		☐	☐

Memory Generator

Figure 6.3 *User interface of the memory generator.*

correspond to the interface of the imported memory component. The memory itself is instantiated outside of the HLS-generated block, and hence the mapping to the technology component does not differ in any way from the traditional RTL design flow.

The internal arrays can also be externalized such that the memory interface is brought to the top-level interface of the HLS block. This is the preferred way as memories often have other ports in addition to the main functional ports. These additional ports are typically used to control power saving modes or test the memory via built-in self test (BIST). It is easier to connect these signals if the memories are located at the top level of the IP than if they were within several levels of hierarchy in the HLS-generated block. With internally instantiated memories, these additional signals can also be brought to the top-level interface of the HLS block. However, not all of these signals are present in the memory wrappers since they might be connected later in the gate-level netlist. Hence, it is preferred to have all of the memories external to the design.

This flow worked well with the existing memory wrappers, and the user interface is clear and easy to follow. Defining the delay is a bit difficult especially with newer technology nodes since the wire delay is more significant, and thus the total delay depends on the distance to the memory macro. It is generally a good practice to register the memory output such that there would be no combinatorial logic between the memory and the register. In this design, there was none, but since there is no option to force the register to the memory, it is possible that the tool will generate some combinatorial logic at the output port. This could be avoided by generating a memory library where the output latency is increased by one clock cycle, and adding the register stage outside of the design between the memory and the HLS-generated IP core. However, this makes the flow more complicated, and hence the preferred way would be to add this option to the tool so that it would always add registers to the memory output.

Using the memory libraries in the design is straightforward. When defining the architectural constraints, the user can select which interfaces or arrays are mapped to memory blocks. This is demonstrated in Figure 6.4 where the output data array *data_out* is mapped to an imported memory. There is also a wide range of options for defining the way the variables are arranged in the address space of the memory. It is, for example, possible to store several variables or elements of an array to a single address. In this way, the throughput of the design can be improved since several values can be written or read within one memory access.

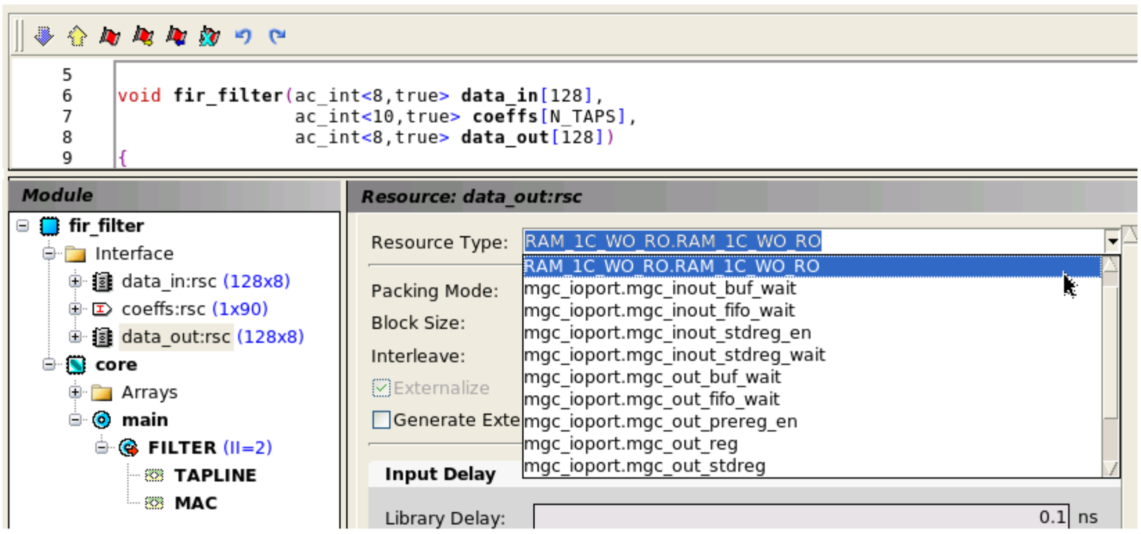

```
void fir_filter(ac_int<8,true> data_in[128],
                ac_int<10,true> coeffs[N_TAPS],
                ac_int<8,true> data_out[128])
{
```
```
Module
fir_filter
  Interface
    data_in:rsc (128x8)
    coeffs:rsc (1x90)
    data_out:rsc (128x8)
  core
    Arrays
    main
      FILTER (II=2)
        TAPLINE
        MAC
```
```
Resource: data_out:rsc
Resource Type: RAM_1C_WO_RO.RAM_1C_WO_RO
    RAM_1C_WO_RO.RAM_1C_WO_RO
    mgc_ioport.mgc_inout_buf_wait
    mgc_ioport.mgc_inout_fifo_wait
    mgc_ioport.mgc_inout_stdreg_en
    mgc_ioport.mgc_inout_stdreg_wait
    mgc_ioport.mgc_out_buf_wait
    mgc_ioport.mgc_out_fifo_wait
    mgc_ioport.mgc_out_prereg_en
    mgc_ioport.mgc_out_reg
    mgc_ioport.mgc_out_stdreg
Packing Mode:
Block Size:
Interleave:
[✓] Externalize
[ ] Generate Exte
Input Delay
Library Delay: 0.1 ns
```

Figure 6.4 *Mapping an output data interface to a memory.*

The memory usage was trialed with the packager module by implementing the data FIFOs as a memory as is shown in Figure 6.5. The used memory type had separate read and write ports such that the read and write accesses could be carried out at the same clock cycle. However, the memory that was imported to the Catapult had undefined behavior when reading and writing simultaneously to the same address. Hence, Catapult would not schedule read and write operations within the same clock cycle, even if the control logic ensured that the read and write addresses are never the same. This caused a scheduling failure, as the packager could not write samples to the memory every clock cycle, which was required by the constraints.

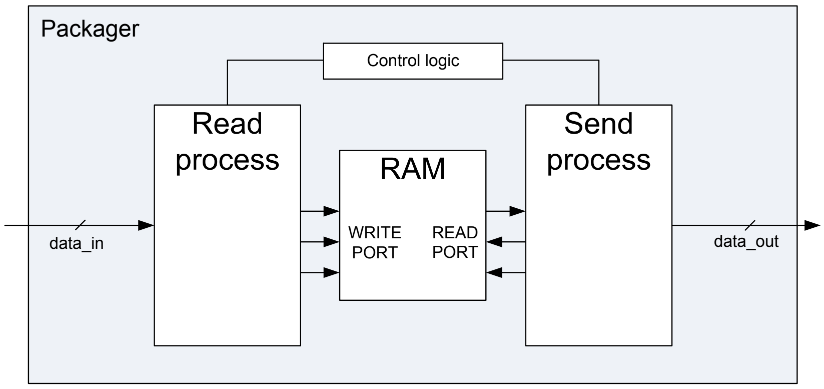


Figure 6.5 Packager with the data FIFOs implemented as a memory.

The problem with the simultaneous read and write accesses was circumvented by creating separate external memory interfaces for the read and write ports of the memory and connecting them to the same memory component at the IP top level (Figure 6.6). Using this approach, Catapult assumes that these two ports are connected to separate memory components, and thus it can schedule the read and write operations to the same clock cycle. This implementation gave the maximum throughput where the packager could take a new sample every clock cycle.

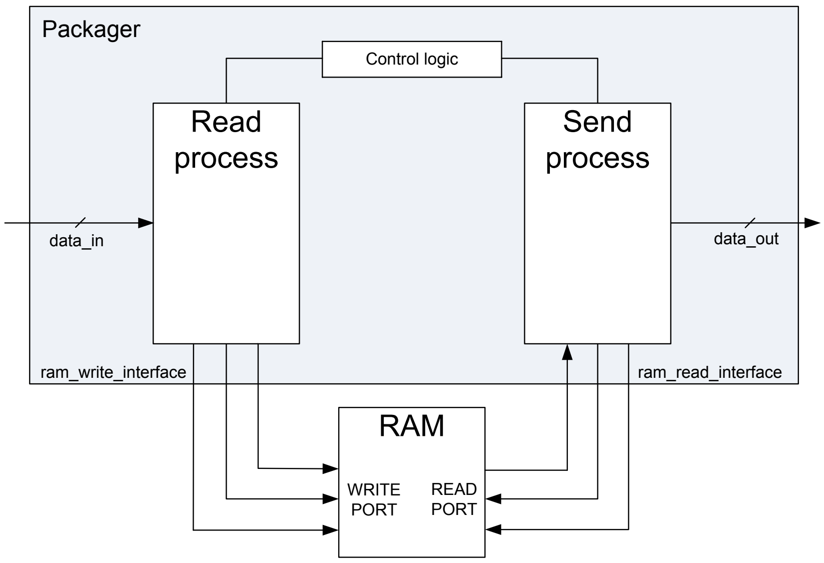


Figure 6.6 The read and write ports are implemented as separate memory interfaces that are connected to the same physical memory.

Later it was found that there is also another way to avoid this problem. Catapult has a separate command for ignoring the memory precedences. This would be a better solution to the problem, as it does not require adding the memory ports to the functions, which keeps the code cleaner.

6.3.2 Custom components

The library builder can also create libraries with customized RTL components. However, the imported components should have such functionality that they can be used in the high-level model. Therefore, the tool cannot import small technology components, such as synchronizers, because they are not seen in the behavioral model. Instead, to be able to use the synchronizer, the user should import it, for example, as a part of a CDC FIFO buffer that can be used as a data channel between two hierarchical blocks.

The imported components require a SystemC model along with the RTL description. However, there is currently little documentation about this flow, as it is not targeted to end users but requires support from the vendor. Hence, creating the SystemC model from scratch is difficult. Some common components, such as RAM-based CDC FIFOs, have templates available, but the existing RTL component would have to be modified to match the interface and functionality of the template. Otherwise, the SystemC model would have to be modified, which is not straightforward due to lacking documentation. Reverse engineering the model and modifying it would be time-consuming, and thus it is preferred to contact the tool vendor for support.

After the component has been imported, it is easy to use in the design. This was demonstrated with a customized stream interface component that maps the handshaking and data signals of the ac_channel to the corresponding signals in the customized stream interface. The component library is imported at the same time with the technology and memory libraries. While setting the architectural constraints, the customized component can be selected from the drop-down menu that selects the interface type for the ac_channel (similarly as the memory interface selection in Figure 6.4).

6.4 Design-for-testability (DFT) structures in RTL code

Catapult has currently no method for including DFT structures in the generated RTL code. However, as mentioned in the previous chapter, the DFT structures that are used in RTL are often related to clock and reset manipulation. Since this is not possible in HLS, there are few needs for specialized DFT structures. If, for instance, manual clock gating is needed, it should be done outside of the generated block, and hence the related DFT structures would also be done externally. Moreover, it is often preferred to do clock manipulations and the DFT structures separately from the functional units (which typically is the HLS block).

The BIST controls of memories can be routed to the top level of the design, for both externally and internally instantiated memories. For external memories, the routing is trivial as they are instantiated in the IP top level that is hand-written RTL code, and thus it does not differ from the traditional flow. For internal memories, the user can define *global* ports that are automatically routed to the top level of the HLS block, and from there the user can connect them to the IP top level.

If there is a need for some other DFT structures, it should be taken into account in the architecture design such that the DFT would be done outside of the HLS block. Another option is to implement the blocks with specialized DFT structures with the traditional hand-written RTL code.

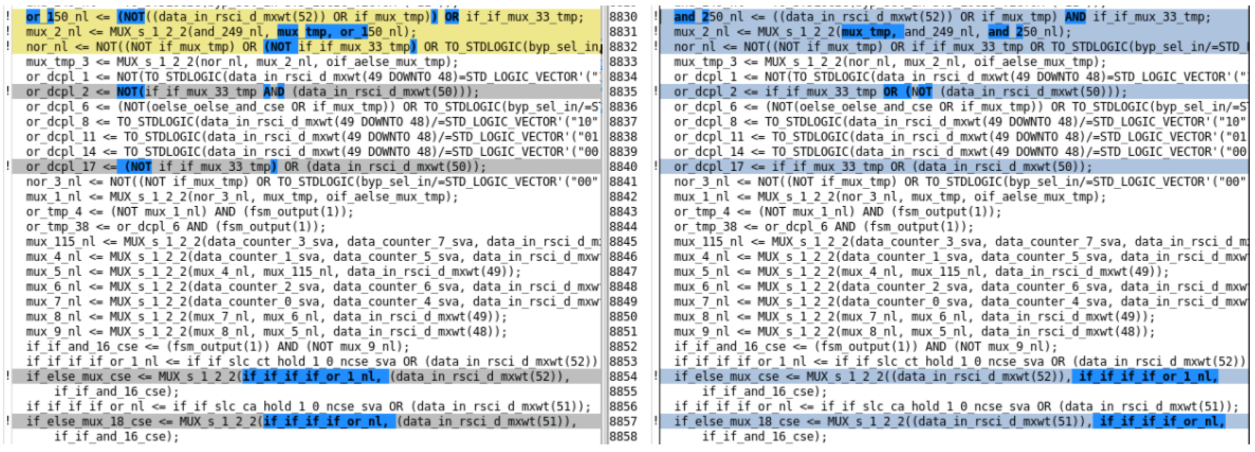
6.5 Engineering change order (ECO)

The incremental flow in Catapult is very similar to the regular HLS flow. It starts by creating a new solution with the incremental option and selecting a baseline design from the list of previous solutions. Now, the flow proceeds with same steps as the basic synthesis flow, and the user can apply the needed changes in the source codes or directives at any point. During the incremental compilation, the tool informs the percentage of variables and operations that have changed compared to the baseline. This information reveals quickly if the change was not as small as intended. After the compilation is complete, the tool also creates a report that describes the changes in more detail.

The incremental flow is intended to be used only for combinatorial changes. Hence, sequential changes, such as adding pipeline stages to fix timing issues, are not recommended. The reason for this is that changing the timing of the logic will also require changing the control logic that is tied to the amount of control steps. It is, nevertheless, possible to apply sequential changes as the tool does not prevent them in any way. However, the RTL changes will be much larger than expected. This was trialed with the CIC filter by tightening the timing constraints such that it would generate an additional register stage in the integrator chain. This resulted in 600 changed lines in the ECO RTL compared to the baseline design.

The experiments with the ECO flow consisted of doing a dozen of different combinatorial changes to the high-level code and examining their effect on the generated RTL code. Some examples of these ECOs are changing the conditions in if-clauses, rounding styles, and intermediate data widths in arithmetic chains. The amount of RTL line changes was in the range of 1–513 for these trials (the whole design containing over 100 000 lines), but most often the number was a few dozens of lines. It should be noted, however, that this is only a rough estimate of the resulting ECO changes, as the number of RTL code changes is generally not a good measure of the ECO implementation quality.

Although the ECOs changed dozens of lines in the generated RTL code, the modifications were often seen in low-level operations that have little effect on the gate-level netlist. Figure 6.7 shows an example where the polyphase decimator branching logic was changed such that the even and odd branch were swapped, and the first sample would be sent as such instead of waiting for both even and odd samples. This resulted in 10 line changes for every polyphase decimator. The changes consisted mostly of inverting control signals, replacing OR with AND, and swapping multiplexer inputs. All of these changes have a relatively small impact on the gate-level netlist.



```
! or_150_nl <= (NOT((data_in_rsci_d_mxwt(52)) OR if_mux_tmp)) OR if_if_mux_33_tmp;
! mux_2_nl <= MUX_s_1_2_2(and_249_nl, mux_tmp, or_150_nl);
! nor_nl <= NOT((NOT if_mux_tmp) OR (NOT if_if_mux_33_tmp) OR TO_STDLOGIC(byp_sel_in
  mux_tmp_3 <= MUX_s_1_2_2(nor_nl, mux_2_nl, oif_aelse_mux_tmp);
  or_dcpl_1 <= NOT(TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)=STD_LOGIC_VECTOR'("
! or_dcpl_2 <= NOT(if_if_mux_33_tmp AND (data_in_rsci_d_mxwt(50)));
  or_dcpl_6 <= (NOT(oelse_oelse_and_cse OR if_mux_tmp)) OR TO_STDLOGIC(byp_sel_in/=S
  or_dcpl_8 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("10"
  or_dcpl_11 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("01
  or_dcpl_14 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("00
! or_dcpl_17 <= (NOT if_if_mux_33_tmp) OR (data_in_rsci_d_mxwt(50));
  nor_3_nl <= NOT((NOT if_mux_tmp) OR TO_STDLOGIC(byp_sel_in/=STD_LOGIC_VECTOR'("00"
  mux_1_nl <= MUX_s_1_2_2(nor_3_nl, mux_tmp, oif_aelse_mux_tmp);
  or_tmp_4 <= (NOT mux_1_nl) AND (fsm_output(1));
  or_tmp_38 <= or_dcpl_6 AND (fsm_output(1));
  mux_115_nl <= MUX_s_1_2_2(data_counter_3_sva, data_counter_7_sva, data_in_rsci_d_m
  mux_4_nl <= MUX_s_1_2_2(data_counter_1_sva, data_counter_5_sva, data_in_rsci_d_mxw
  mux_5_nl <= MUX_s_1_2_2(mux_4_nl, mux_115_nl, data_in_rsci_d_mxwt(49));
  mux_6_nl <= MUX_s_1_2_2(data_counter_2_sva, data_counter_6_sva, data_in_rsci_d_mxw
  mux_7_nl <= MUX_s_1_2_2(data_counter_0_sva, data_counter_4_sva, data_in_rsci_d_mxw
  mux_8_nl <= MUX_s_1_2_2(mux_7_nl, mux_6_nl, data_in_rsci_d_mxwt(49));
  mux_9_nl <= MUX_s_1_2_2(mux_8_nl, mux_5_nl, data_in_rsci_d_mxwt(48));
  if_if_and_16_cse <= (fsm_output(1)) AND (NOT mux_9_nl);
  if_if_if_if_or_1_nl <= if_if_slc_ct_hold_1_0_ncse_sva OR (data_in_rsci_d_mxwt(52))
! if_else_mux_cse <= MUX_s_1_2_2(if_if_if_if_or_1_nl, (data_in_rsci_d_mxwt(52)),
      if_if_and_16_cse);
  if_if_if_if_or_nl <= if_if_slc_ca_hold_1_0_ncse_sva OR (data_in_rsci_d_mxwt(51));
! if_else_mux_18_cse <= MUX_s_1_2_2(if_if_if_if_or_nl, (data_in_rsci_d_mxwt(51)),
      if_if_and_16_cse);
```

```
! and_250_nl <= ((data_in_rsci_d_mxwt(52)) OR if_mux_tmp) AND if_if_mux_33_tmp;
! mux_2_nl <= MUX_s_1_2_2(mux_tmp, and_249_nl, and_250_nl);
! nor_nl <= NOT((NOT if_mux_tmp) OR if_if_mux_33_tmp OR TO_STDLOGIC(byp_sel_in/=STD_
  mux_tmp_3 <= MUX_s_1_2_2(nor_nl, mux_2_nl, oif_aelse_mux_tmp);
  or_dcpl_1 <= NOT(TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)=STD_LOGIC_VECTOR'("
! or_dcpl_2 <= if_if_mux_33_tmp OR (NOT (data_in_rsci_d_mxwt(50)));
  or_dcpl_6 <= (NOT(oelse_oelse_and_cse OR if_mux_tmp)) OR TO_STDLOGIC(byp_sel_in/=S
  or_dcpl_8 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("10"
  or_dcpl_11 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("01
  or_dcpl_14 <= TO_STDLOGIC(data_in_rsci_d_mxwt(49 DOWNTO 48)/=STD_LOGIC_VECTOR'("00
! or_dcpl_17 <= if_if_mux_33_tmp OR (data_in_rsci_d_mxwt(50));
  nor_3_nl <= NOT((NOT if_mux_tmp) OR TO_STDLOGIC(byp_sel_in/=STD_LOGIC_VECTOR'("00"
  mux_1_nl <= MUX_s_1_2_2(nor_3_nl, mux_tmp, oif_aelse_mux_tmp);
  or_tmp_4 <= (NOT mux_1_nl) AND (fsm_output(1));
  or_tmp_38 <= or_dcpl_6 AND (fsm_output(1));
  mux_115_nl <= MUX_s_1_2_2(data_counter_3_sva, data_counter_7_sva, data_in_rsci_d_m
  mux_4_nl <= MUX_s_1_2_2(data_counter_1_sva, data_counter_5_sva, data_in_rsci_d_mxw
  mux_5_nl <= MUX_s_1_2_2(mux_4_nl, mux_115_nl, data_in_rsci_d_mxwt(49));
  mux_6_nl <= MUX_s_1_2_2(data_counter_2_sva, data_counter_6_sva, data_in_rsci_d_mxw
  mux_7_nl <= MUX_s_1_2_2(data_counter_0_sva, data_counter_4_sva, data_in_rsci_d_mxw
  mux_8_nl <= MUX_s_1_2_2(mux_7_nl, mux_6_nl, data_in_rsci_d_mxwt(49));
  mux_9_nl <= MUX_s_1_2_2(mux_8_nl, mux_5_nl, data_in_rsci_d_mxwt(48));
  if_if_and_16_cse <= (fsm_output(1)) AND (NOT mux_9_nl);
  if_if_if_if_or_1_nl <= if_if_slc_ct_hold_1_0_ncse_sva OR (data_in_rsci_d_mxwt(52))
! if_else_mux_cse <= MUX_s_1_2_2((data_in_rsci_d_mxwt(52)), if_if_if_if_or_1_nl,
      if_if_and_16_cse);
  if_if_if_if_or_nl <= if_if_slc_ca_hold_1_0_ncse_sva OR (data_in_rsci_d_mxwt(51));
! if_else_mux_18_cse <= MUX_s_1_2_2((data_in_rsci_d_mxwt(51)), if_if_if_if_or_nl,
      if_if_and_16_cse);
```

Figure 6.7 *Incrementally synthesized RTL code compared to the original file. The ECO has affected several lines, but the changes are seen mostly in gate-level operations.*

In all trials, the RTL changes were confined within the hierarchical subblock in which the code was modified. This is convenient as it also means that the changes in the physical netlist are likely located in a limited area and do not spread all around the IP block. Hence, from ECO point of view, it is preferred to divide the design into smaller subblocks. However, this approach also creates area overhead due to additional registers and data channels between the subblocks.

The results seemed to be rather situational at times. This was noticed when changing the rounding scheme for fixed point numbers. In the original design, if the number to be rounded was exactly at half point, positive numbers were rounded towards positive infinity and negative numbers towards negative infinity. Now, if the rounding style was changed such that the numbers are always rounded towards negative infinity, the difference was seen only on 2 lines in the RTL. However, rounding always towards positive infinity changed 513 lines in the RTL code. Hence, it is difficult to estimate the extent of the ECO changes as the result can vary greatly even among similar changes.

Overall, the flow is easy to use and the results are usually reasonable, but there are some cases that cause larger changes than expected. The modifications are seen in many lines of the RTL code, but as most of them involve changes in low-level operations and signal connections, they will likely translate into relatively small changes in the gate-level netlist. However, as the number of RTL line changes does not necessarily correlate with the resulting ECO changes, a further study could also show the actual effect on the gate-level netlist and layout.

6.6 Static code analysis

The static code analysis was run for the generated RTL using a 3rd party tool. It revealed no errors or other severe problems in the code. There were 543 warnings in total, most of them being duplicates of the same warnings. Excluding these duplicates, there were only 6 unique warnings reported. These warnings are listed in Table 6.1. Same linting check was also run for the hand-written RTL code for comparison. It had 4 warnings in total, none of them being duplicates.

Table 6.1 *Lint warnings in the HLS-generated RTL code.*

Warning type	**Source**	**Amount**
Multiplexer depth exceeds limit (3)	Generated RTL	233
Unnecessary signal in sensitivity list	Common libraries	219
Output port of an entity is not used	Generated RTL	47
Null range in for-loop	Common libraries	33
X value used	Common libraries	10
While statement may be unsynthesizable	Common libraries	1
Total		543

Closer inspection – which took approximately 15 minutes per warning type – showed that all of the warnings in the generated RTL code were harmless. The most suspicious thing was the assignment of X value to the signals. For example, the X value was assigned to the output of a built-in 4-to-1 multiplexer in a case where the select input is something else than 00, 01, 10, or 11. However, as this is an exhaustive list of all the possible inputs, there is never a case where the X value would actually be assigned to the output. Moreover, the X value is often used as a don't care term, which allows the synthesis tool to optimize the logic.

Other warnings were also harmless cases, such as unused signals. For instance, the interconnect components had an output that shows the amount of samples in the FIFO buffer. Since this output was not needed by the following block, it was left open which triggered the lint warning. Another warning that had a lot of duplicates was a signal that was included in a sensitivity list of a process that did not use that particular signal. The only problem with this is that the process will be executed in simulation unnecessarily, and thus it slows down the simulation slightly. However, in practice the simulators will likely optimize the sensitivity lists during compilation. For synthesis, this does not make any difference either.

Most of the warnings were within the common RTL libraries of Catapult that are shared by many designs. Hence, it is possible to create common waivers for the warnings that are considered harmless, and thus remove the need for every designer to check the same warnings many times. This would ease the waiver creation and consequently improve the productivity of the design flow. On the other hand, it is also easy for the tool vendor to fix the problems within these common libraries, and hence it is preferable to report these issues to the vendor to help them improve the libraries for later releases.

6.7 Logical equivalence checking (LEC)

Logical equivalence checking was run after RTL synthesis to verify that the netlist matches the RTL code. The purpose of this trial was to see if the generated RTL code contains structures that are difficult to verify or cause failures.

None of the verification runs found failing points, and most of them were finished within 30 minutes. However, on one occasion, the LEC tool got stuck for 10 hours trying to verify one point, and the verification was terminated before finishing. The long runtime was caused by the CIC filter, where Catapult had scheduled the whole integrator chain within one clock cycle and allocated separate adders for each channel. As these adders were redundant, the RTL synthesis tool optimized the arithmetic chain heavily, especially since it formed the critical path. Now, the RTL and gate-level netlist were structurally so different that proving the equivalence became difficult for the tool. This problem has been identified also in previous studies [34].

The reason for the redundant adders was a coding style where the arithmetic operations were embedded within several loops and conditions. This was a remainder from the trials with the multiplexer optimizations (more details in Section 6.8) and was not supposed to be synthesized. Hence, the code was fixed, re-synthesized, and the formal verification was repeated, this time finishing successfully within 30 minutes. Nevertheless, the learning point is that complex, nested conditions and loops should be avoided, as the HLS tool gets confused and cannot produce optimal logic. Moreover, this shows that it is possible to run into problems with formal verification if the HLS tool generates redundant logic that is later optimized away at RTL synthesis, and thus HLS tools should not rely too much on the backend tool optimizations.

6.8 Area and timing

The HLS-generated and hand-written RTL codes were synthesized with a RTL synthesis tool to compare the areas and to see if the HLS block meets the timing constraints. Although Catapult provides synthesis scripts automatically, this experiment used the existing ones that have been used in previous ASIC projects. In this way, the results are expected to be more realistic.

The synthesis was repeated for over 5 different versions of the HLS block to examine the area variation. All of these versions had the same basic architecture, and only the implementation details and coding style changed. Previous studies have already shown that hardware-oriented coding style improves the quality of results significantly compared to simple algorithm description [18], and thus that kind of comparison was not considered here. Instead, the different coding style changes were done from the point of view of a hardware designer.

Table 6.2 lists the area results for 5 variations of the HLS block and also for the hand-written design that is used as a reference. The table shows both the estimations given by Catapult and the RTL synthesis results, and also the distribution between sequential and combinatorial area. The HLS results listed here had the most significant area variations within all trials. The area increase compared to hand-written RTL was between 1–9% for all implementations.

In the first trial, the area was 9% larger compared to the hand-written design. A closer look at the synthesis results revealed that the data channels between hierarchical blocks had FIFO buffers even if there was no need for buffering. The reason for this was that the FIFO depths were not explicitly defined, but Catapult determined them automatically to a value that varied between 3–11 words. The area

***Table 6.2** Area comparison of the hand-written and several versions of the HLS-generated designs. All values are scaled to the total area of the hand-written design.*

Design description	**Catapult estimation**			**Synthesis result**		
	Total	**Seq**	**Comb**	**Total**	**Seq**	**Comb**
Hand-written RTL (reference)	–	–	–	1.00	0.78	0.22
First trial with Catapult 8.3a	1.29	0.75	0.54	1.09	0.80	0.29
Optimized FIFO depths	1.25	0.71	0.54	1.01	0.74	0.27
Same as above with Catapult 10.0	1.25	0.71	0.54	1.01	0.74	0.27
Optimized multiplexer structures	0.96	0.70	0.26	1.01	0.74	0.27
Timing violations fixed (final)	0.97	0.71	0.26	1.02	0.75	0.27

decreased by 7% after fixing the issue by explicitly defining the FIFO depths to 0. The time used for this fix was 20 minutes of manual work (including both locating the issue and fixing it), 30 minutes of RTL generation, and 1 hour of RTL synthesis.

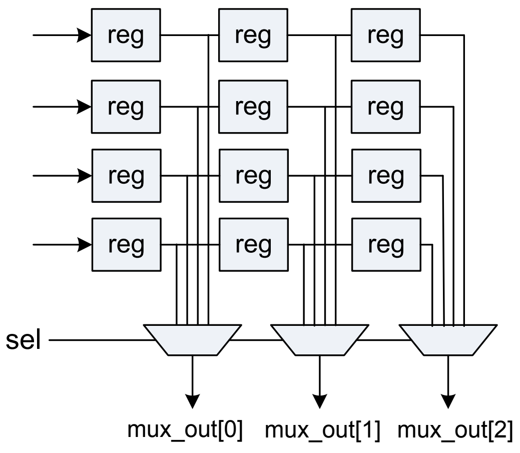
While the FIFO depths were adjusted, a new version of Catapult became available, so the same design was synthesized also with Catapult 10.0 (8.3a was used previously). The software update caused practically no difference in the results.

The first area estimations given by Catapult were 24% larger compared to the final synthesis results. Closer inspection of the resource allocation in Catapult showed that the multiplexers in the design were larger than expected. As these multiplexers contributed to a large part of the combinatorial logic, the cause for this was investigated in more detail. Moreover, large multiplexers may be challenging to route and cause routing congestion, which increases the area further or might even be impossible to implement [43].

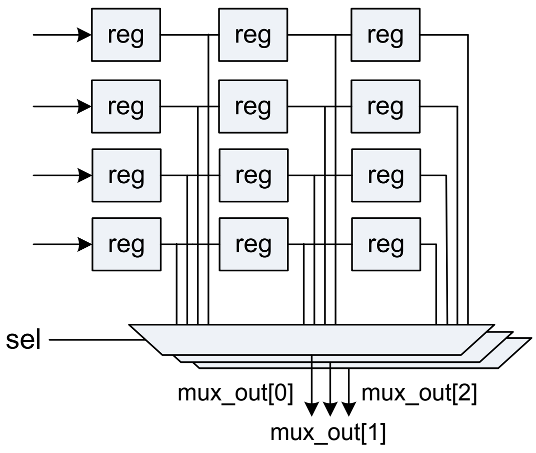
The problem with the multiplexers was related to the use of 2-dimensional arrays in the C++ code. An example of such case is shown in Program 6.2 that instantiates a 4×3 array of integers and a for-loop that reads 3 values from the array. The values are selected with the *sel* input. The expected RTL structure is shown in Figure 6.8 as well as the actual result. In the expected result, three 4-to-1 multiplexers select the values from the register array. However, the actual HLS result instantiates three 12-to-1 multiplexers instead. The reason for this is likely that the 2-dimensional array is flattened to a 1-dimensional, 12-element array during compilation, and the HLS tool does not recognize the optimization possibility.

```
int reg_array[4][3]; // Later changed to [4][4]

// Write data to the register array here
// ...

// Select the register outputs with the sel-input
for (int i = 0; i < 3; i++)
{
    mux_out[i] = reg_array[sel][i];
}
```

Program 6.2 *2-dimensional integer array and a for-loop that selects three integers from the array.*



(a) Goal



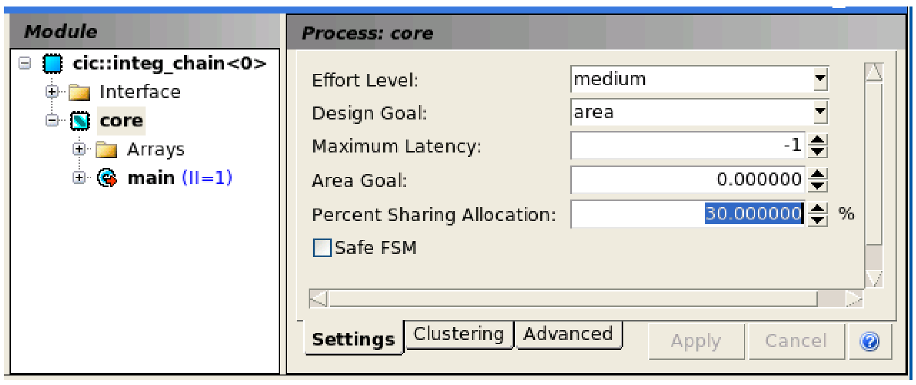
(b) Result

Figure 6.8 *(a) The expected RTL architecture of the code example consists of a 4×3 register array and three 4-to-1 multiplexers. (b) The actual result contains three 12-to-1 multiplexers instead.*

A couple of 2-dimensional arrays in the design had properly optimized multiplexers. The difference with the others was that the array size was defined with powers of 2. Following this notion, all 2-dimensional arrays were defined such that their sizes were rounded up to the closest power of 2. For instance, in Program 6.2 the array would be instantiated as a 4×4 array. Now, if the code was otherwise left as is, the HLS tool would optimize the unused registers away and the final result would have the expected structure shown in Figure 6.8(a). After modifying the code this way, the estimated area was reduced by 23%. However, the modifications also made the code harder to follow, and thus the issue should be fixed in the tool instead.

RTL synthesis of the optimized design showed no difference in the area results compared to previous versions. Seemingly, the synthesis tool had been able to detect the redundant multiplexers and optimize them away. Now the estimations by Catapult were rather close to the synthesis results, having slightly smaller area. The smaller area indicates that the technology characterization has been optimistic, and Catapult has been using smaller but slower components in scheduling. This was also seen as timing violations in the netlist.

The timing violations were in the integrator and comb chains of CIC filter, as expected. Therefore, the design constraints were modified in Catapult. Since the issue was seen only in the CIC filter, the constraints were modified locally for these arithmetic chains. While setting directives to hierarchical blocks, there is an option called *sharing allocation* (see Figure 6.9) that defines the percentage of the clock cycle that is reserved for control logic and routing. Increasing this percentage requires the tool to add pipeline stages in the chain.



Effort Level: medium
Design Goal: area
Maximum Latency: -1
Area Goal: 0.000000
Percent Sharing Allocation: 30.000000 %
☐ Safe FSM

Figure 6.9 *Sharing allocation can be adjusted separately for each hierarchical block.*

If the timing issue was seen all over the design, it would have been better to adjust the *clock uncertainty* for the whole IP in the HLS tool, or re-characterize the technology library. The clock uncertainty option was also tried in solving the timing issue. It had to be set to a 60% larger value than what is used in the RTL synthesis scripts to make the scheduler create an additional pipeline stage. This also indicates that the technology characterization has been too optimistic, which would require repeating the characterization with different parameters. However, this was not seen as necessary for this trial since the locally increased sharing allocation solved the issue.

While fixing the timing issue, it was noticed that the scheduler does not divide the operations evenly into the control steps (i.e. clock cycles). For example, if there was a 5-adder chain, but only 4 adders would fit into one clock cycle, the scheduler would put the first 4 adders to the first clock cycle and the remaining adder to the second cycle. It relies on the RTL synthesis tool to optimize the pipelining through register retiming. Therefore, if the synthesis flow does not use register retiming, the designer would have to manually adjust the operations in the control steps. Fortunately, Catapult lets the user to adjust these in the schedule diagram with a drag-and-drop interface as shown in Figure 6.10, or with TCL commands.

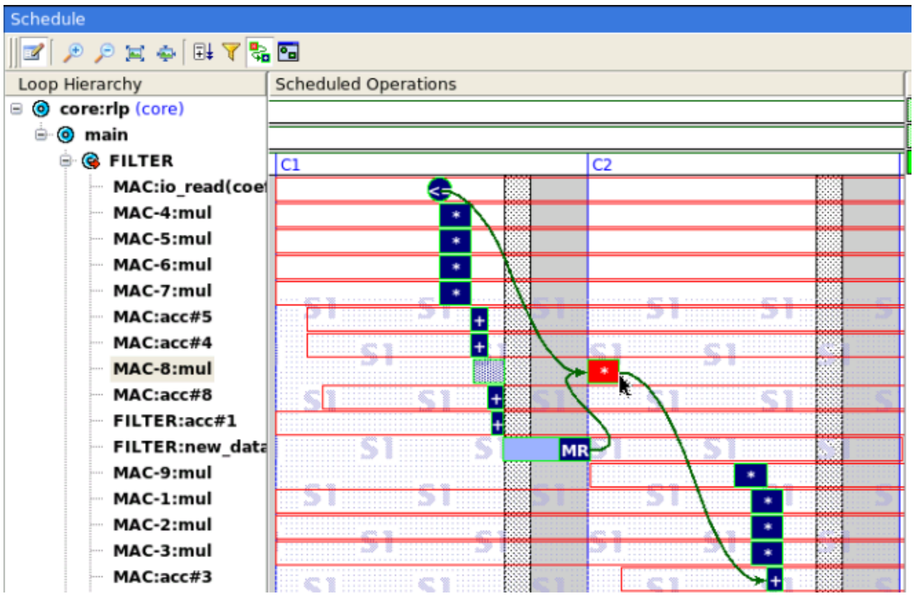


Figure 6.10 *The scheduling diagram lets the user move operations from clock cycle to another within the limits of data dependencies. The red boxes around the operations show the sliding window in which the operations can be moved.*

With the timing violations fixed, the total area increased by 1%, most of the increase being in the sequential area due to the additional pipeline stage. Area breakdown of this final design is shown in Figure 6.11. It shows the sequential and combinatorial area of each subblock for both the hand-written and HLS-generated designs. The figure also shows the area estimates given by Catapult. The areas are rather close to each other, which speaks for the predictability of the results. The estimation error for the total area was -4%. Generally, it seems that the HLS blocks have slightly smaller sequential area but larger combinatorial area compared to the hand-written design. This is expected since the HLS tool does more optimized pipelining which requires less registers, but faster and larger logic cells.

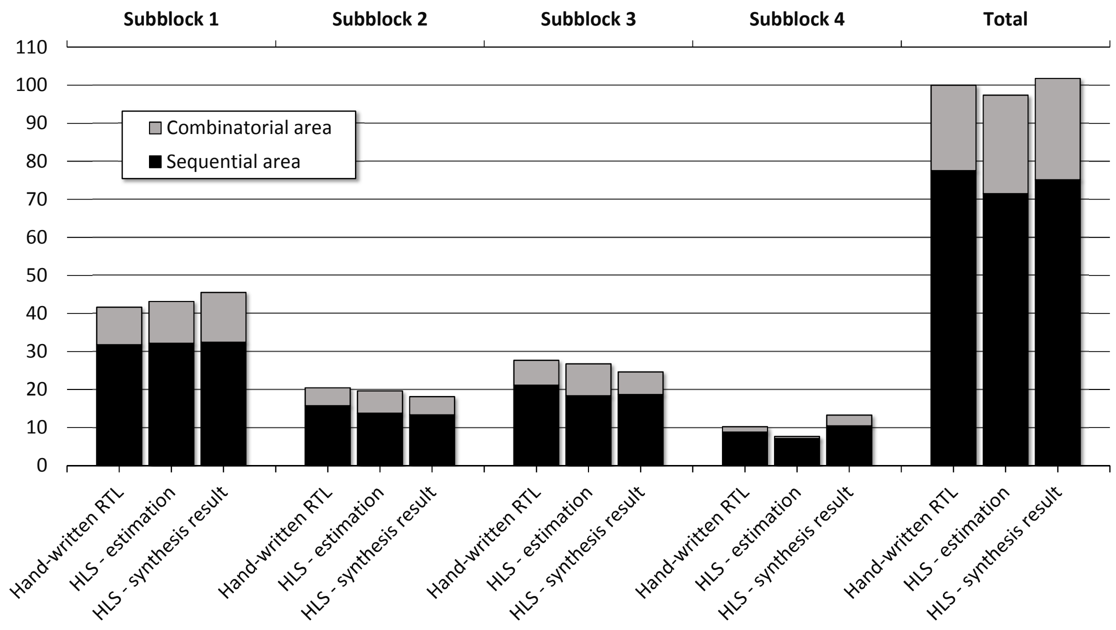


Figure 6.11 *Area breakdown of the hand-written and HLS-generated design. Both the estimation by Catapult and the RTL synthesis result are shown for the HLS block.*

6.9 Power efficiency

The power efficiency of the HLS-generated block was evaluated mainly by the clock gating efficiency and compared to the manually coded version. The first measure of the clock gating quality was given by the RTL synthesis tool as the percentage of registers that were clock gated. The HLS block had 99.7% of its registers clock gated which is slightly better than the hand-coded RTL block for which the same measure was 97.4%. The result is promising and indicates that at least in the idle case, only small part of the registers are toggling. As previously mentioned, however, this is only a partial thruth since this simple measure does not take the use case into account. Hence, the dynamic clock gating efficiency was measured based on 3 simulations.

The power estimations were run for the RTL code generated with the base version of Catapult. However, Catapult has also a Low Power version that focuses especially on the clock gating and generates conditions to the RTL code that enhance the clock gating efficiency. As these features were not used, the results do not fully represent the power optimization capabilities of the tool, and hence they are not shown here.

7. RECOMMENDATIONS

This chapter offers a few recommendation both for the designers to incorporate HLS in their design flow, and for the tool developers to further improve the HLS tools for ASIC design.

7.1 Designers

The learning curve should be taken into account when starting to work with HLS, as it takes time to learn the hardware-oriented C++ coding style and get a basic understanding of the synthesis flow. Creating the first design with HLS might take twice as long as it normally would, and therefore, the designers that are starting to use HLS should reserve time for this learning period.

There should be dedicated persons in the company that can help with the tool-related issues. These key users should also use the HLS tool actively in design projects and be in close contact with the tool vendor. Asking help from more experienced users is extremely important with HLS, as it was noticed quickly that the tool has many ways to approach the design problems, and all of them are not so apparent. There were plenty of times when the design goals seemed difficult to reach, but after expressing the issue to the application engineers at Mentor Graphics, they came up with a simple solution.

Learning to interpret the resource allocation and scheduling information is essential to reach good quality of results. Even though the tool creates the RTL architecture automatically, the designer should have a basic understanding of the resulting hardware and check the resource allocation and scheduling for large deviations from the expected results, which could reveal problems in the code or directives. Moreover, the HLS tool might have bugs, and reporting them helps the vendor to further develop the tool.

Best results are generally achieved by keeping the code simple. Trying to optimize the design with complicated code structures usually ended up in worse results. Moreover, several nested loops and conditions appeared to confuse the tool and make it more difficult for it to optimize the logic. This also resulted in redundant logic, which caused problems in logical equivalence checking as the heavily optimized gate-level netlist differed greatly from the RTL description.

The HLS should be taken into account already in the system architecture design. As was seen in this design (e.g. with the arbiter), some RTL architectures might cause problems either in the HLS design or verification flow. Hence, this limits the possibility to use HLS in every module, and it should be considered when deciding what parts of the system are done with HLS. Generally, the most suitable design types for HLS seemed to be simple signal processing pipelines with little dynamic control. Moreover, specialized DFT structures and technology-dependent parts would be better to leave outside of the HLS-generated modules, if possible.

7.2 Tool developers

HLS tool developers – especially those targeting ASICs – should generally consider the backend flow more. In addition to area, timing and power, the tools should also take other aspects of the physical implementation into account, such as routing congestion. This could be done possibly with the help of feedback from the backend tool as is proposed in [43], or by avoiding hardware architectures that are known to cause problems. The tools should also provide some means for implementing ECOs or continue improving the existing ECO flows such that the resulting changes would be minimized.

The signal and entity names in the generated RTL code should be informative and concise, and related to the variable and function names used in the high-level code. This helps debugging and also keeps reports, such as synthesis and linting logs, cleaner and easier to read. Moreover, tracing the RTL code structures back to the high-level code is faster if the signals have informative names.

The HLS tools should not rely too much on the optimization of the backend synthesis tool, as generating a lot of redundant logic can cause problems later in the flow. This was noticed in logical equivalence checking where the tool got stuck for several hours proving the equivalence of largely different RTL and netlist.

In this study, the IP was structured such that all of the interface components were outside of the HLS-generated block. However, the downside of this approach that the interface components are not included in the high-level verification, and thus it requires additional verification for the IP top level to check that all of the interfaces function properly. Therefore, it would be preferred if the tool could also generate the streaming and configuration interfaces with some common protocols. The register bank that contains the configuration values could also be generated, as that is a common structure in many IPs. If this was possible, the whole IP could be generated with the HLS tool without the need to separately generate and manually integrate all of the components.

More effort is needed in the standardization of the HLS design entry. Currently, each tool uses different libraries and pragmas, which requires extensive modification to the code if the code is synthesized with another tool. This effort would be minimized if there was a standardized way to create HLS designs. Fortunately, some standardization work has been already started by Accellera to define a synthesizable subset of C/C++/SystemC that HLS tools should support [35].

8. CONCLUSIONS

This thesis work studied the suitability of HLS for the implementation flow of ASICs. The HLS flow was found to be mostly compliant with the existing RTL design flow, but a few problem areas were also identified. ECOs may be difficult to implement, as the amount of RTL changes can vary significantly. In addition, importing customized components and adding DFT structures are challenging, which should be taken into account in the system architecture design.

The second aim of this study was to assess the quality of results and compare design effort to the hand-written RTL design. The physical area of the HLS design was practically equal to the reference design, the area increase being only 2%. Both designs met the given timing constraints. The design effort in HLS was estimated 20–50% smaller, depending on the optimization needs, and the code contained 60% fewer lines of code. In itself, this is not a significant improvement to productivity, considering that the coding is often a relatively small part of the whole design work that includes also other tasks, such as specification and documentation. The productivity gain is more likely seen later as improved reusability, since the code is completely technology-independent and faster to modify and update. However, as the code is slightly different for each tool, designers have to keep using the same tool to benefit from this. The other – and probably more significant – opportunity for productivity improvement is in the verification flow. The faster test case development and increased simulation speed have a chance of boosting verification considerably. Hence, the future of HLS will depend largely on the success of the high-level verification flow.

The current generation of HLS tools seems to be capable of generating RTL code with less effort and reasonable quality of results. Hence, the HLS could be already tried in real applications. However, the first trials should be carried out on a small scale by implementing only a few simple blocks. It is, after all, possible that there are other complications in the flow that were not encountered in this thesis due to limited scope.

The decimator was still a relatively simple design, and hence it would be interesting to try creating more complex design types with HLS. These could be implemented with SystemC instead of C++ to see if it allows easier design entry for more complicated logic. Future study could also evaluate the backend-suitability of other HLS tools and provide comparison. Moreover, as this thesis considered only the first steps of the backend flow, it would be interesting to evaluate also the rest of the physical implementation flow.

BIBLIOGRAPHY

[1] Accellera, IP-XACT, [Online]. Available (accessed on 2017-02-08): http://www.accellera.org/downloads/standards/ip-xact

[2] A. N. Akansu and R. A. Haddad, Multiresolution Signal Decomposition - Transforms, Subbands, and Wavelets. Academic Press, Inc., 1992, 376 p.

[3] ARM Developer Day, 3rd, Cypress PSoC, [Online] Available (accessed on 2016-09-18): https://armdeveloperday3rd.wordpress.com/cypress-psoc/.

[4] A. D. Belegundu and T. R. Chandrupatla, Optimization Concepts and Applications in Engineering, 2nd Ed., 2011, 480 p.

[5] Cadence, Stratus High-Level Synthesis, [Online]. Available (accessed on 2017-02-08): https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[6] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. Wiley-IEEE Press, 2006, 694 p.

[7] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, An Introduction to High-Level Synthesis, IEEE Design Test of Computers, vol. 26, no. 4, 2009, pp. 8–17. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/5209958/

[8] Design & Reuse, [Online] Available (accessed on 2017-01-24): https://www.design-reuse.com/.

[9] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, Embedded System Design: Modeling, Synthesis and Verification, 1st ed. Springer, 2009, 352 p.

[10] Greenbot, Qualcomm reveals 64-bit Snapdragon 810 and 808, [Online] Available (accessed on 2017-01-22): http://www.greenbot.com/article/2140500/qualcomm-reveals-64-bit-snapdragon-810-its-fastest-ever-mobile-chip.html.

[11] I. Grout, Digital Systems Design with FPGAs and CPLDs, 1st ed. Newnes, 2008, 784p.

[12] E. Hogenauer, An Economical Class of Digital Filters for Decimation and Interpolation, IEEE Transactions on Acoustics, Speech, and Signal Processing,

vol. 29, no. 2, 1981, pp. 155–162. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/1163535/

[13] ITRS, International Technology Roadmap for Semiconductors, edition 2011, [Online] Available (accessed on 2017-01-23): http://www.itrs2.net.

[14] L. Juhola, Improving IP Block Design Flow Practices, Tampere University of Technology, Department of Pervasive Computing, Master's thesis, 2016, 69 p.

[15] J. Järviluoma, Rapid Prorotyping from Algorithm to FPGA Prototype, University of Oulu, Department of Electrical Engineering. Master's thesis, 2015, 59 p.

[16] P. Kapur, J. P. McVittie, and K. C. Saraswat, Technology and Reliability Constrained Future Copper Interconnects. I. Resistance Modeling, IEEE Transactions on Electron Devices, vol. 49, no. 4, 2002, pp. 590–597.

[17] K. Karras, M. Blott, and K. A. Vissers, High-Level Synthesis Case Study: Implementation of a Memcached Server, 1st International Workshop on FPGAs for Software Programmers, 2014, pp. 77–82. Available (accessed on 2017-02-08): http://arxiv.org/abs/1408.5387

[18] I. Kivimäki, High-Level Synthesis Design Flow in FPGA Design, University of Oulu, Department of Electrical Engineering. Master's thesis, 2016, 60 p.

[19] I. Kuon and J. Rose, Quantifying and Exploring the Gap Between FPGAs and ASICs. Springer, 2009, 180 p.

[20] L. Lavagno, A. Kondratyev, Y. Watanabe, Q. Zhu, M. Fujii, M. Tatesawa, and N. Nakayama, Incremental High-Level Synthesis, 15th Asia and South Pacific Design Automation Conference (ASP-DAC), 2010, pp. 701–706.

[21] G. Martin and G. Smith, High-Level Synthesis: Past, Present, and Future, IEEE Design Test of Computers, vol. 26, no. 4, 2009, pp. 18–25. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/5209959/

[22] Mathworks, HDL Coder, [Online]. Available (accessed on 2017-02-08): https://www.mathworks.com/products/hdl-coder.html

[23] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, An Overview of Today's High-Level Synthesis Tools, Design Automation for Embedded Systems, vol. 16, no. 3, 2012, pp. 31–51. Available (accessed on 2017-02-08): http://dx.doi.org/10.1007/s10617-012-9096-8

[24] Mentor Graphics, Catapult High-Level Synthesis, [Online]. Available (accessed on 2017-02-08): https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[25] L. Milic, T. Saramäki, and R. Bregovic, Multirate Filters: An Overview, IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), 2006, pp. 912–915. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/4145542/

[26] S. P. Mohanty, Nanoelectronic Mixed-Signal System Design. McGraw-Hill Education, 2015, 832 p.

[27] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, A Survey and Evaluation of FPGA High-Level Synthesis Tools, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, 2016, pp. 1591–1604. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7368920/

[28] NEC, CyberWorkBench, [Online]. Available (accessed on 2017-02-08): http://www.nec.com/en/global/prod/cwb/index.html

[29] U. of Toronto, LegUp High-level synthesis, [Online]. Available (accessed on 2017-02-08): http://legup.eecg.utoronto.ca/

[30] P. Ollikainen, SoC Subsystem Design Using SystemC based High-Level Synthesis, University of Oulu, Department of Electrical Engineering. Master's thesis, 2016, 48 p.

[31] OpenCores, [Online] Available (accessed on 2017-01-24): https://www.opencores.org/.

[32] P. R. Panda, B. V. N. Silpa, A. Shrivastava, and K. Gummidipudi, Power-efficient System Design, 1st ed. Springer, 2010, 253 p.

[33] Real Intent, Ascent Lint, [Online]. Available (accessed on 2017-02-08): http://www.realintent.com/real-intent-products/ascent-lint/

[34] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra, Lessons and Experiences with High-Level Synthesis, IEEE Design Test of Computers, vol. 26, no. 4, 2009, pp. 34–45. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/5209961/

[35] Semiconductor engineering, High Level Synthesis, [Online]. Available (accessed on 2017-02-08): http://semiengineering.com/kc/knowledge_center/High-Level-Synthesis/105

[36] D. Sinha and S. Kumar, FIR Filter Compensator for CIC Filter Suitable for Software Defined Radio, World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave), 2016, pp. 1–7. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7583915/

[37] G. Strawn and C. Strawn, Moore's Law at Fifty, IT Professional, vol. 17, no. 6, 2015, pp. 69–72. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7332204/

[38] G. Stringham, Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development, 1st ed. Newnes, 2009, 376 p.

[39] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis, 21st Asia and South Pacific Design Automation Conference (ASP-DAC), 2016, pp. 218–225. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7428014/

[40] Synopsys, SpyGlass Lint, [Online]. Available (accessed on 2017-02-08): https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html

[41] Synopsys, Synphony C Compiler, [Online]. Available (accessed on 2017-02-08): https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html

[42] A. Takach, High-Level Synthesis: Status, Trends, and Future Directions, IEEE Design & Test, vol. 33, no. 3, 2016, pp. 116–124. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7445863/

[43] M. Tatsuoka, R. Watanabe, T. Otsuka, T. Hasegawa, Q. Zhu, R. Okamura, X. Li, and T. Takabatake, Physically Aware High Level Synthesis Design Flow, 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015, pp. 1–6. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7167348/

[44] Tech Design Forum, Lint for Hardware Design, [Online]. Available (accessed on 2017-02-08): http://www.techdesignforums.com/practice/guides/lint-rtl/

[45] E. Torppa, High-Level Synthesis in IP based SoC Development, University of Oulu, Department of Electrical Engineering. Master's thesis, 2015, 69 p.

[46] H. Tulla, Exploring High-Level Synthesis for ASIC Module Design, Tampere University of Technology, Department of Pervasive Computing, Master's thesis, 2017, Manuscript in preparation.

[47] F. R. Wagner, W. O. Cesário, L. Carro, and A. A. Jerraya, Strategies for the Integration of Hardware and Software IP Components in Embedded Systems-on-Chip, Integration, the VLSI Journal, vol. 37, no. 4, 2004, pp. 223–252. Available (accessed on 2017-02-08): http://www.sciencedirect.com/science/article/pii/S0167926003001093

[48] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, Eds., Electronic Design Automation. Morgan Kaufmann, 2009, 972 p. Available (accessed on 2017-02-08): http://www.sciencedirect.com/science/article/pii/B9780123743640500023

[49] Wikipedia, NAND gate, [Online]. Available (accessed on 2017-02-08): https://en.wikipedia.org/wiki/NAND_gate

[50] W. H. Wolf, Hardware-Software Co-Design of Embedded Systems, Proceedings of the IEEE, vol. 82, no. 7, 1994, pp. 967–989. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/293155

[51] Wonderful Engineering, What Is An Integrated Circuit? [Online] Available (accessed on 2016-09-18): http://wonderfulengineering.com/what-is-an-integrated-circuit/.

[52] Xilinx, Vivado High-Level Synthesis, [Online]. Available (accessed on 2017-02-08): https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[53] Q. Zhu and M. Tatsuoka, High Quality IP Design using High-Level Synthesis Design Flow, 21st Asia and South Pacific Design Automation Conference, 2016, pp. 212–217. Available (accessed on 2017-02-08): http://ieeexplore.ieee.org/document/7428013/

[54] M. D. Zwagerman, High Level Synthesis, a Use Case Comparison with Hardware Description Language, Grand Valley State University, Master's thesis, 36 p.