Martin Webhofer, BSc

Evaluation of High-Level Synthesis for High-Speed Data Processing Applications

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Electrical Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Institute for Technical Informatics

Advisor
Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. (FH) Mario Ploner
Durst Phototechnik Digital Technology GmbH

Graz, September 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Kurzfassung

Elektronische Systeme sind heutzutage überall vorzufinden. Der stetige Fortschritt eröffnet weitere Möglichkeiten für den Einsatz von Elektronik. Das führt auch dazu, dass die Systeme komplexer werden und dadurch auch der jeweilige Rechenaufwand steigt. Für manche Systeme reicht die Rechenkapazität der Prozessoren nicht aus, um die Tasks echtzeitgetreu auszuführen. So hilft meist nur mehr die Auslagerung dieser Berechnungen auf speziell erstellte Koprozessoren.

Die Firma *Durst Phototechnik Digital Technology GmbH* entwickelt industrielle Hochleistungsdrucker, wobei sie mit dem Problem des stetig steigenden Rechenaufwands für die Vorverarbeitung der zu druckenden Bilder konfrontiert ist. Derzeit wird diese Vorverarbeitung auf leistungsfähige Workstations ausgeführt. Um die Verarbeitungszeit zu verringern, wird überlegt die entsprechenden Algorithmen auf Hardwaresystemen wie FPGAs zu implementieren. Für eine einfache und zeitsparende Umsetzung dieses Vorhabens wird die Verwendung der High-Level Synthese in Betracht gezogen.

High-Level Synthese steht für die automatische Erzeugung eines Modells auf Register-Transfer Ebene. Ausgangspunkt ist dabei eine entsprechende Implementierung mit einer Hochsprache wie zum Beispiel C/C++ oder SystemC. Diese Masterarbeit gibt einen Einblick in diese Designmethode. Zudem wird der Nutzen der High-Level Synthese bezüglich Zeitersparnis während des Designprozesses sowie der Qualität der resultierenden Hardwareimplementierung hinsichtlich Design Metriken im Vergleich zur herkömmlichen Designmethode mit Hardwarebeschreibungssprachen präsentiert. Neben der Recherche von bereits ähnlichen veröffentlichten Arbeiten umfasst diese Arbeit den Vergleich der zwei Methoden für ein Datenflussmodul, das in den Drucksystemen der genannten Firma verwendet wird, und für ein Verschlüsselungsmodul, um auch die Verwendbarkeit der High-Level Synthese für die Umwandlung von Algorithmen auf Register-Transfer Ebene zu untersuchen.

Das Ergebnis der Arbeit bestätigt, dass die Verwendung der High-Level Synthese für die Erstellung von Hardwaremodulen von Nutzen sein kann. Zwar erhöhen sich minimal die Anzahl der allozierten Ressourcen und der Leistungsbedarf, jedoch kann der Zeitaufwand des Designprozesses für Hardwaremodule, die komplexe Algorithmen ausführen, deutlich verringert werden. Für einfache Datenflussmodule ist hingegen die Verwendung von Hardwarebeschreibungssprachen empfehlenswert. Denn der Einsatz der High-Level Synthese verringert kaum den Aufwand des Designprozesses, aber erhöht trotzdem die allozierte Ressourcenanzahl um ungefähr ein Drittel.

Abstract

Nowadays, electronic systems are everywhere. The steady progress of electronic devices opens up further possibilities for their usage. This leads to a higher complexity and increasing computational effort for these systems. In some cases, the computing power of processors is insufficient to perform all tasks in real-time. Therefore, an alternative is the outsourcing of the CPU-intensive executions to specially designed co-processors.

The company *Durst Phototechnik Digital Technology GmbH* develops industrial high-performance printers. They are confronted with the same problem of the increasing computational effort for the preprocessing of images for their printers. Currently, this preprocessing is performed on high-performance workstations. For the reduction of processing time, they are willing to transfer the algorithms to hardware systems like FPGAs but this transfer should be done as simple and as time-saving as possible. Therefore, the use of high-level synthesis is considered.

High-level synthesis means an automatic generation of a model on register-transfer level. The starting point is a corresponding implementation with high level languages like C/C++ or SystemC. An insight into this design method is provided by this thesis. It presents the benefits of high-level synthesis in terms of time savings during the design process and the quality of the resulting register-transfer level model in terms of design metrics compared to the traditional design method using hardware description languages. Additionally the research of similar papers, this thesis includes the comparison of the mentioned design methods for a data flow module used in printing systems of the mentioned company and an encryption module to present the applicability of high-level synthesis for the conversion of algorithms to register-transfer level.

The result of this thesis confirms that the use of high-level synthesis can be useful for the creation of hardware modules. Although the number of allocated resources and the power consumption is slightly increasing, the required time for the design process of hardware modules including complex algorithms can be significantly reduced. For simple data flow modules, the use of hardware description languages is recommended. The use of high-level synthesis hardly reduces the time exposure of the design process, but increases the allocated resources by about one third.

Danksagung

Diese Diplomarbeit wurde im Jahr 2018 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Ganz besonders gilt dieser Dank Herrn Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger, der mich betreut hat. Nicht nur durch sein kritisches Hinterfragen gab er mir immer wieder wertvolle Hinweise. Auch seine kontinuierliche Motivation haben einen großen Teil zur Vollendung dieser Arbeit beigetragen. Vielen Dank für die Geduld und Mühen.

Daneben gilt mein Dank Herrn Dipl.-Ing. (FH) Mario Ploner und Herrn Dipl.-Ing. Christian Halbfurter, die als Ansprechpartner seitens der Firma *Durst Phototechnik Digital Technology GmbH* fungierten. Ihre Überlegungen und Hinweise waren für den Erfolg dieser Masterarbeit maßgebend. Deshalb möchte ich ihnen ebenfalls meinen Dank aussprechen.

Auch Herrn Dipl.-Ing. Wolfgang Knotz gilt ein Dank, da er es überhaupt ermöglicht hatte, diese Masterarbeit in Kooperation mit der Firma *Durst Phototechnik Digital Technology GmbH* durchzuführen.

Nicht zuletzt gebührt meiner Mutter, meiner Familie und meiner Freundin Isabel Dank, ohne die dieses ganze Unternehmen schon im Vorhinein niemals zustande gekommen wäre.

Graz, September 2018 Martin Webhofer, BSc

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The increasing capabilities of silicon technology in integrated circuits and the more complex designs lead us to use new methodologies and tools for the generation of such systems. Acceleration of the design process is also one central point in the generation of embedded systems. In early days assembler was used to program different processors, but over time the usage of high level programming languages were introduced, because the increasing packing density of transistors in integrated circuits was determining this necessity. In embedded programming the language C/C++ has been established over the years, which allows to create software applications with relatively low time-to-market.

Sometimes, processors usually haven't the necessary amount of computational power. In such cases, Hardware-Software-Codesign is an essential design task. Engineers have to partition the whole system into hardware and software parts. Tasks, with a huge computational effort, have to be designed in hardware, whereas control tasks are often implemented with software on microcontrollers. As we consider the hardware parts, which are mostly implemented on a FPGA or an ASIC, we have to be aware of their design flow. Conventional designs are used on Register-Transfer Level (RTL) to implement the subsystem, but in some cases the complexity of sub-systems is very high and the designing on RTL will increase the time-to-market significantly. Since some decades, engineers are researching for new methodologies to increase the abstraction level of the design flow for hardware implementations like FPGAs or ASICs. Thus, the High-Level Synthesis (HLS) was developed to decrease the time-to-market for embedded systems and to simplify the design process. HLS uses high-level programming languages for designing the hardware on integrated circuits (ICs). The most common used languages are C/C++ or SystemC. Nowadays, it is also possible to create hardware designs on FPGAs with tools like Matlab or Labview. [2]

The outcome of this master thesis will be a comparison between possibilities and opportunities of HLS versus the traditional design flow on RTL for FPGAs and ASICs. The implemented hardware modules deal on the one hand with a data flow application for large printing systems and on the other hand with a complex cryptography algorithm. The aim is an evaluation about the benefits or drawbacks of HLS regarding simple data flow operations and high sophisticated algorithms.

The master thesis is a cooperation with the company *Durst Phototechnik Digital Technology GmbH*, which is one of the market leaders in developing industrial printers. The headquarter is located in Brixen (South Tyrol) and a further development department

in Lienz (East Tyrol). The matter for this collaboration is the adoption of better design methodologies for their embedded systems and the balancing of opportunities to accelerate the design process.

1.1 Motivation

Since the packing density is increasing and the complexity of digital systems is also rising, the usage of new tools and methodologies are necessary to keep pace with the technology progress. Such a methodology for designing hardware in FPGAs or ASICs is as mentioned before HLS. The distinction to conventional design flow is that we are moving it to a more abstract level, which leads up to an amount of benefits [13]:

- **Improved productivity for hardware designers**
  Hardware designers can work at a higher level of abstraction while designing of new hardware modules or hardware systems.
- **Improved system performance for software engineers**
  Software engineers can transfer the CPU-intensive parts of their algorithms to co-processors like FPGAs or ASICs. This allows to improve the whole performance of the system.
- **Developing algorithms at a higher level like C-level**
  Raising the abstraction level and in further consequence reducing the implementation details to get a better overview about the whole system improve the design process.
- **Verify at the C-level**
  Verification of the correct behavior of a system is done much quicker than with traditional hardware description languages (HDLs).
- **High-level Synthesis process controlled by optimization directives**
  Engineers can define the exact behavior of the system after HLS by assigning directives instead of defining it on RTL.
- **Creation of multiple hardware implementations by assigning different optimization directives**
  The generation of different solutions helps finding the best implementation for a special system. This is done by assigning various optimization directives before the HLS process.
- **Create readable and portable C source code**
  Re-usage of C implementations for different FPGAs or ASICs can be done very easily. In addition, adding parts of an existing C implementation into a new project can also be achieved without troubles.

"Time is Money", this is the best argument, why it is very important to look for new methodologies to keep in pace with the technology evolution. The presented possibilities and opportunities of HLS give a short overview of this design technique for engineers.

1.2 Goals

Before starting a project, it is important to know about the outcome of the whole project. This master thesis is a cooperation with the company *Durst Phototechnik Digital Technology GmbH*. Therefore it should point out the sense of using new methodologies like HLS to decrease the time of generating designs or implementations for their printing systems.

Currently, the conventional hardware design flow, which means the designing on RTL, has been established in this company. While the complexity of digital systems are increasing steadily, it is necessary to consider new tools like HLS. This master thesis is an approach to compare the results of a design on RTL and a design created by HLS. Regarding results, there will be comparison between the different metrics of a hardware design (area, power consumption, and time-to-market) and it is possible to see the benefits of using Cosimulation for the hardware design flow.

In addition the practical outcome of the master thesis should present two different modules each designed on RTL and generated by HLS. One module deals with a data flow application, which is used in printing systems of *Durst Phototechnik Digital Technology GmbH* and the second module executes the cryptography algorithm called Ascon. Generally, the benefits and drawbacks of using HLS for simple data flow modules and computationally intensive applications are presented. The implementations are tested on a SoC of the company *Xilinx Inc.* SoCs consist of a FPGA part and a mounted processor, which acts like a powerful microcontroller. The implemented modules are participated on the FPGA.

1.3 Outline

In **chapter 2** the basics of HLS will be explained. The history and the evolution of HLS until now are part of this chapter. In addition, there are presented some areas of application for HLS, wherein this design methodology has got a major impact on engineering.

In **chapter 3** related works about the comparison of designing on RTL and HLS generated designs are presented. Several papers published in the last few years are summarized and the most interesting facts about the design quality between this two design techniques are shown. It gives an insight into expected results for this thesis.

Chapter 4 deals with the design process for the practical part of this master thesis. The main parts of this chapter are a general description of the generation of RTL implementations and the generation of a hardware module using HLS. In addition, the two different modules designed and implemented in this thesis are presented in more detail. This includes a description about the behavior of these modules. The evaluation kit is also mentioned in the last part of this chapter.

The implementation and the results of the HLS designed system and the RTL system is presented in **chapter 5**. Detailed information about the implementation is depicted by several state diagrams or data flow graphs. Further on, the different modules are compared accordingly design metrics like acquired area or utilized resources, power consumption, and time exposure.

A discussion of this topic and an outlook into the future will conclude this master thesis in **chapter 6**.

Chapter 2

Basics and Principle of High-Level Synthesis

As already mentioned this master thesis deals with designing of modules generated with HLS. For better understanding of the design steps, this chapter describes the evolution of HLS and the fundamental principle of the design flow.

2.1 What is High-Level Synthesis?

In the beginning of this chapter, the first part is to clarify the meaning of HLS and therefore we consider the principle of abstraction levels in the area of digital system design. The definition of raising the abstraction level implies the removing of details to simplify the description of a system. The higher the abstraction level the easier the definition of the design and also more details of the system are hidden. Figure 2.1 shows the elemen-

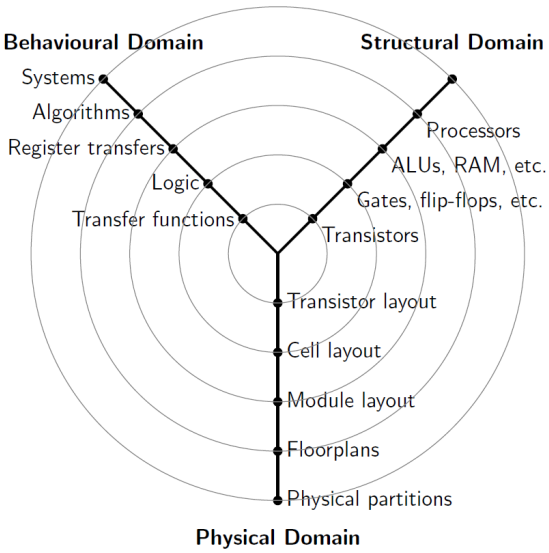


Figure 2.1: Abstraction model of digital system design. [7]

tary abstraction model for digital system design. It is called the Y-chart and has been introduced by Daniel Gajski and Robert Kuhn in 1983. Nowadays, HLS is the automatic process for the transition from *Algorithms* to *Register Transfers* on the behavioral domain. A detailed description can be found on [7].

Now, we will have a look on current FPGA design practices in more detail. This abstraction model consists of four levels. These are the following: Structural, Register-Transfer Level (RTL), Behavioral and High Level as shown in figure 2.2.

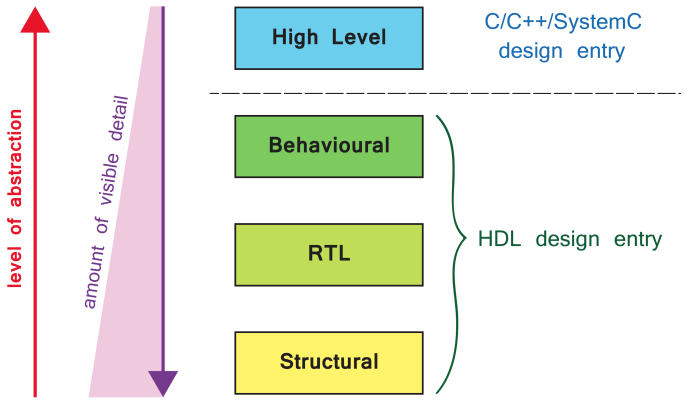


Figure 2.2: Abstraction levels of FPGA design process. [3]

The *Structural* is the lowest level of the abstraction model. It includes the instantiating and connecting of all components and elements of the hardware design. It is comparable to the level consisting of lookup tables (LUTs) and flip flops (FFs). On this level the designer has full control over the design and this leads to less capabilities of using automatic synthesis processes for optimization.

The abstraction level above is *RTL*. At this level the design is described by registers and operations. The technology aspect isn't considered anymore. Logic synthesis is used for creating hardware out of the RTL code.

The next abstraction level in order is the *Behavioral* description. The whole hardware behavior is designed by an algorithmic description. Hence, the designer has less control over the implementation, but the creation of a new hardware design can be achieved much faster. This is a crucial aspect for the establishment of this design level. Generally, HDLs are used for the implementation.

The uppermost abstraction level is called the *High-Level*. The design uses high-level languages instead of HDLs. One of the biggest benefit is, that the system is generated, simulated and verified as a high-level model, which is implemented in C/C++ or SystemC. Further on, the hardware architecture is generated by using HLS. So, the HDL code is generated automatically by this synthesis process.

For better understanding it's essential to add a short discussion about *logic synthesis* and *high-level synthesis (HLS)*. In the literature the notion *synthesis* is the same as *logic synthesis* and their meaning is the translation of HDL code into an associated netlist. In contrast, HLS is used to generate the HDL code out of a high-level model implemented in C/C++ or SystemC. In figure 2.3, the differences of high-level synthesis and logic synthesis are depicted.

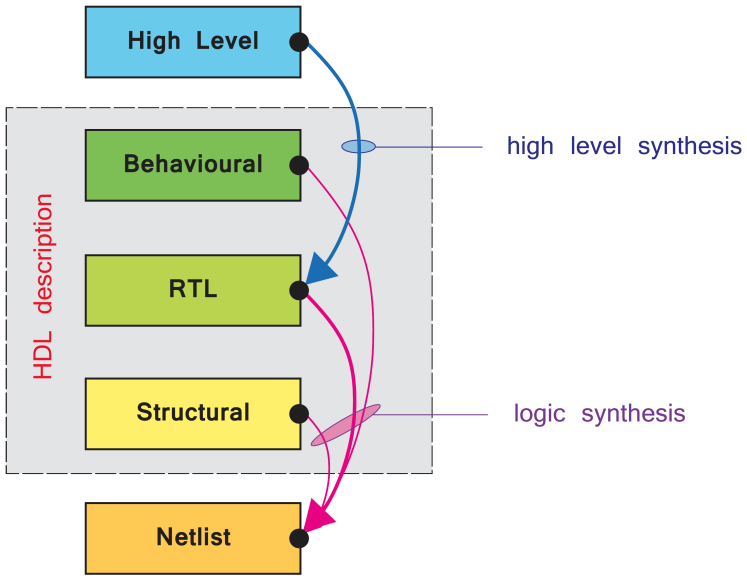


Figure 2.3: Difference between High-Level Synthesis and Logic Synthesis. [3]

2.2 Motivation for using High-Level Synthesis

The time to market is an important aspect in designing of new products. It is defined from the beginning of the concept phase until the release on market. A long duration for the design and development process of new products can affect negatively the sales of the company. Thus, researchers are looking for new methodologies to reduce the development time and keep up the system quality. Hence, HLS has been introduced some years ago. More details about the history are presented in chapter 2.3. [24]

One of the key aspects, why HLS is used, is the design productivity gap as depicted in figure 2.4. Over the years such productivity gaps occurred several times and new methodologies were introduced to keep pace with the transistor density (blue line) in integrated circuits, which corresponds also to the complexity of a system. The HLS is utilized to close the latest design productivity gap, i.e. HLS is an important method for future electronic designs.

In addition, the target of a high-level implementation is to get a simple and clear description of the system. It is possible to generate a design for FPGAs faster than using common hardware description languages (HDLs). One further important aspect during the design process is the ability to affect the resulting architecture by assigning optimization directives. More information about directives are presented in chapter 4. Generally, a redesign of the high-level source code isn't needed to change the resulting hardware design. Assigning various optimization directives influences the whole architecture, but it is not as exact as using HDL for the definition of hardware designs. So, engineers have to trust on the used HLS tool, because they have only restricted influence on the synthesis process.

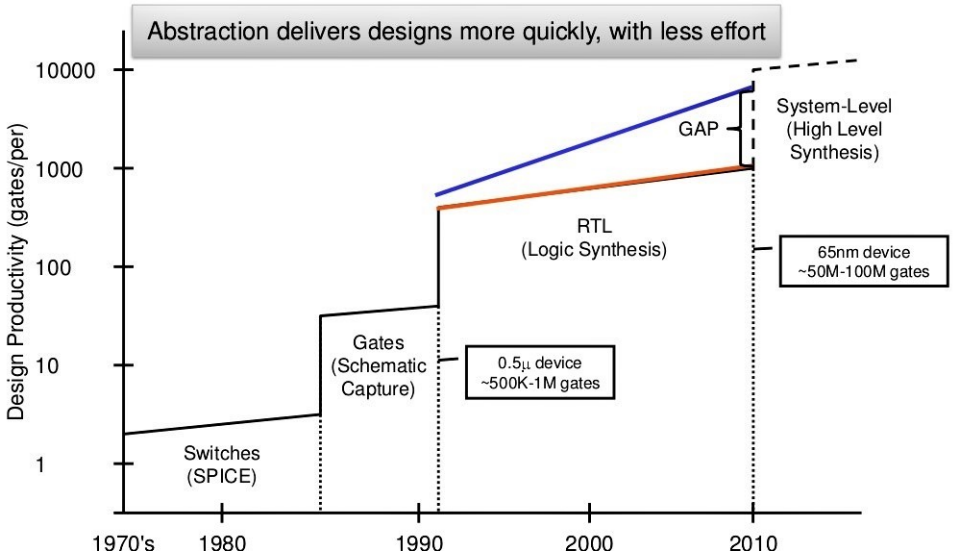


Figure 2.4: Closing the productivity gap with HLS. [5]

From a more practical point of view, engineers are more familiar with high level programming languages like C/C++ or SystemC instead of hardware description languages like VHDL or Verilog. The opportunity to create the basic framework of whole hardware designs on high level would simplify the design process. To sum up, one can say that the main benefit of HLS is the rising productivity. [3]

2.3 Review of High-Level Synthesis

The development of HLS is divided into three generations. The starting point was in 1970, but the first decade brought no seminal success. From 1980 to 1990 the first generation was developed and used. The next decade was called the second generation and since 2000 the third generation of HLS exists. Today, engineers are talking about a fourth generation, which should appear in the next few years. In the following, there will be a short description about the mentioned generations of HLS. [15]

2.3.1 Generation 1 (about 1980 to 1990)

In this decade, many different ideas have been appeared and they were written down in different research papers. Some important engineers were e.g. Pierre Paulin and John Knight, who investigated results in the area of scheduling for HLS. Other important results regarding HLS were presented by Daniel Gajski and his colleagues. They can be found in [6], [17], and [1]. In summary, the Generation 1 was mainly a development phase.

Further projects in the digital signal processing domain were e.g. the Cathedral and Cathedral-II project. This projects used a specific high-level language called Silage for the description of a DSP-system and the resulting tools were the first one, which enter the market. Although, the projects looked promising, they ultimately failed. The main reasons were indicated as follows: unnecessary need, unpopular input language and poor

quality of the results. The first reason "unnecessary need" is caused by the circumstance, that engineers were still not familiar with the adoption of RTL synthesis. So a behavioral synthesis was not successful for high-level models.

Another reason was the must to learn a new unpopular programming language called Silage. In times of establishing RTL synthesis, engineers were not ready to learn another language beside the hardware description languages (HDLs). Furthermore the resulting hardware design allocated much more resources than the traditional design process. So the poor quality of the implementation is also a reason for the failure of this generation. [15]

2.3.2 Generation 2 (about 1990 to 2000)

In this decade, companies like Synopsys, Cadence, and Mentor Graphics started the development of HLS tools and offered them on the market, although the focus for designing hardware systems was on the RTL synthesis. The new technology was tested by many engineers, but it was considered as insufficient in some reasons. The main reasons for the failure of this generation are listed in the following: [15]

- Criteria like the quality of results regarding area and performance were not improved. Thus hardware designer were not willing to admit this technology.
- The second generation of HLS used hardware description languages as input language. This would be in conflict with logic synthesis, which also uses hardware description languages as input. Hence, engineers were not satisfied to change their design process from RTL synthesis to HLS.
- The validation of the resulting RTL code was hard to achieve. The model had maybe the same behavior, but getting the right timing often evoked some unpredictable problems.

Finally one can say that the benefits for HLS in generation 2 were not given to replace the traditional RTL synthesis. Hence, engineers preferred the common design process for their hardware systems.

2.3.3 Generation 3 (from about 2000)

Through the years, a lot of vendors has brought their HLS tools on the market. Mentor Catapult C Synthesis, Celoxica, Bluespec or Synfora PICO are some of them, just to name a few. Many of them are using C/C++ or SystemC as input language. Few tools has also invented their own description language. In comparison to the previous generations, this HLS generation was succeeded, because: [15]

- many of the tools deliver a good quality of results, which is the main reason why this tools are established. In addition, the HLS tools are focusing on the synthesis of data flow or DSP design applications.
- Previous tools had their own input languages. Now, many tools are using programming languages, which are familiar for engineers, e.g. C/C++. The effort to learn utilization of this tools will become much lower.

- Another reason is the technology advance. Today, the demand on data processing systems are increased. As mentioned in chapter 2.2, keeping up with the increasing capability of integrated circuits requires new tools to reduce the time-to-market.

2.3.4 Commercial Progress of High-Level Synthesis

In the amount of sales, one can identify the popularity of HLS tools. In figure 2.5, there is the revenue of electronic system-level synthesis tools listed. If we take a closer look on the individual years, we can recognize that there was an experimental phase from 1994 to 1996. From 1997 to 1999 the first useful tools were emerged on the market, but they were oversold. Thus, the interest of engineers decreases, which results in fewer sales.

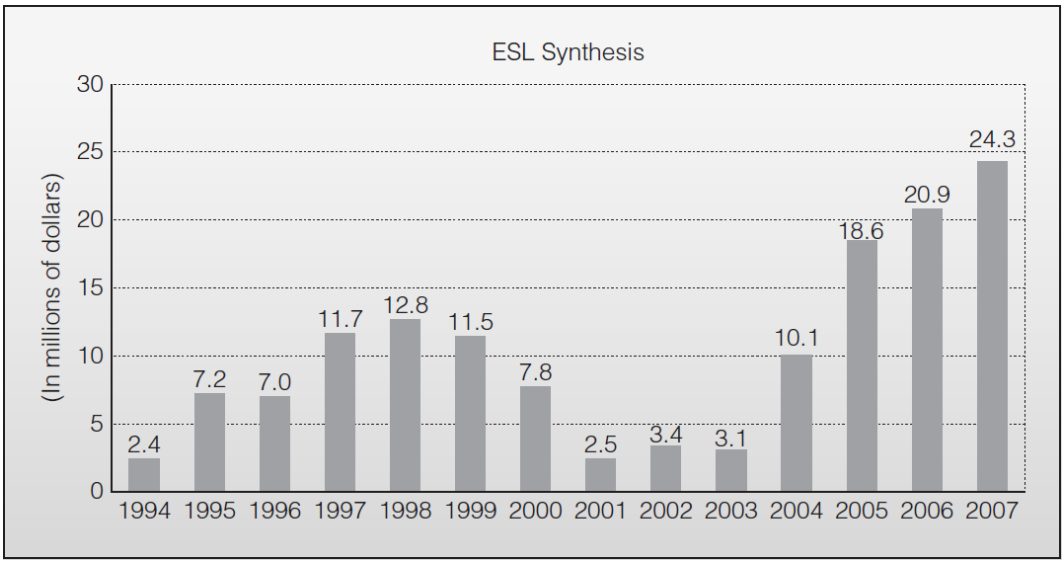


Figure 2.5: Sales of electronic system-level synthesis tools. [15]

Next the compiler of the third generation were published, which led to higher revenue for HLS tools again. In the next couple of years a fourth generation will emerge on the market. About the last few years there are no numbers of the amount of sales available, but according to some reports of engineers the popularity of HLS tools is still increasing.

2.3.5 Outlook - fourth Generation

As already mentioned the tools of generation three delivers a good quality of results according to the data flow and control. The actual problem is, that the individual tools are not perfect for all different domains and that is exactly the point, where the fourth generation of HLS will have their exploration space. Tools, which cover all application domains would simplify the whole design process and companies have only to adopt one new tool to reduce the time-to-market for their systems. [15]

2.4 HLS Kernel

New methodologies facilitate raising the abstraction level for digital system design. Based on the increased transistor density looking for new methodologies would be an inalienable process. The utilization of HLS allows engineers accelerating the design process and still getting a high efficient hardware.

The first step in every design process is the specification of the system behavior. Starting point is a high-level model of the intended functionality, which should be implemented on a co-processor or a custom hardware unit. The high-level model contains less importance on the timing of the behavior. So, the operations of the high-level description don't have any delay and the data are processed as a software program. In addition, the variables and data types of the high-level model are not adjusted for a hardware design. Hardware designs need bit-accurate data types with an appropriate length. Hence, engineers have to mind the necessary bit-length of the individual variables to get a good quality and an optimized hardware design. Standard data types, which are used in software, are mostly not very suitable, because the variables would be over-sized.

The result of the HLS process is a timed implementation. The HLS tool takes a high-level model and transforms it into a clocked hardware description. The generated architecture is created automatically or sometimes semi-automatically and is designed as efficient as possible. The resulting architecture consists of a data path and a controller, whereby the data path contains registers, multiplexers, functional units, etc. and the controller is responsible for the general behavior of the given specifications. [2]

2.4.1 Key Concepts

The process of HLS is depicted in figure 2.6. Starting point is the high-level model with some design constraints and a RTL component library. During the synthesizing, the following tasks are executed. Additionally, they will be explained in more detail in chapter 2.4.2 - 2.4.6.

1. Compilation of the high-level model
2. Allocation of the hardware resources like registers, functional units, etc.
3. Scheduling of the individual operations
4. Binding of the operations to functional units
5. Binding of variables to storage elements
6. Binding of data transfers to buses
7. Generation of the whole resulting RTL architecture using HDL

As seen in figure 2.6 the synthesis tasks for allocation, scheduling and binding can be done in parallel to achieve an optimized architecture, because of their dependences to each other. However, the computationally intensive effort synthesizing all tasks simultaneously leads to a specific order of the synthesis process, whereby the chosen order has a big impact on the quality of the resulting hardware regarding utilized resources. This is a reason, why every HLS tool works better or worse for different application domains.

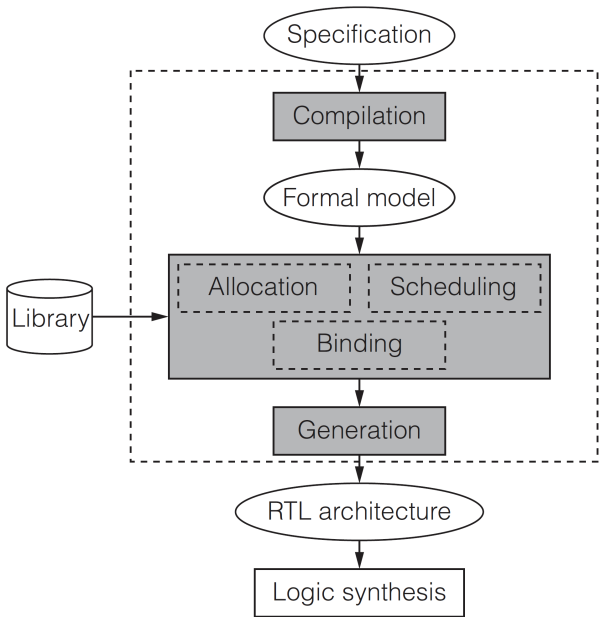


Figure 2.6: High-Level Synthesis tasks. [2]

2.4.2 Compilation and Modeling

As described in [2], first step of HLS is the compilation of the high-level model. This means, that the model has to be checked out for dead-code, wrong data dependencies and constant folding and loop transformations have to be done. The resulting model contains the data and control dependencies, whereby the data dependencies are depicted with data flow graphs (DFGs). Such DFGs can be generated easily by removing the control parts of the high-level model, i.e. the loops has to be completely unrolled and conditional operations have te be substituted with multiplexed definitions. The DFGs consist of nodes, which represent the operations, and arcs for the definition of input and output as well as the temporary variables. DFGs are very useful for many designs, but for models with *goto*-statements or unbounded number of iterations DFGs have their problems. Hence, such high-level descriptions are represented by using advanced control and data flow graphs (CDFGs). The edges depict the control flow and the nodes contain a sequence of statements without e.g. branches or exit points. Conditional statements like *if* or *switch* are represented by the edges. Summarized, CDFGs are used to show the data dependencies of basic blocks and the control flow between some of them. A detailed description of DFGs and CDFGs can be found in [19].

Generally, the benefit of CDFG compared to DFG is that it is more powerful, because as mentioned before they are able to describe unbounded loops. However, the representation of parallelism seems to be more difficult. The reason is that it takes place within some basic blocks. Description of parallelism between different blocks are not so obvious. These would need additional directives like loop unrolling, loop pipelining, loop merging, etc. In addition, such directives have impact on the metrics (throughput, area, power) of a hardware design.

2.4.3 Allocation

Next synthesis task according to [2] and [3] is allocation. This step maps resources like functional units, registers, etc. by using the RTL library to each operation of the formal description of the high-level model. So the generated DFG or CDFG is used for this task. The exact procedure depends on the HLS tool. Some of them just allocate resources like buses or connection elements before or after the binding or scheduling task. Thus, the quality of the resulting hardware architecture varies according to the used HLS tool. The RTL library consists of all possible components including their metrics like area, delay and power consumption.

2.4.4 Scheduling

For the timing of the high-level model it is necessary to assign each operation of the DFG or CDFG into clock cycles. For instance, the variables $var1$ and $var2$ of the operation in equation 2.1 have to be read and executed by the assigned component and the result has to be written to the destination, which could be a register or another functional unit. The number of clock cycles depends on manually chosen directives. So, several operations can be scheduled in such a way that they are executed in parallel as long as no data dependencies exist. But keep in mind that parallelism results in a higher number of utilized resources.

$$res = val1 + val2 \tag{2.1}$$

2.4.5 Binding

The binding process is used to assign each variable to a storage element like registers. If there are variables, which have a non-overlapping lifetime, they can use the same element. In addition, several operations has also to be assigned to functional units. Here, too, different operations can use the same functional unit as far as it is not assigned to another operation at the same clock cycle. Otherwise the binding process has to optimize the scheduling of the functional units to keep the number of it as low as possible to achieve the same behavior. Furthermore, the binding of connection units affects the binding of functional and storage units. The order of the individual binding steps depends on the used HLS tool and binding algorithm.

2.4.6 Generation

The last step of the high-level synthesis is the generation of the RTL architecture. In this step the HLS kernel creates the HDL-files considering all the defined directives, which have been made manually during the whole design process.

2.5 Resulting Architecture

The architecture primarily consists of a controller and a data path, whereby the data path includes storage elements (registers, memories), functional units (multipliers, shifters, ALUs, etc.) and elements for the interconnection (buses, multiplexers, etc.). The amount of allocated elements depends on the respective application. The data path is controlled

by the controller and its control signals. The controller is designed as a finite state machine (FSM). The changes of the states are affected by input signals. They are controlled by an user or an other hardware module. In particular, the controller consists of a state register, a next-state logic and an output logic. A detailed description can be found in [2]. Figure 2.7 shows a simple example. On the left hand side the controller is depicted and on the right hand side the data path is shown.

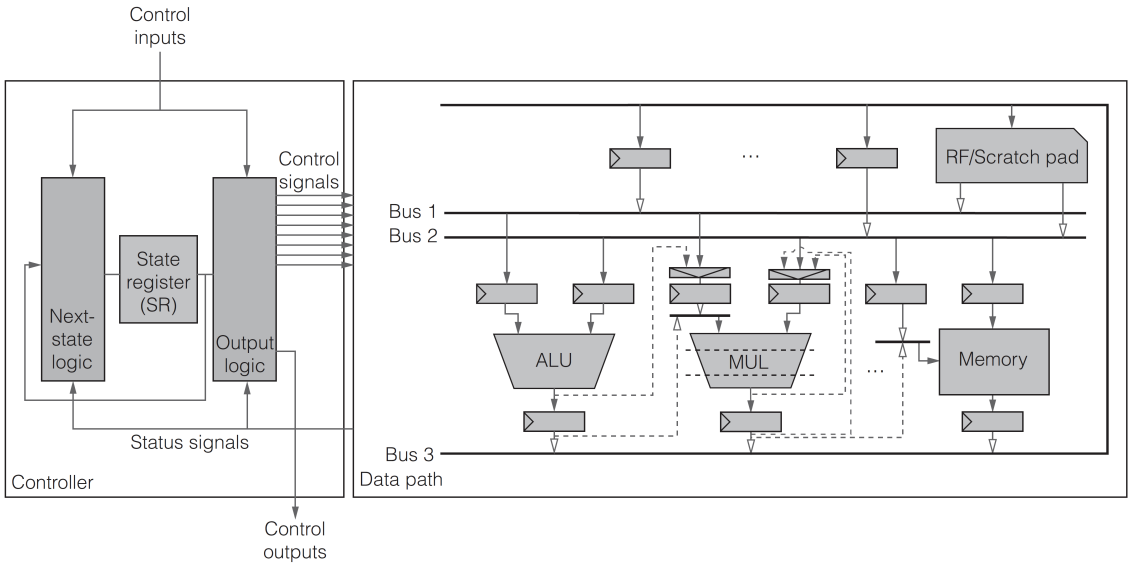


Figure 2.7: Architecture of a HLS generated design. [2]

2.6 HLS-based Development Flow

In figure 2.8, the design flow of a HLS generated hardware design is shown. First step is to implement the intended design with a high-level language like e.g. C/C++. For the transformation by the HLS tool, some modifications have to be added manually. These are e.g. defining of the interfaces, adding some directives to the design (pipelining, parallelization, defining the latency, etc.). The selected modifications are depending on the desired design according to metrics. The output equals to the HLS-ready C code and is verified about their correct behavior by test vectors. As long as everything is correct, the HLS is performed for transforming the C code into HDL. At this point, the RTL description is checked again by a hardware functional verification. The further steps equates to the traditional FPGA design flow. Sometimes the amount of allocated resources for the hardware design exceeds the specifications, therefore the reference C code has to be revised to improve the HLS-ready C code. Afterwards, HLS generates another different RTL design, which acquires the given specifications.

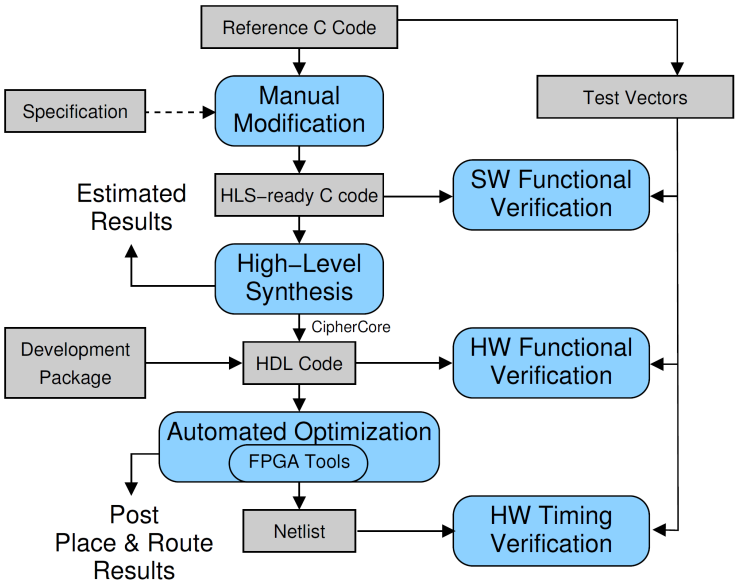


Figure 2.8: HLS design flow. [9]

Chapter 3

Related Works

The research area of the productivity of different HLS tools is very important for the customers of commercial HLS tools and also for the vendors of them. Companies, which are offering systems consisting of FPGAs, know that the traditional way of implementing FPGAs take a long design time. Hence, they are looking for methods to decrease the development time. As already mentioned in previous chapters, HLS tools are developed to raise the abstraction level, to ease the designing and to reduce the time-to-market. Raising the abstraction level also means to remove design details, which are generated automatically during HLS. This chapter reflects related publications about the quality of the hardware design generated by HLS.

3.1 HLS generated designs vs. HDL designs

There exist several papers about a comparison of HLS generated designs versus RTL designs. The most important publications will be described in the following, which deals with the comparison of cryptographic algorithms, a video filter algorithm and a sub-pixel interpolation filter from the MPEG HEVC standard.

3.1.1 Cryptographic Algorithm

The first comparison between HLS generated designs and RTL designs deals with hash functions (detailed description about hash functions can be found in [14]) or more specifically with the finalists of the SHA-3 contest. On the one hand the hardware of the algorithms was designed on RTL with traditional HDLs and on the other hand with the high-level language C using Xilinx Vivado HLS. Figure 3.1 depicts the design, which consists of three modules: input processor, hash core and output processor. The gray boxes within the modules contain signals, which are generated automatically by the HLS tool according to the optimization directive of the interfaces, and the black written signal names are generated by the engineers. For example port *din* of the input processor creates automatically the ports *din_empty_n* and *din_read*.

In addition, it is necessary to swap the input data and the outgoing data of the hash module to the appropriate endianness for getting the right functionality. This is illustrated in figure 3.1 by the *swap-endian* symbol.

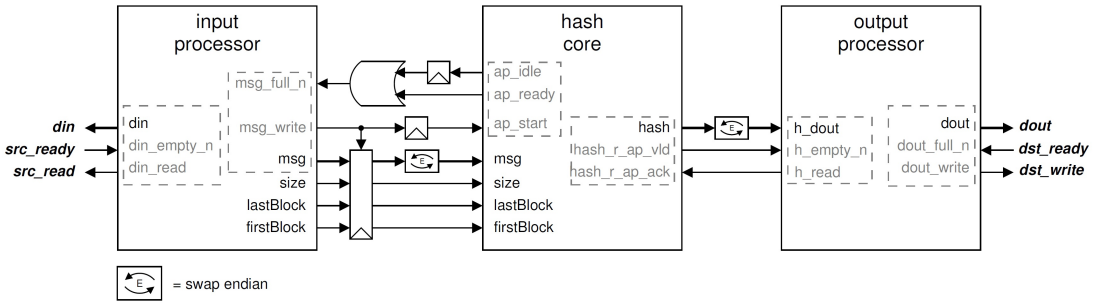


Figure 3.1: Top-Level of SHA-3 Implementation. [10]

Moreover, before synthesizing the high-level model adding optimization directives are unavoidable to get a good quality of the hardware architecture. More details can be found in [10].

Results of the comparison between HLS and RTL design

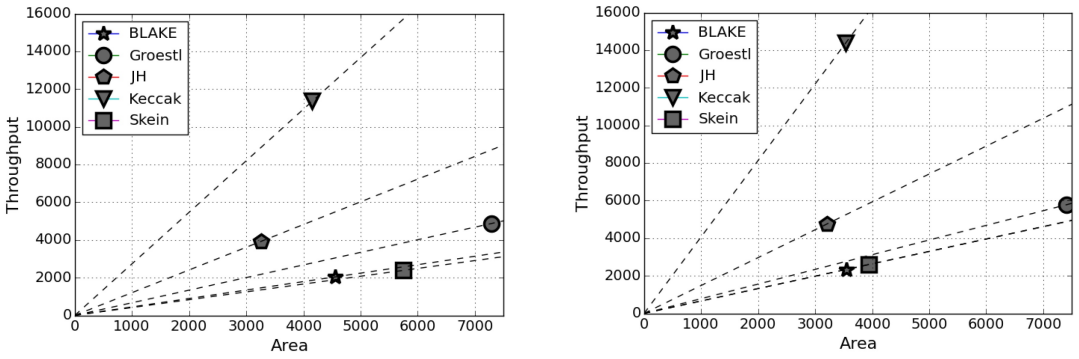
The results of the implementations, i.e. RTL design and HLS generated design, for all different hash functions are shown in table 3.1. Column two to five show the maximal frequency, throughput, area, and the ratio of throughput over area for the HLS generated design. The next four columns show the results for the RTL design and in column nine to 13 the ratio of the RTL design over the HLS generated design is presented. It is possible to see that the results for the RTL model are slightly better according to all mentioned hash functions. Only in few cases, the required area is a little bit smaller by using the HLS generated design. Generally, the comparison is done for the implementations of two different FPGA types: Altera Stratix IV and Xilinx Virtex 6.

Altera Stratix IV												
	HLS				RTL				RTL/HLS			
	Freq.	TP[1]	A[2]	TP/A[3]	Freq.	TP	A	TP/A	Freq.	TP	A	TP/A
BLAKE	119.6	2040	4557	0.45	132.3	2337	3543	0.66	1.11	1.15	0.78	1.47
Groestl	218.8	4871	7290	0.67	236.9	5776	7404	0.78	1.08	1.19	1.02	1.17
JH	336.8	3919	3256	1.20	399.7	4759	3210	1.48	1.19	1.21	0.99	1.23
Keccak	271.4	11356	4156	2.73	317.7	14401	3541	4.07	1.17	1.27	0.85	1.49
Skein	97.9	2387	5752	0.41	96.2	2592	3936	0.66	0.98	1.09	0.68	1.59
Xilinx Virtex 6												
	HLS				RTL				RTL/HLS			
	Freq.	TP	A	TP/A	Freq.	TP	A	TP/A	Freq.	TP	A	TP/A
BLAKE	150.6	2570	1289	1.99	126.1	2226	1257	1.77	0.84	0.87	0.98	0.89
Groestl	242.7	5403	2016	2.68	296.1	7220	1870	3.86	1.22	1.34	0.93	1.44
JH	291.8	3395	1141	2.98	454.6	5412	849	6.37	1.56	1.59	0.74	2.14
Keccak	211.2	8838	1494	5.92	261.2	11839	1086	10.90	1.24	1.34	0.73	1.84
Skein	107.3	2616	1426	1.83	125.2	3373	1005	3.36	1.17	1.29	0.70	1.83

Table 3.1: Results of HLS generated design and RTL design (Frequency in MHz, Throughput (TP) in MBits/s, Area (A) in ALUTs for Altera Stratix IV and in CLB-slices for Xilinx Virtex 6). [10]

[1] TP = throughput
[2] A = area
[3] TP/A = ratio of throughput over area

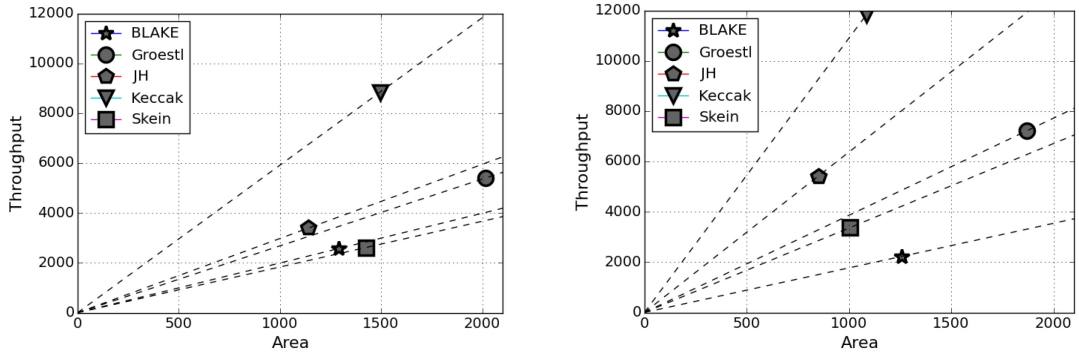
Better visualization of the presented results are provided in the following two figures. Figure 3.1.1 shows the relation of throughput to area for the Altera Spartix IV. As one can see the highest throughput for a specific area is achieved by the hash function called *Keccak*, whereby the RTL design achieve a higher throughput than the HLS generated design.



HLS generated Design RTL design

Figure 3.2: Comparison of the results for Altera Spartix IV. [10]

The same figure exists for Xilinx Virtex 6. As in figure 3.3 shown, the highest gradient of the curves provides *Keccak*, again. Generally, the RTL design achieves a higher throughput for a specific number of allocated chip area.



HLS generated Design RTL design

Figure 3.3: Comparison of the results for Xilinx Virtex 6. [10]

Summarizing, it's mentionable that *Keccar* has been the winner of the SHA-3 contest in 2012. The comparison of RTL models and HLS generated designs presents, that the creation of HDL-written models provide better results according to area or throughput.

3.1.2 Video Filter Algorithm

A further comparison between HLS generated designs and HDL models for a video filter algorithm is published in the master thesis [28] of Michael D. Zwagerman. It deals with the implementation of an image filter technique. In particular, the technique calculates

the convolution of an input image and the kernel as depicted in figure 3.4. The high-level model is written in C and the RTL design is implemented with VHDL. For the HLS of the high-level model, Xilinx Vivado HLS is used. In earlier years, this tool was called AutoPilot developed by AutoESL.

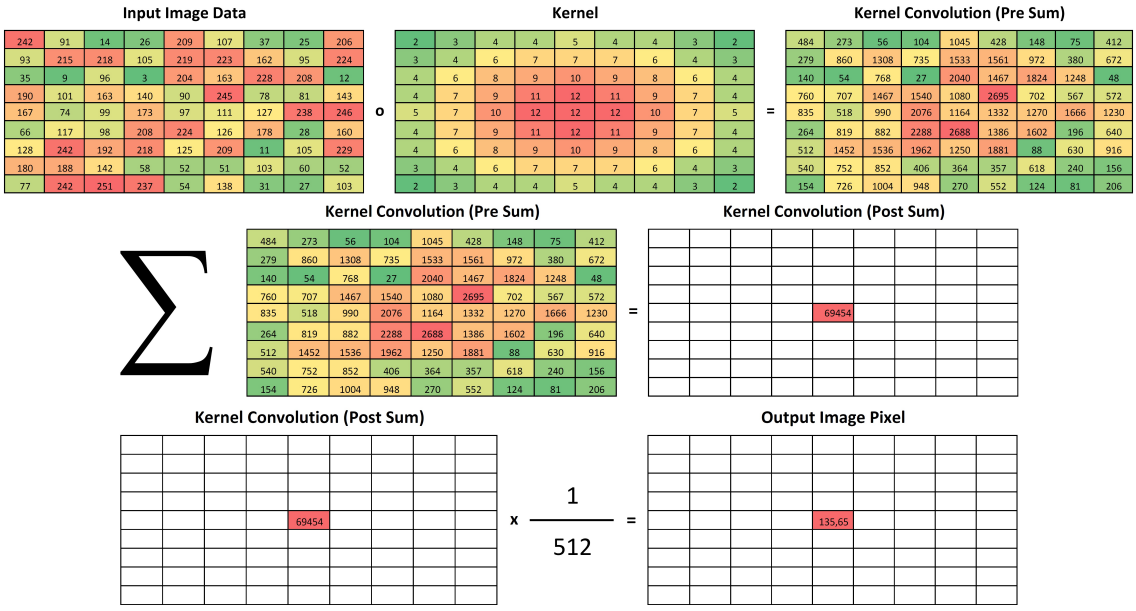


Figure 3.4: Image filter algorithm description. [28]

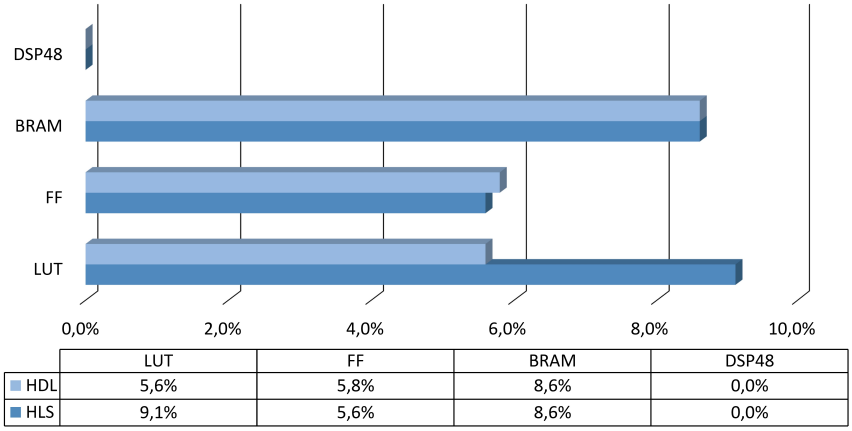
By executing this filter on a full HD film, the convolution of the 9x9 kernel generates 167,961,600 multiplications per color channel and per frame. The procedure starts with a *Hardamard*-product, which corresponds to the element-wise multiplication. Further on, all elements are summed up and finally the sum is normalized.

Results of the comparison between HLS and RTL design

In table 3.2 and figure 3.5, the results for the video filter algorithm are shown. The amount of utilized LUTs for the HLS generated design are about 61.5% higher than the RTL design requires. The number of FFs, BRAMs and DSP48s are more or less the same.

	RTL	HLS	Total Available
LUT	2989	4827	53200
FF	6139	5970	106400
BRAM	12	12	140
DSP48	0	0	220

Table 3.2: Results of the resource utilization for Video Filter Algorithms.



	LUT	FF	BRAM	DSP48
HDL	5,6%	5,8%	8,6%	0,0%
HLS	9,1%	5,6%	8,6%	0,0%

Figure 3.5: Utilized resources for the video filter algorithm (f_{max} constrained at 150 MHz). [28]

3.1.3 Sub-Pixel interpolation filter from the MPEG HEVC standard

The publication [23] deals with a comparison of HLS generated hardware design and a RTL design for a sub-pixel interpolation filter. The high-level model is implemented in CAPH ([25]) and the RTL design is written in VHDL. The following software tools were used:

- Altera Quartus II
- Mentor Graphics Modelsim ASE
- CAPH Compiler

The HEVC interpolation filter is used for video compression. The filter consists of a shift register and eight filter coefficients, which are multiplied with the pixels (figure 3.6). The block of pixels can be left shifted by 1/4 and 3/4 using seven taps and by 1/2 using eight tabs (information can be found in [26]).

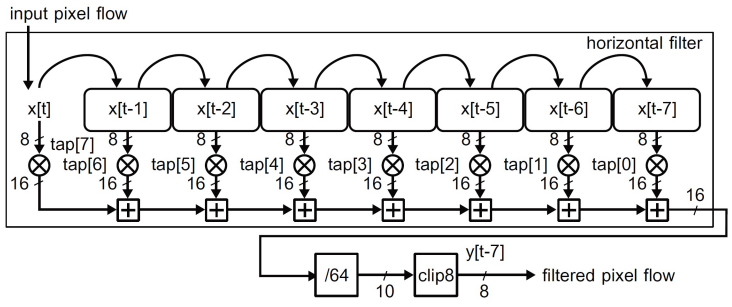


Figure 3.6: Horizontal interpolation filter (1-D). [23]

For Filtering of a 2-D image eight horizontal 1-D filters are necessary. The result of each horizontal filter is multiplied with further coefficients to get also an upper shift of 1/4, 1/2, and 3/4. Figure 3.7 shows the block diagram of the whole interpolation filter.

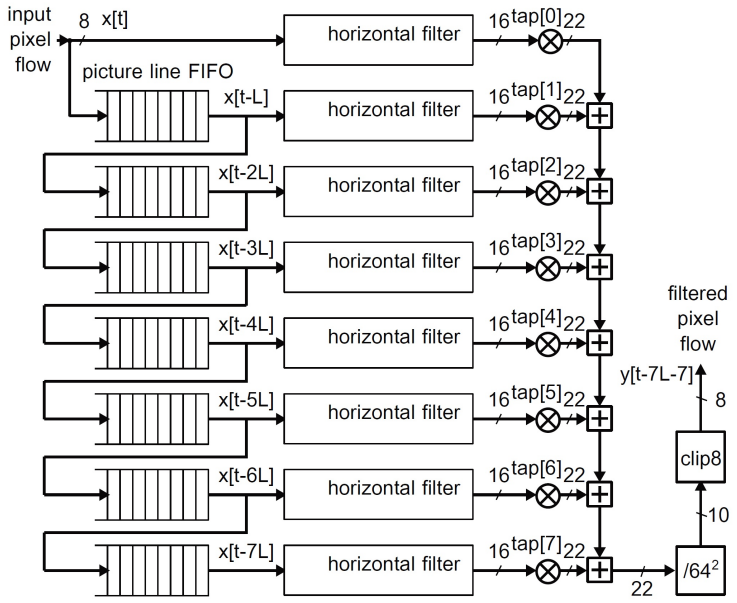


Figure 3.7: Horizontal interpolation filter (2-D). [23]

Results

Firstly, for better understanding of the results some definitions have to be explained. One parameter is the global non-recurring engineering G_{NRE}. The global non-recurring engineering is the ratio of the design and verification time of the RTL model over the design and verification time of the HLS generated model (equation 3.1). This parameter shows the ability to save time by using HLS, i.e. a value greater than one corresponds to less design effort for HLS generated designs than for RTL designing.

$$G_{NRE} = \frac{t_{design}^{HDL} + t_{verif}^{HDL}}{t_{design}^{HLS} + t_{verif}^{HLS}} \tag{3.1}$$

Moreover, the definition of the parameter for the quality loss L_Q is a little bit more complex and is described in equation 3.2, whereby the variable lut is the number of LUTs, reg corresponds to the number of necessary registers, ram is the number of allocated BRAMs, lat equals to the latency and prd is the operating period. Additionally, these variables are normalized and are multiplied with some weights α_i. More details are described in [23].

$$\begin{aligned} L_Q = & \frac{\alpha_1 \cdot lut_{norm}^{HLS} + \alpha_2 \cdot reg_{norm}^{HLS} + \alpha_3 \cdot ram_{norm}^{HLS} +}{\alpha_1 \cdot lut_{norm}^{HDL} + \alpha_2 \cdot reg_{norm}^{HDL} + \alpha_3 \cdot ram_{norm}^{HDL} +} \\ & \frac{\alpha_4 \cdot dsp_{norm}^{HLS} + \alpha_5 \cdot lat_{norm}^{HLS} + \alpha_6 \cdot prd_{norm}^{HLS}}{\alpha_4 \cdot dsp_{norm}^{HDL} + \alpha_5 \cdot lat_{norm}^{HDL} + \alpha_6 \cdot prd_{norm}^{HDL}} \end{aligned} \tag{3.2}$$

The weights are defined as follows

$$\alpha_i = \begin{cases} 0 & \text{if } \max(\phi_i^{HLS}, \phi_i^{HDL}) = 0 \\ (max(\phi_i^{HLS}, \phi_i^{HDL}))^{-1} & \text{otherwise} \end{cases} \tag{3.3}$$

whereby for instance the variable ϕ_i are equal to the number of LUTs, REGs, BRAMs, DSP slices, etc.

$$\phi_1 = lut_{norm}^{HLS/HDL} \tag{3.4}$$

In table 3.3, the results of all three different versions are listed. Version 1 deals with a horizontal filter like in figure 3.6. Version 2 is very similar to version 1, but some distinctions according to the interface type are made. This version is designed for the modular smart camera called DreamCam. Finally, version 3 is the implementation of the 2-D interpolation filter.

	VHDL v1	CAPH v1	VHDL v2	CAPH v2	VHDL v3	CAPH v3
t_{design}	358	103	162	71	232	187
$t_{verification}$	288	65	783	72	775	169
# SLOCs	147	43	333	61	805	194
# characters	4114	1351	9465	2395	22072	6099
# LUTs	193	226	282	3161	2868	11398
# Regs	81	103	115	2209	1252	7557
# BRAMs	0	0	0	0	18	14
max. Frequency	64.7	68.0	71.8	83.0	65.2	84.2

Table 3.3: Results for the sub-pixel interpolation filter (Frequency in MHz, time in minutes). [23]

Better understanding of the results for the two design methodologies provides table 3.4. There are listed the values in relation to each other. A value greater than one for parameter G_{NRE}, which corresponds to the time exposure, means that the HLS design requires less time for designing and implementation. As one can see, HLS designing is less time consuming for all three versions then designing of the corresponding RTL model. In addition, for getting a good HLS generated hardware design the quality loss L_Q should be as low as possible. This means that the higher the value the poorer the HLS generated design. If this value equals to one, the HLS generated design is qualitative as good as the RTL design. Finally, the parameter design-productivity P_D equals to the ratio of the global design effort G_{NRE} over quality loss L_Q. A value greater than one equals to a sensible usage of HLS for these system specifications.

	CAPH vs. VHDL v1	CAPH vs. VHDL v2	CAPH vs. VHDL v3	Average
G_{NRE}	3.84x	6.60x	2.82x	4.42x
L_Q	1.70x	2.53x	1.47x	1.90x
P_D	2.26x	2.61x	1.92x	2.26x

Table 3.4: Design Gain. [23]

In summary, considering only resource utilization RTL models achieve better results than HLS generated models, but considering the whole design process including time exposure, resource utilization, etc. using HLS for the designing is the better choice.

3.2 BDTI High-Level Synthesis Tool Certification Program

The BDTI[1] High-Level Synthesis Tool Certification Program (HLSTCP)[20] was initiated by the Berkeley Design Technology, Inc. The goal is to represent the quality of hardware designs using HLS. Hence, customer should see the benefits establishing HLS tools in their product design flow. In particular, this certification program shows on the one hand the performance and efficiency of designs generated by HLS and on the other hand the usability for engineers, which includes the ease-of-use and the design productivity. In addition, the results are not only compared to traditionally handwritten RTL designs, but also with designs running on a DSP processor. As yet, two different HLS tools are tested. The first tool is the *AutoPilot* from *AutoESL*, whereby nowadays this HLS tool is offered by *Xilinx Inc.*, and the second is *Synopsys C Compiler*.

The evaluation consists of two applications. Application one deals with a video processing algorithm, which is used for the analysis of motion. In particular, they recognize the motion of objects by using a movie with a resolution of 720p. The outcome of the algorithm are matrices for the horizontal and vertical motion in the movie. This application is designed twice. On the one hand it should minimize the resource utilization and on the other hand the throughput should be maximized. The second application consists of a DQPSK receiver with an input stream of 18.75 MSamples/s at a frequency of 75 MHz. The output data rate equals to 4.6875 Mbits/s. For this application the resource utilization should be minimized.

3.2.1 Evaluation results of AutoPilot

In this section the results of the HLS tool from AutoESL called AutoPilot are presented. As already mentioned before, the first application deals with a video processing algorithm. The corresponding results are shown in the next two tables, whereby table 3.5 contains the results for minimizing the resource utilization by fixing the throughput to 60 frames per second and table 3.6 shows the values for maximizing the throughput. [21]

As one can see in table 3.5 (video processing algorithm, minimizing resource utilization, fixed throughput) the chip cost for a quantity of 10,000 adds up to $ 26.65 and is a little

Platform	Chip Unit Cost (USD, Quantity 10,000)	Chip Resource (Lower is Better) Utilization
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$ 26.65	39%
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$ 21.25	N/A (a minimum of 12 DSPs would be required to meet this operating point)

Table 3.5: Comparison of HLS generated and DSP design for video processing applications - **minimizing resource utilization**. [21]

[1] BDTI = Berkeley Design Technology, Inc.

bit more expensive than a design using DSP processors. In addition, 39% of the resources are allocated.

In comparison to that, the same design for maximizing the throughput yield to the same design cost. However, the achieved throughput is 183 frames per second for the HLS generated design and resulting cost per frame is much lower than a DSP design. Summarizing, RTL designs generated by a HLS tool are more suitable for high speed data processing.

Platform	Chip Unit Cost (USD, Quantity 10,000)	Chip Resource Maximum Frames per Second (FPS)	Cost per FPS (Lower is Better)
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$ 26.65	183	$ 0.14
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$ 21.25	5.1	$ 4.20

Table 3.6: Comparison of HLS generated and DSP design for the video processing application - **maximizing throughput**. [21]

Table 3.7 depicts the results for the hardware design of the DQPSK receiver. This application was written in C for using the HLS tool and additionally it was hand-written with RTL code. The comparison shows that the number of allocated resources by the HLS tool is lower than by the hand-written design. Summarizing, it is difficult to say that this results are valid for all application, but it's true to say that this application is suitable for using HLS. However, the difference of the resource utilization is very small and therefore both design techniques are sensible.

Platform	Chip Resource Utilization (Lower is Better)
AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	5.6 %
Hand-written RTL code using Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	5.9 %

Table 3.7: Comparison of HLS generated and DSP design for a DQPSK receiver. [21]

3.2.2 Evaluation results of Synphony C Compiler

The BDTI program provides also the similar results for the *Synopsys Synphony C Compiler*. The chip resource utilization for the video processing algorithm design is nearly the same as previous. Only 0.6% more resources are allocated in relation to the *AutoPilot*. The chip cost is constant.

Platform	Chip Unit Cost (USD, Quantity 10,000)	Chip Resource (Lower is Better) Utilization
Synopsys Synphony C Compiler plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$ 26.65	39.6%
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$ 21.25	N/A (a minimum of 12 DSPs would be required to meet this operating point)

Table 3.8: Comparison of HLS generated and DSP design for the video processing application - minimizing resource utilization. [22]

Furthermore, the *Synopsys Synphony C Compiler* delivers better efficiency according to maximizing the throughput of the same application. The number of frames per second is increased to 204 FPS and so it is a little bit better than using *AutoPilot*.

Platform	Chip Unit Cost (USD, Quantity 10,000)	Chip Resource Maximum Frames per Second (FPS)	Cost per FPS (Lower is Better)
Synopsys Synphony C Compiler plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	$ 26.65	204	$ 0.13
Texas Instruments software development tools targeting the TMS320DM6437 DSP processor	$ 21.25	5.1	$ 4.20

Table 3.9: Comparison of HLS generated and DSP design for the video processing application - maximizing throughput. [22]

Finally, the second application (table 3.10), which deals with the DQPSK receiver, is analyzed. In this case, the chip resource utilization is higher by using the *Synopsys Synphony C Compiler* than the traditionally hand-written RTL design. Concluding one can say, that the difference is very small and therefore the quality of the design methodologies are quite similar.

Platform	Chip Resource Utilization (Lower is Better)
Synopsys Synphony C Compiler plus Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	6.4 %
Hand-written RTL code using Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA	5.9 %

Table 3.10: Comparison of HLS generated and DSP design for the video processing application - maximizing throughput. [22]

3.3 Comparison of different HLS tools

In paper [16] different HLS tools were considered and compared with each other. The several scopes are depicted in figure 3.8. They are structured into: design experience, implementation, tool capabilities and quality of the results. The scope called *design experience* deals with the valuation of documentation and easiness of learning HLS tools. *Implementation* assesses the ease of implementation and the complexity of the abstraction level. Furthermore, *tool capabilities* include the additional features, which are provided by the HLS tool. Finally, this kind of evaluation for different HLS tools shows also the quality of the generated RTL model.

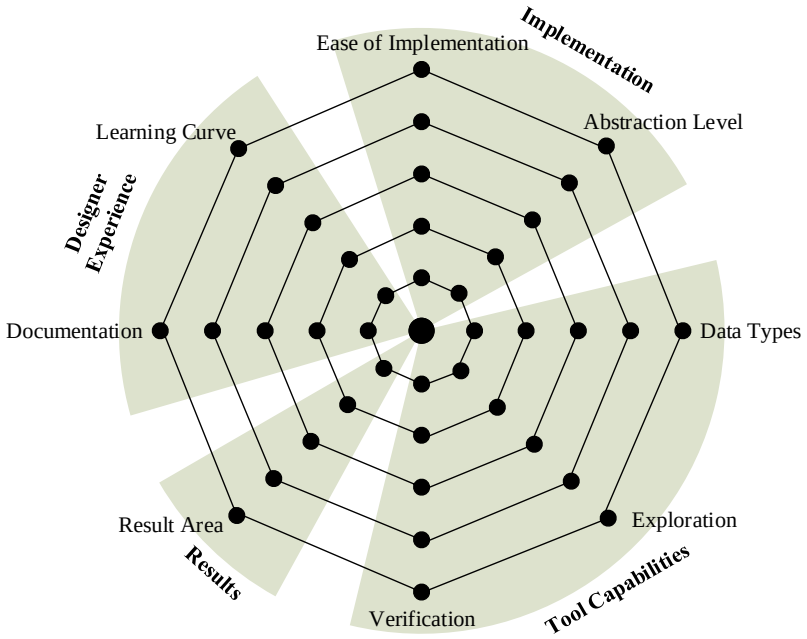


Figure 3.8: Evaluation factors for HLS tools. [16]

In figure 3.9 one can see an evaluation of some selected HLS tools. The left upper plot shows the evaluation of *Autopilot*. As already mentioned, the *AutoPilot* was developed by *AutoESL* and was purchased by *Xilinx Inc.* some years ago. Usable input languages are C, C++ or even SystemC and it is easy to optimize the design according to the desired metric (throughput, area, power) . Hence, as one can see in the spyder diagram, the implementation gets a very good assessment. Data types can be designed as arbitrary precision data types to avoid unnecessary over-sized data types. Apart from that, AutoPilot got a very good evaluation for the transformation from a high-level description to the RTL model.

Another popular HLS tool is the *Synopsys Synphony C Compiler*. According to the implementation scope, this HLS tool resembles to *AutoPilot*. The resulting area of the RTL model generated by this HLS tool is bigger compared to other tools, because additional input and output buffers are allocated. Generally, one can say that the overall evaluation for this tool is worser than using the HLS tool mentioned before.

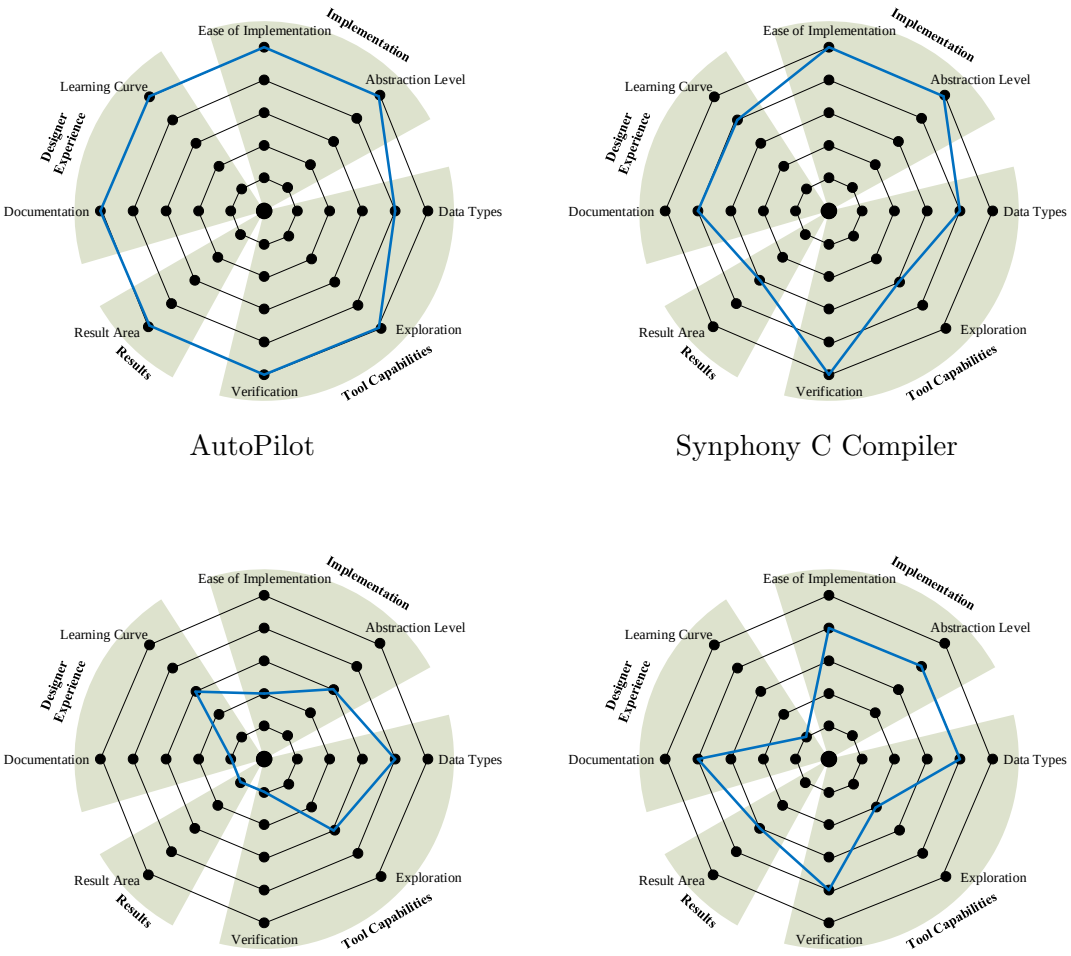


Figure 3.9: Comparison of different HLS tools. [16]

The next described tool is *Impulse CoDeveloper* developed by *Impulse Accelerated Technologies*. The high-level model is implemented in ImpulseC, an own extension to C. The evaluation of the implementation (ease-of-implementation, abstraction level) is worser, because of adopting a new high level programming language. Other scopes of evaluation are also much worser. The resulting area of the generated RTL design is very huge compared to the other HLS tools. So generally one can say, that *AutoPilot* or *Synopsys Synphony C Compiler* are better suited for HLS.

Finally, the last HLS tool picked out of [16] is *C-to-Silicon*. This tool is developed by Cadence. It is mainly used for the designing of ASICs, whereas the tools above are used for generation of FPGA designs. Generally, *C-to-Silicon* achieves ordinary better assessment as *Impulse CoDeveloper*.

Summarizing, AutoPilot delivers the best overall results according to evaluation results of this paper. Further details about the evaluation itself or about the results can be found in [16].

Chapter 4

Design

This chapter deals with tasks, which are necessary to reflect before the implementation can be done, because the differences of generation a RTL model and a HLS generated design have to be known. First of all the specifications of the resulting hardware module are defined. Afterwards, some basics about hardware implementations will be described and then the procedure of RTL and HLS designing is mentioned. Finally, some design deliberations about the two implemented modules and the choice of the evaluation kit are done in section 4.5, 4.6 and 4.7.

4.1 Specifications

As mentioned in the beginning of this thesis, target is the comparison of design metrics, e.g. area, power, and time exposure, for RTL designs and designs generated by HLS. The following specifications for this master thesis are defined by the electronic development department of the partner company *Durst Phototechnik Digital Technology GmbH*:

- Many of the existing FPGA designs of the company deals with controlling of a huge amount of data. Generally, this modules are designed on RTL with FSMs achieving a very high data throughput. Part of the thesis is the evaluation of the design results according to metrics using HLS. In future, it could be useful adopting HLS for the design of such "data flow" modules.
- In addition, the image pre-processing, which is essential for printing systems, deals with very complex algorithms. This requires a lot computational effort. The utilization of FPGAs would bring an increase of computational power. Therefore, the second specification is the evaluation of using HLS for the generation of hardware modules implemented different algorithms. A cryptography algorithm is implemented on RTL and generated by HLS for this evaluation.

The following sections describe the way to the implementation of the two modules. Chapter 5 presents the implementation itself and the results of the comparison of the two modules.

4.2 Metrics of FPGA Designs

The metrics of a FPGA design have influence on the implementation. They determine the design process on RTL and assigning of optimization directives to interfaces, loops, etc. of the high-level model. Generally their exists three main metrics. They are listed in the following:

- area
- power
- throughput

In many cases, engineers generate a hybrid solution. Only a low area design without consideration on the power or throughput don't achieve a good solution. In addition, the correlation of throughput and resource cost should be linear. The figure 4.1 shows this linearity. Designs, which don't conform to this behavior, are poor solutions. If they are located under this line, the amount of allocated resources exceeds the necessary number to achieve the throughput.

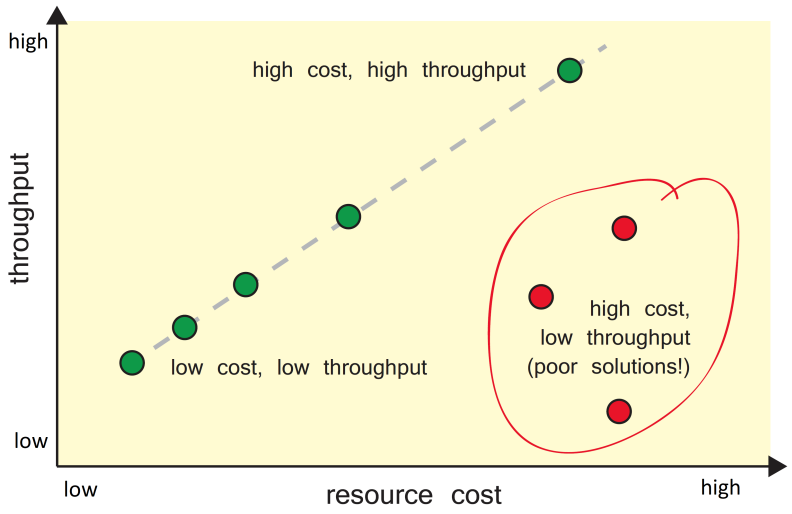


Figure 4.1: Correlation of throughput and resource cost. [3]

The target of this master thesis is designing hardware modules with high throughput, because they are used in printing systems, which have to handle a huge amount of data for the control of printing headers. Important factors are clock frequency or data latency. It is useful maximizing the clock frequency and minimizing the data latency of the modules to achieve this design requirement.

4.3 Generation of a Register-Transfer-Level Design

Figure 4.2 depicts the design flow of a module designed on RTL. Starting point is a description of the behavior by HDLs like VHDL or Verilog. The implementation process includes the synthesis, mapping and place/route procedure. This steps generate the resulting hardware module.

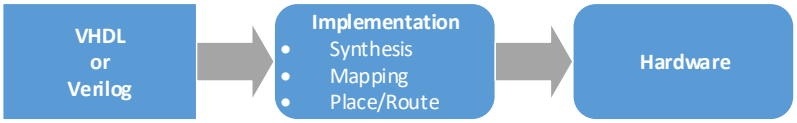


Figure 4.2: RTL design flow.

4.3.1 Programming Languages on RTL

On RTL, several different programming languages are available for the designing. All of them has their own benefits and drawbacks. Some of the available languages are listed in the following:

- ABEL (Advanced Boolean Expression Language)
- AHDL (Altera Hardware Description Language)
- Bluespec
- VHDL
- Verilog

The most popular languages are Verilog and VHDL. Verilog is used mainly in North America and VHDL is adopted mainly in Europe. In addition, VHDL is suitable for system level design, whereas Verilog is used for "low level" hardware implementations. European universities utilize VHDL, because the trend is towards designing on higher level or system level. Generally, the choice of the used HDL includes three aspects. First of all, the experience of the engineers is the main factor. The time, which is necessary to pick up a new programming language is very time-consuming and can be avoided. Second, for the usage of VHDL or Verilog different tools are necessary. Sometimes tools are already available for free or have to be bought. And third, sometimes companies or institutes have a preference for a specific language, because e.g. other projects use them or libraries are written by them. [27]

For this master thesis the hardware description language VHDL is used, because all of the mentioned arguments are true. The following example gives an overview for the designing of hardware using VHDL.

RTL Example: Matrix Multiplication

The example of a matrix multiplication of [8] has three ports, i.e. two input ports and one output port. All of them are assigned a specific type, which is defined in the working package. This package is depicted in listing 4.1. First, the type *vec_1d* is defined as an array of *std_logic_vector* with a width of eight bits and the final type of the matrix *vec_2d* is an array of the type *vec_1d*.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package mult_2x2_pack is
6
7  type vec_1d is array(integer range 0 to 1) of std_logic_vector(7 downto 0);
8  type vec_2d is array(integer range 0 to 1) of t_1d_array;
9
10 end  mult_2x2_pack;
```

Listing 4.1: Package for Matrix Multiplication.

The following Listing 4.2 shows the hand-written code of the matrix multiplication. As already mentioned, this implementation consists of three ports, which are defined in line 15 to 19 within the *entity* block. Generally the *entity* is mainly used for defining the interfaces. The actual behavior is determined in the *architecture* block. This part is defined next after the *entity*. Within the architecture, there are clarified processes. In the example, the process calculates the multiplication itself. The sensitivity list contains the variables *in_matrix1* and *in_matrix2*. The process is executed, if one of this variables was changed.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.mult_2x2_pack.all;
7
8  entity mult_2x2 is
9  port (
10   in_matrix1 : in  vec_2d;
11   in_matrix2 : in  vec_2d;
12   out_matrix : out vec_2d
13 );
14 end mult_2x2;
15
16 architecture rtl of mult_2x2 is
17 begin
18
19   process (in_matrix1, in_matrix2)
20   begin
21     for i in 0 to 1 loop
22       for j in 0 to 1 loop
23         out_matrix(i)(j) <= std_logic_vector(
24         signed(in_matrix1(i)(j)) * signed(in_matrix2(j)(i)) +
25         signed(in_matrix1(i)(j)) * signed(in_matrix2(j)(i)));
26       end loop;
27     end loop;
28   end process;
29 end rtl;
```

Listing 4.2: Matrix Multiplication.

This example gives a short introduction for those, who don't be familiar with VHDL. Generally, there exists much more functions, which can be used for designing a specific application. The basics about hardware development on RTL are described very briefly.

4.3.2 Tools for the generation of RTL models

A further step in the generation of RTL models is the choice of the design tool. There are several different vendors. Hence, it is always the first step in the design process of RTL models to select an appropriate tool. One of the most popular FPGA developer is Xilinx Inc., which also provides a HLS tool. Since the *Durst Phototechnik Digital Technology GmbH* is using FPGA solutions of this vendor in many printing systems, it is obvious to choose this design tools. The design flow from the RTL design down to the design debugging is shown in the following picture.

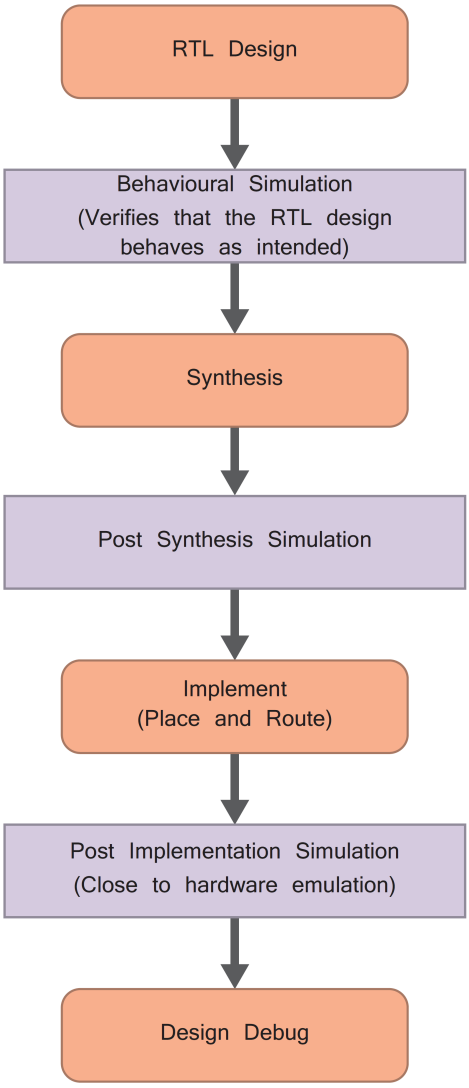


Figure 4.3: Design flow of the Xilinx Vivado Suite. [3]

The design tool of Xilinx Inc. is called Xilinx Vivado Design Suite. As shown in figure 4.3, the design flow consists of several steps. The behavior of an application is implemented in the first step. The design tool offers a behavioral simulation on this level. This is used to verify the correct behavior of the hand-written RTL code. The next step is

the logic synthesis or also called synthesis. The RTL code is transformed into logic level. In further consequence, it is possible to do a post synthesis simulation. Thereby engineers can distinguish between a functional and a timing simulation. The functional simulation checks only the behavior of the synthesized implementation and the timing simulation considers also the parasitics of all gates and the different timing parameters of the storage elements. Timing parameters are e.g. the setup and hold time of a gate. The setup and hold time determine the duration of how long the data at a clock edge must be fixed previously and afterwards at a storage element. Otherwise a wrong behavior appears. The next step of the FPGA design flow is the implementation. This includes all steps, which are necessary to place and route the generated netlist, whereby logical, physical and timing constraints of the design are considered. Likewise the generated implementation can be simulated on this level. As mentioned before it's possible to do a functional and/or a timing simulation. In addition, the implementation generates a bitstream, which will be uploaded on a FPGA.

4.4 Generation of High-Level Model

Besides the generation of the RTL model, a high-level model with the same behavior and functionality is part of this master thesis. The way to the final high-level design is described in the following paragraphs. Due to the fact that hardware generation by using high level programming languages isn't as popular as using hardware description languages. The design steps are explained in more detail.

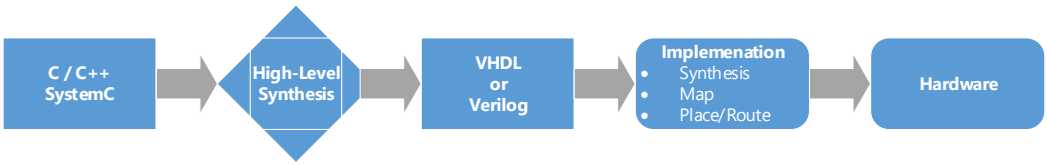


Figure 4.4: HLS design flow.

4.4.1 Tools for High-Level Synthesis

For the HLS process it is necessary to choose an appropriate tool. Generally there exists a lot of different vendors providing their own tools. This variety leads to difficulties selecting a suitable HLS Compiler, because all of them has their individual benefits and drawbacks. The basic feature influencing the choice is the input language. The most famous input language is C/C++. Some vendors developed their own high level programming language with the aim to improve resulting RTL designs. Another important feature is the output language, i.e. it has to correspond to the tool, which is used for the logic synthesis (transformation of the RTL behavior into gate level). In the following table 4.1 most commonly used tools are depicted.

In this thesis, the HLS tool called Vivado HLS is used for the synthesis process. It is described briefly in the following paragraph. The followed description of the high-level

Compiler	Owner	License	Input	Output
Bambu	PoliMi	Academic	C	Verilog
Bluespec	BlueSpec Inc.	Commercial	BSV	SystemVerilog
Catapult-C	Calypto Design Systems	Commercial	C/C++ SystemC	VHDL/Verilog SystemC
CHC	Altium	Commercial	C subset	VHDL/Verilog
CoDeveloper	Impulse Accelerated	Commercial	Impulse-C	VHDL
CtoS	Cadence	Commercial	SystemC TLM/C++	Verilog SystemC
CyberWorkbench	NEC	Commercial	BDL, SystemC	VHDL/Verilog
Cynthesizer	FORTE	Commercial	SystemC	Verilog
DK Design Suite	Mentor Graphics	Commercial	Handel-C	VHDL/Verilog
DWARV	TU. Delft	Academic	C/C++	VHDL
eXCite	Y Explorations	Commercial	C	VHDL/Verilog
GAUT	U. Bretagne	Academic	C/C++	VHDL
LegUp	U. Toronto	Academic	C	Verilog
MaxCompiler	Maxeler	Commercial	MaxJ	RTL
ROCCC	Jacquard Comp.	Commercial	C subset	VHDL
Synphony C	Synopsys	Commercial	C/C++	VHDL/Verilog SystemC
VivadoHLS	Xilinx	Commercial	C/C++ SystemC	VHDL/Verilog SystemC

Table 4.1: Available HLS tools on the market. [18]

modules dealing with a data flow application and a cryptography-algorithm is mainly done using this HLS tool. Furthermore, Table 4.2 presents more important properties about different HLS tools. An interesting feature is the generation of a testbench by using high-level languages. In addition, floating point operations are not supported by every HLS tools. So, this is also a criteria when choosing the HLS tool. However, in this thesis fixed point or floating point operations are not used and therefore this is negligible.

Compiler	Year	TB	FP	FixP
Bambu	2012	Yes	Yes	No
Bluespec	2007	No	Yes	No
Catapult-C	2004	Yes	No	Yes
CHC	2008	Yes	No	Yes
CoDeveloper	2003	Yes	Yes	No
CtoS	2008	Only cycle accurate	No	Yes
CyberWorkbench	2011	Cycle/Formal	Yes	Yes
Cynthesizer	2004	Yes	Yes	Yes
DK Design Suite	2009	No	No	Yes
DWARV	2012	Yes	Yes	Yes
eXCite	2001	Yes	No	Yes

GAUT	2010	Yes	No	Yes
LegUp	2011	Yes	Yes	No
MaxCompiler	2010	No	Yes	No
ROCCC	2010	No	Yes	No
Synphony C	2010	Yes	No	Yes
VivadoHLS	2013	Yes	Yes	Yes

Table 4.2: Properties of different HLS tools (TB - Testbench, FP - Floating Point, FixP - Fixed Point). [18]

Originally, AutoESL developed the HLS tool called AutoPilot. In 2011 Xilinx Inc. adopted this tool for their design process and was called Vivado HLS [12]. Input languages are C/C++ or SystemC. In addition, optimizations can be assigned to the design for controlling the synthesis process, i.e. loops can be merged, unrolled or pipelined. In addition, the latency, iteration interval (number of cycles to load new data into the computation) or maximal possible frequency for the hardware module can be influenced. The synthesis process of Vivado HLS is depicted in figure 4.5. It needs C/C++ or SystemC files as input. These files include the definition of the high level model. A complete high level design also contains a high level testbench for the verification of the implemented model. The verification process will be described in 4.4.3. Adding constraints and directives enables user impact on the automated high-level synthesis. Directives and constraints are

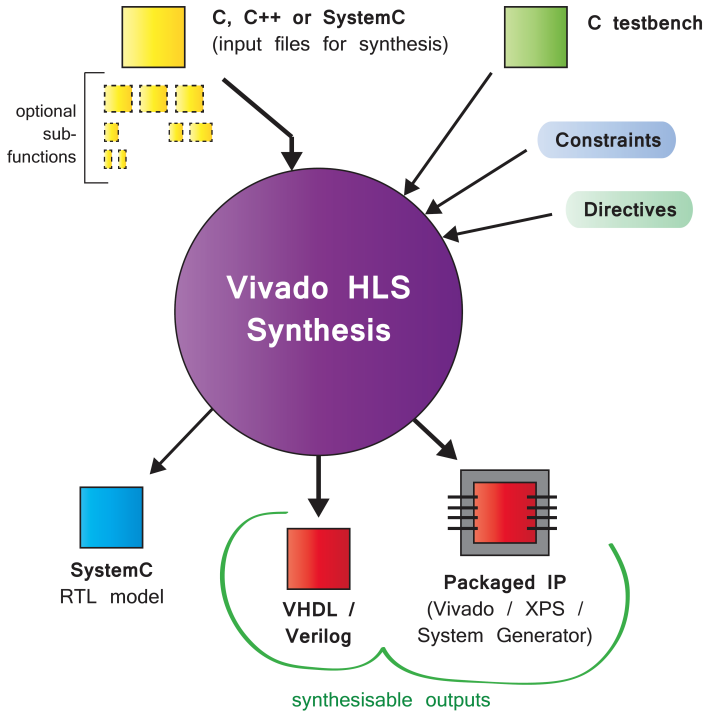


Figure 4.5: Synthesis process of Vivado HLS. [3]

described in 4.4.3 and 4.4.4. Outcome of the synthesis is the RTL model generated in SystemC or VHDL/Verilog. In addition, Vivado HLS provide an automatic generation of a packaged IP, which can be used in Xilinx Vivado Suite for the programming of FPGAs or additionally in Matlab (System Generator).

4.4.2 Programming languages for High-Level Synthesis

As already mentioned, the used programming language for the high-level model depends on the selected tool. Most of those support C, C++ or SystemC, whereby SystemC is an extension of C++. It enables the generation of modules in software, which is very similar to the generation of hardware on RTL. But there are no experience necessary according to utilization of hardware description languages like VHDL. Next, SystemC is introduced and evaluated about the suitability for the implementation of the modules in chapter 5.

SystemC: language for High-Level synthesis

SystemC is an extension to C++ or provides features that improves the productivity for engineers generating electronic systems. Hardware and Software are developed together. This means, that the implementation includes the software syntax as well as the hardware modules and they can be simulated simultaneously. In the following paragraphs the most important constructs are explained briefly.

Modules define parts of the high-level model, which should be implemented in hardware. The definition looks like shown in listing 4.3. The modules communicate with others via ports and can also include further modules.

```
1 SC_MODULE (module name) {
2 // module content
3 };
```

Listing 4.3: SystemC: Definition of a module.

An entity will be generated by invoking the constructor. The syntax is depicted in the following listing.

```
1 SC_CTOR (modul name) {...}
```

Listing 4.4: SystemC: Creation of a module.

As mentioned before, the communication between the modules is done via ports. They can be defined as input, output and bidirectional (read and write data on the same port) ports. Additionally, signals can be defined for the communication within modules.

```
1 sc_in<Porttyp> PortInName; // input
2 sc_out<Porttyp> PortOutName; // output
3 sc_inout<Porttyp> PortInOutName; // bidirectional
4 sc_signal<Signaltyp> SigName; // signal
```

Listing 4.5: SystemC: Port definition.

The actual behavior of the modules are defined as processes. Generally there exists three different types. One of them is called **SC_METHOD()**. This process is invoked, if a variable of the sensitivity list is changed.

```
SC_METHOD (function name);
```

Listing 4.6: SystemC: process type 1.

The sensitive list is defined as depicted in listing 4.7.

```
sensitive << Signal1 << Signal2 ...
```

Listing 4.7: SystemC: sensitivity list.

Another process is called **SC_THREAD()**. These are started only once and can be paused by wait-statements.

```
SC_THREAD (function name);
```

Listing 4.8: SystemC: process type 2.

At latest, processes called **SC_CTHREAD()** are synchronous. The second parameter defines trigger of the clock edge. There is no sensitivity list necessary.

```
SC_CTHREAD (function name, clock edge);
```

Listing 4.9: SystemC: process type 3.

As one can plainly see, the extension called SystemC enables features for the generation of hardware. Generally, it is very useful for huge systems containing software and hardware parts. Within this thesis the functionality of SystemC is not necessary. For the comparison of RTL designs and HLS generated modules, it's sufficient to implement them with plain C++.

4.4.3 Cosimulation

Verification is the foundation of every design. Otherwise the right behavior can't be ensured. The Cosimulation is described in figure 4.6. On the left side, the original testbench for functional verification is depicted. The testbench itself includes the code for the generation of the test vectors. These are passed to the C implementation of the modules and to the golden reference. Alternatively, the input test vectors can be got by reference files, which include also the golden reference output vectors. The test fails, if the results of the C implementation doesn't match the results of the golden reference. Additionally, implementation of the testbench can include the reporting the number of fails.

Cosimulation implies the automatic test of the generated RTL design. On the right side of figure 4.6, the test procedure or Cosimulation is depicted. Generally, the testbench is transformed into RTL like the high-level model. The input test vectors are provided by the testbench. These are passed to the HLS generated model and to the golden reference. Finally the outputs are compared and analysed. Like before there exists an alternative test procedure. The golden reference provides input test vectors and the corresponding output vectors, which are compared with the output of the implemented module to verify the correctness.

As already mentioned, the testbench is generated automatically during HLS. Manual transformation is not necessary. Generally, verification is always absolutely essential. Vivado HLS provides this automatic step. Thus, the design process can be accelerated and the automatic RTL testing increases the productivity. The automatic generation of the testbench also reduces the error-proneness. However, one have to keep in mind that

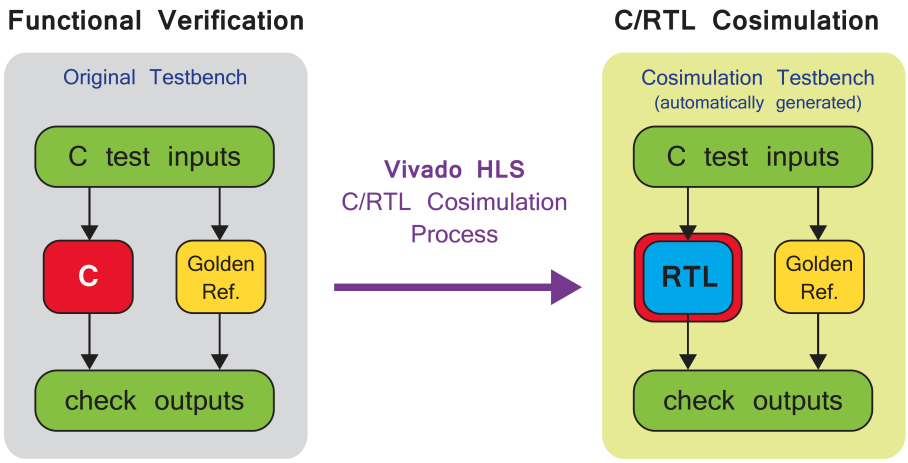


Figure 4.6: Procedure of C/RTL Cosimulation. [3]

Cosimulation takes place on RTL and RTL simulation conforms to behavioral simulation. Parasitics of gates or other timing impacts on the resulted RTL model are neglected. [3]

4.4.4 Interface Synthesis

Generally, high-level models implemented in C/C++ are designed as functions, whereby the functional parameters can be defined as input, output or bidirectional. A variable is called input, if the data is read from the parameter and never written to it. An ouput is the inverse, data is written to the parameter and never read from it. Some arguments are bidirectional, they are read from and written to it. In most cases a hardware design consists only of input and output ports.

Block-level interface define the control between several hardware blocks in a FPGA design. It ensures the correct data flow in a system. Vivado HLS provides three different types of protocol, which are listed in the following. [3]

- *ap_ctrl_none*: No additional protocol is used for the hardware block. The control of the data flow between several different blocks is done by manually defined ports. No ports are generated automatically.

- *ap_ctrl_hs*: This protocol provides a handshaking mechanism, i.e. four different ports are generated automatically during the synthesis. The first one is an input port called *ap_start* and is used to start the processing of the hardware block. The other three output ports are used to get information about the actual state of the block. The signal *ap_ready* indicates that new data can be read, *ap_idle* shows if data is processing and *ap_done* indicates that data can be read from or data is ready on the output port.

- *ap_ctrl_chain*: Similar to the protocol before, this protocol provides also a handshaking mechanism and additionally a port called *ap_continue*. This port is used to pause the hardware block. If ap_continue is Low, the hardware block will stop the processing of the current data and waits until the same signal is High. Then the computation with new data will continue.

A further important step during the design process is the interface type definition for the arguments of a function written in C/C++. There exists several different types, but not all of them are suitable or can be used for every kind of variable. In figure 4.7, every combination is depicted, whereby **D** symbolize the default type. This will be used, if nothing else is assigned. The letter **S** stands for supported type. The argument type *variable* can be only used as input port, except for the interface type s_axilite, which is together with axis and m_axi an own bus protocol developed by Xilinx Inc. and isn't taken into account in further consequence. The other three argument types can be used as input, output, or bidirectional ports like shown in the following figure.

Argument Type	**Variable**			**Pointer Variable**			**Array**			**Reference Variable**		
	pass-by-value			**pass-by-reference**			**pass-by-reference**			**pass-by-reference**		
Interface Type[a]	**I**	**IO**	**O**	**I**	**IO**	**O**	**I**	**IO**	**O**	**I**	**IO**	**O**
ap_none	**D**	-	-	**D**	S	S	-	-	-	**D**	S	S
ap_stable	S	-	-	S	S	-	-	-	-	S	S	-
ap_ack	S	-	-	S	S	S	-	-	-	S	S	S
ap_vld	S	-	-	S	S	**D**	-	-	-	S	S	**D**
ap_ovld	-	-	-	-	**D**	S	-	-	-	-	**D**	S
ap_hs	S	-	-	S	S	S	S	-	S	S	S	S
ap_memory	-	-	-	-	-	-	**D**	**D**	**D**	-	-	-
bram	-	-	-	-	-	-	S	S	S	-	-	-
ap_fifo	-	-	-	S	-	S	S	-	S	S	-	S
ap_bus	-	-	-	S	S	S	S	S	S	S	S	S
axis	S	-	-	S	-	S	S	-	S	S	-	S
s_axilite	S	-	S	S	S	S	-	-	-	S	S	S
m_axi	-	-	-	S	S	S	S	S	S	S	S	S

Figure 4.7: Interface protocols (Interface type: I - input port, IO - bidirectional port, O - output port; Abbreviations: S - supported; D - default). [3]

The following bullet points explain the different interface types:

- *ap_none*: This interface protocol doesn't generate additional control signals. The timing between input and output data must be considered during the design process.
- *ap_stable*: This is generally the same protocol as *ap_none*. No additional control signals are available, but it is useful for input ports, which change very rarely. They

haven't to be stored in registers, because this protocol keeps automatically the input data as a constant. In addition, it can only be utilized for input ports.

- *ap_ack*: This protocol can be used for input and output ports. For input ports an additional signal is generated, which equals to High in the same cycle during the port is read. Output ports with this protocol create an input signal called acknowledge and is used to identify that the provided data on the output port is used by the next hardware block before operations resume.

- *ap_vld*: Selecting protocol *ap_vld* implies the automatic generation of an additional port for validating the data. Data provided on input ports can be confirmed as valid data with this port. For output ports, it indicates that the provided data is valid in the same clock cycle and can be read.

- *ap_ovld*: The difference to protocol *ap_vld* mentioned above is that it can only be used for output ports or the output part of a bidirectional interface port.

- *ap_hs*: A handshaking mechanism is generated by this protocol. Thus, additional signals like *ap_ack*, *ap_vld*, and *ap_ovld* are created. The meaning of this signals is the same as described above. This protocol can be utilized for inputs and outputs. The handshaking control ensures the correct data communication between to different hardware modules. The drawback is that additional control ports lead to a design overhead.

- *ap_memory*: This protocol is used for arrays. It is suitable for all three types of ports. Furthermore, three additional control signals are necessary. These are a clock, a write signal and an address port to read or write the correct data from the memory.

- *bram*: This protocol is similar to the *ap_memory* protocol. The difference is, if the module is packed into an IP, the control ports are represented by one single port.

- *ap_fifo*: A further protocol for arrays is ap_fifo. It is suitable for input and output ports. The data is passed sequentially to the module. Thus, an address port is not necessary. The generated control ports are used to check if the buffer is full or empty and this port stops and continues the processing of the module.

- *ap_bus*: This protocol is not a specific bus type, but it can be used to communicate with a standard bus via a bus bridge. Several generated control signals ensure the different operations like single write, single read or burst transfer operations. More details about this standard can be found in [13]. For this thesis, this type is not necessary to use because the communication of the two designed modules will be realized using the other protocol types.

Figure 4.8 shows an example for different protocol types. The example deals with the computation of the average for a set of numbers. The numbers are stored in an array. First of all, the block gets *ap_ctrl_hs* as block-level interface. It generates automatically a clock, reset, start, ready, done and idle control signal. Thus, the module can be controlled with these signals by a processor. The input array of all numbers gets the interface protocol

ap_memory. It creates an address port to assign the data, which should be read and an enable signal to indicate if data is read. The output port, which delivers the average is assigned the protocol *ap_vld*. Therefore, a control signal is generated to symbolize if data is on the output port and if the data is valid. The input port X represents the number of samples for the average. It is defined the protocol *ap_none* and therefore no additional signals are added.

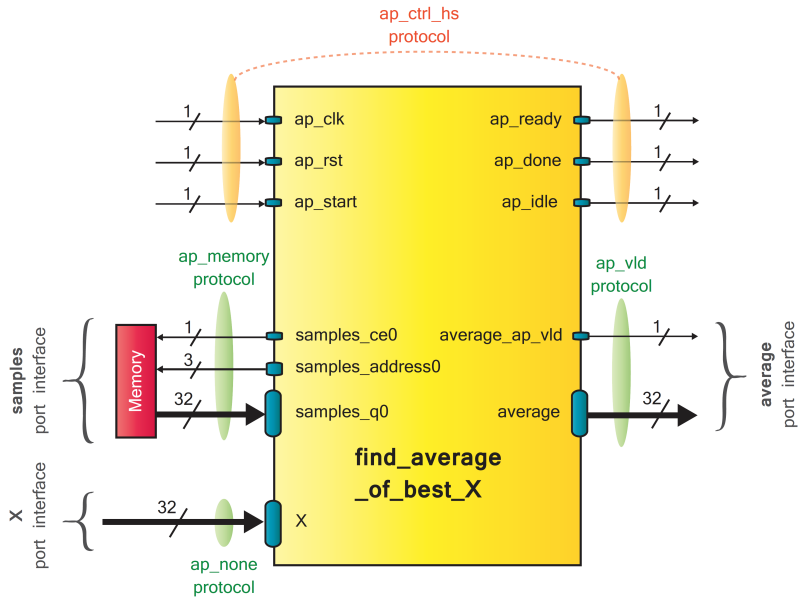


Figure 4.8: Example for different interface protocols and block-level protocol. [3]

4.4.5 Code Optimization

Code optimizations are essential to get a high quality hardware design out of the HLS. These have to be assigned manually to the high-level model. For different programming constructs directives are available to optimize the code. The main code snippets for optimizing are arrays and loops. In addition, there are further directives, which are responsible for the whole behavior of the module. These are e.g. pipelining and dataflow optimization. The mentioned optimization types are explained in the following paragraphs. The adapting on a high-level module is depicted in chapter 5.

Arrays

Different types of directives for arrays are:

- *Array Map*: This directive merges several arrays to one array with a larger size. Thus, less control signals, BRAMs or FIFOs are necessary.
- *Array Partition*: An array is split into several smaller arrays. More control signals are necessary, but it is possible to read more data at the same clock cycle from an array.

- *Array Reshape*: An new array with less elements and increased data width is generated out of an array. Therefore, the array is split into smaller arrays and reshaped together.
- *Resource*: This directive allows to assign an array to a specific memory resource. Engineers can map different arrays to the same memory.

Pipelining

One opportunity to optimize a high-level model is pipelining. Pipelining enables a higher data throughput, because the function is split into stages and loading new data can be done in smaller intervals. Thus, the modules, which are implemented in chapter 5, are used in high-speed printing systems and such systems have to process a huge amount of data in short time. Therefore, it is necessary to optimize the high-level model according to achieve a large data throughput. For that reason, pipelining is an essential design step. So, the meaning of pipelining will be explained below.

Figure 4.9 shows a simple example with data dependencies. This example is used to be optimized by pipelining. As already mentioned, there are data dependencies. Thus, Op1 has to be finished before Op2, and Op2 has to be finished before Op3 can start. All three operations together are called processing stage. The computation of the whole

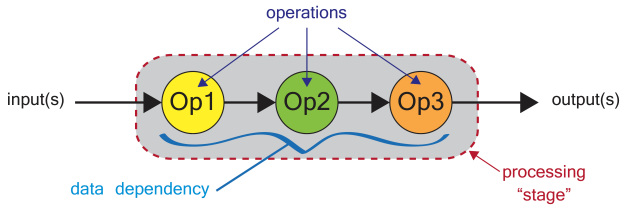


Figure 4.9: Example "Pipelining": Data dependencies between Op1, Op2, and Op3. [3]

processing stage requires one clock cycle. This leads to a data latency of one clock cycle. Figure 4.10, depicts the processing of data and their latency. Latency means the number of

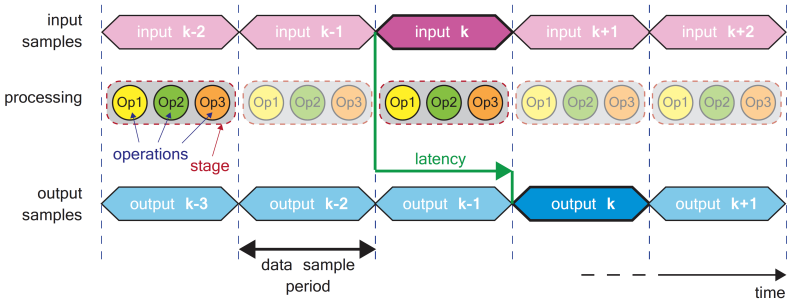


Figure 4.10: Example "Pipelining": Dataflow graph. [3]

necessary clock cycles to offer the corresponding data at the output. Since the operations are executed all in one clock cycle, the clock period is very high. Pipelining is used for the reduction of the clock period. Registers are added between the operations. Thus,

the whole function doesn't consist of one processing stage, but every operation represents an own processing stage. Every processing stage requires a clock cycle and since the

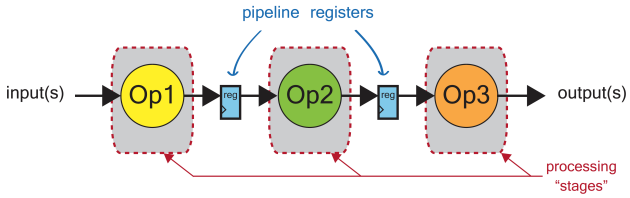


Figure 4.11: Example "Pipelining": Adding registers. [3]

complexity (number of operations) of a processing stage is reduced, the clock period can be decreased. Figure 4.12 depicts the data flow graph of the resulting pipelined function. Data can be already loaded after the completion of processing stage "Op1". As one can see in this figure, the clock period is shorter, but the latency (measured in clock cycles) is increased. Overall the time during loading data and providing it on the output equals to the example in figure 4.10. Using this directive leads to the mentioned increase of the

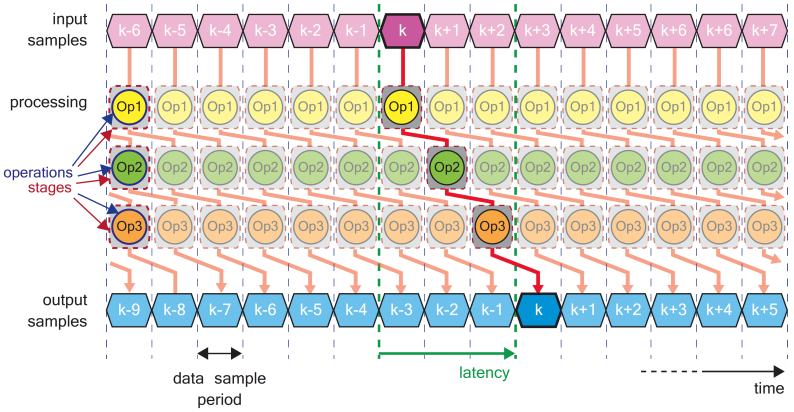


Figure 4.12: Example "Pipelining": Dataflow graph of pipelined stages. [3]

data throughput. It cannot only be used for whole functions, but also for e.g. single loops of a model. More information are provided in the following paragraph.

Loops

Optimization of loops comprises three types. These are called loop merging, loop unrolling and loop pipelining, which was mentioned before. This section presents a short example for each type to understand the utilization. [3]

Merging of loops of a function leads to a reduction of the latency. The example in listing 4.10 includes a loop for the addition of variables and a loop for the multiplication of the same variables. Assuming that an addition takes one clock cycle and a multiplication two clock cycles, this example requires 12 clock cycles for loop *add_loop* and 24 clock cycles for loop *mult_loop*. Additionally, entering and closing the loops need four clock cycles. Overall 40 clock cycles are necessary without any loop optimization.

```
void add_mult (short c[12], short m[12], short a[12], short b[12]){
  short j;

  add_loop: for (j=0;j<12;j++) {
    c[j] = a[j] + b[j];
  }

  mult_loop: for (j=0;j<12;j++) {
    m[j] = a[j] * b[j];
  }
}
```

Listing 4.10: Example: Merging loops (no merging).

By using loop merging, the two loops are combined together, because there are no data dependencies. The required number of clock cycles for additions are avoided. The addition takes place simultaneously with the multiplication. So, overall only 28 clock cycles are necessary for the completion of this example.

```
void add_mult (short c[12], short m[12], short a[12], short b[12]){
  short j;

  add_mult_loop: for (j=0;j<12;j++) {
    c[j] = a[j] + b[j];
    m[j] = a[j] * b[j];
  }
}
```

Listing 4.11: Example: Merging loops (merged loops).

The addition of two matrices is done by iterating over every element in a matrix. For two dimensional arrays two for loops are necessary. Since the effort and latency increases enormously without any optimizations, it is possible to **flatten the loops**. This means that more resources are allocated to calculate the individual additions simultaneously. The order of flattening can be adjusted manually. The whole addition can be executed in one single clock cycle or it is possible to flatten only the inner loop. In this way a hybrid solution can be created according to the number of utilized resources and latency.

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{01} & c_{11} & c_{12} & c_{13} \\ c_{02} & c_{21} & c_{22} & c_{23} \\ c_{03} & c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{01} & a_{11} & a_{12} & a_{13} \\ a_{02} & a_{21} & a_{22} & a_{23} \\ a_{03} & a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{01} & b_{11} & b_{12} & b_{13} \\ b_{02} & b_{21} & b_{22} & b_{23} \\ b_{03} & b_{31} & b_{32} & b_{33} \end{pmatrix} \tag{4.1}$$

The third type of loop optimization is **Loop Pipelining**. Generally, a loop iteration starts after the previous loop iteration is finished. Sometimes, many operations are done during an iteration. In such cases, it is useful to add loop pipelining, because the data throughput can be increased and the computation of the loop doesn't lead to be a bottleneck of the whole module. A short description of general pipelining is already noted in this thesis, whereas no further definitions are added in this section.

Dataflow

The optimization directive called *Dataflow* is similar to *Pipelining*. It is used to increase the data throughput. The difference is that it takes place on a higher level in the design process. Generally, a design contains different functions. In example of figure 4.13 there exists F1 (take order), F2 (food), F3 (coffee) and F4 (payment). Dataflow optimization leads to a reduction of the latency. As depicted in the example, using only one resource

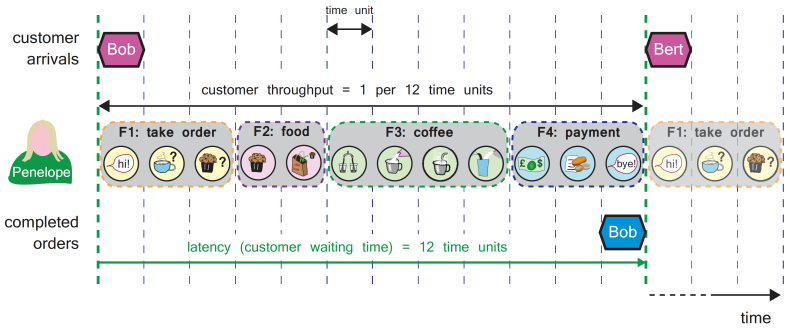


Figure 4.13: Example for dataflow optimization: Without dataflow optimization. [3]

(called Penelope) would take 12 time units to finish the execution. First, she has to take order, which requires three time units. After that, the food (two time units) and the coffee (four time units) are prepared and finally, she has to cash the customer. Then she can serve the next person. Dataflow optimization does a reduction of the overall latency. Therefore, more resources are used to execute all tasks.

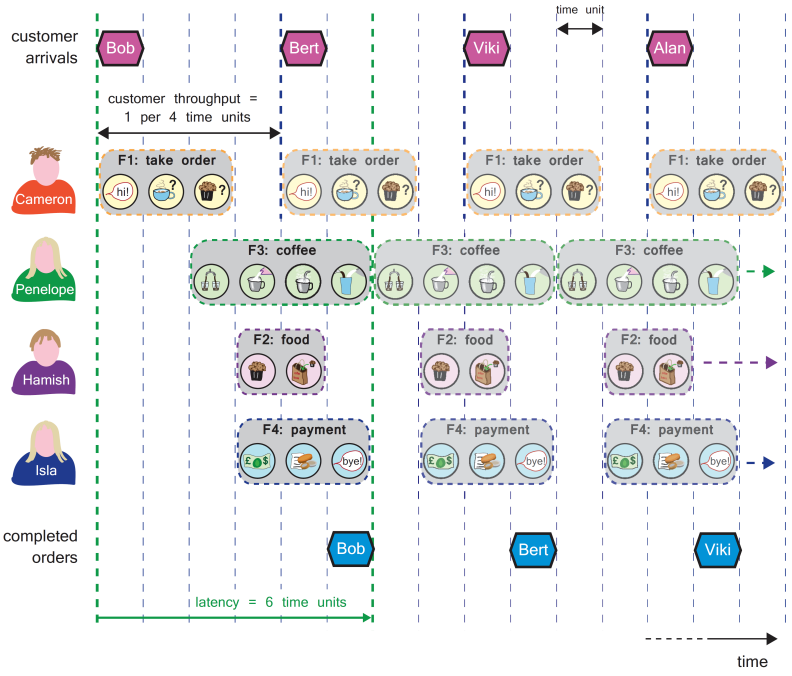


Figure 4.14: Example for dataflow optimization: With dataflow optimization. [3]

Now, Penelope gets three assistants called Cameron, Hamish and Isla. Each of them handles one task. So, every four time units a new customer can be served. However, the data dependencies have to be taken into account. The task of preparing a coffee cannot start before ordering the type of coffee. Preparing the food can just start after ordering the kind of food, too. These data dependencies are considered automatically by the HLS tool. So, engineers have the opportunity to assign manually this directive to increase the data throughput. Everything else is done automatically by the synthesis process.

4.4.6 Arbitrary Datatypes

A further important part of the design process is the choice of the bit width for variables. In hardware development, standard bit widths are not very suitable, but they are available. Table 4.3 shows the standard size in Vivado HLS. Generally, the size is not fixed in C programming definition, so engineers have to take care about the selection of standard types. Furthermore, Vivado HLS provides arbitrary precision integer types. Therefore,

Type	**Number of Bits**	**Range**
char	8	-128 to 127
short int	16	-32,768 to 32,767
int	32	-2,147,483,648 to 2,147,483,647
long int	32	-2,147,483,648 to 2,147,483,647
long long int	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Table 4.3: Standard Datatypes. [3]

two header files are availabe. One can be used for the programming of the high-level model in C and is called "*ap_cint.h*" and the other called "*ap_int.h*" is used for C++ models. They provide unsigned and signed data types. The following table 4.4 shows the usage of this arbitrary precision integer types. In addition, also fixed point and floating point

Language	**Integer Data Type**	**Description**	**Required Header**
C	int$N$ (e.g. int7)	signed integer of N bits precision	#include "ap_cint.h"
	uint$N$ (e.g. uint7)	unsigned integer of N bits precision	
C++	ap_int$\langle N \rangle$ (e.g. ap_int$\langle 7 \rangle$)	signed integer of N bits precision	#include "ap_int.h"
	ap_uint$\langle N \rangle$ (e.g. ap_uint$\langle 7 \rangle$)	unsigned integer of N bits precision	

Table 4.4: Arbitrary Datatypes. [3]

arbitrary precision datatypes are available. These are not necessary for the implementation of the modules in chapter 5. Information is available in [3].

4.5 Module: "Data flow"

This section describes the design process of the protocol interface, which is used as the module called "data flow" in the further progress of this thesis. Thereby, the usage in large industry printing systems is explained as well as the basic behavior of this module. In the corresponding section of chapter 5, tighter implementation definitions are presented.

4.5.1 Application field of this module

Generally, a printing system consists of several print heads and one control electronic is necessary for each eight print heads, i.e. the control electronic receives the printing data and processes them to actuate the print heads for achieving the correct behavior. Finally, the desired image should be printed. The GTP (Gigabit Transceiver) ring is used for the distributing of the print data. So each control electronic has implemented a GTP interface, to collect the corresponding data from the GTP ring. By the way, the printing data are already rehashed. So the control electronic gets the information and data for each print head instead of the pure image data.

Figure 4.15 shows the block diagram of the GTP interface of the control electronic. The block *ax7_GTP* is the common interface, which is provided by the producer or developer of the FPGA chip. In this thesis, a FPGA evaluation kit from Xilinx Inc. is used. The abbreviation *ax7* stands for for the FPGA family called Artix 7. Chapter 4.7 presents more information about the evaluation kit choice. This block processes the input data and delivers the receiving data and the receiving *charisk* data word. Thereafter, the *Alignment FIFO* buffers the input data, which is in further progress provided to the protocol interface. This part of the overall GTP interface is implemented in this master thesis in two different ways. As already mentioned, one implementation is done with the hardware description language VHDL and for the second implementation HLS is used as design tool. This module analyses the incoming data from the fifo buffer and determines, if the data are print data (p_data_s) or configuration data (c_data_o, c_data_flag_o). Print data is forwarded to the corresponding print head. Configuration data is used to change the behavior of the control electronic. If the protocol interface recognizes the appearance of configuration data in the receiving data stream and in the receiving *charisk* port, the configuration block will be changed according to the new incoming data. In further chapters, this module is described with the label "data flow". The reason is, that no complex algorithm is implemented within this module. It is only used to detect some keywords in the input buffer of the *charisk* stream and it has to forward the incoming data to the corresponding output port. No mathematical complexity like signal processing, image processing, or cryptography algorithms are implemented in this module.

4.5.2 Module Interface

During the design process, deliberations have to be done according to the number of interface ports. Figure 4.16 depicts all necessary ports. On the left-hand side the input ports are illustrated and on the right-hand side the output ports are shown. These are pooled into five groups. The *RX stream* includes the receiving data and the receiver for the *charisk* data word. The *configuration* consists of some signals, which control the protocol interface. *Status* is conducive to get some information about the actual state. *Tx stream*

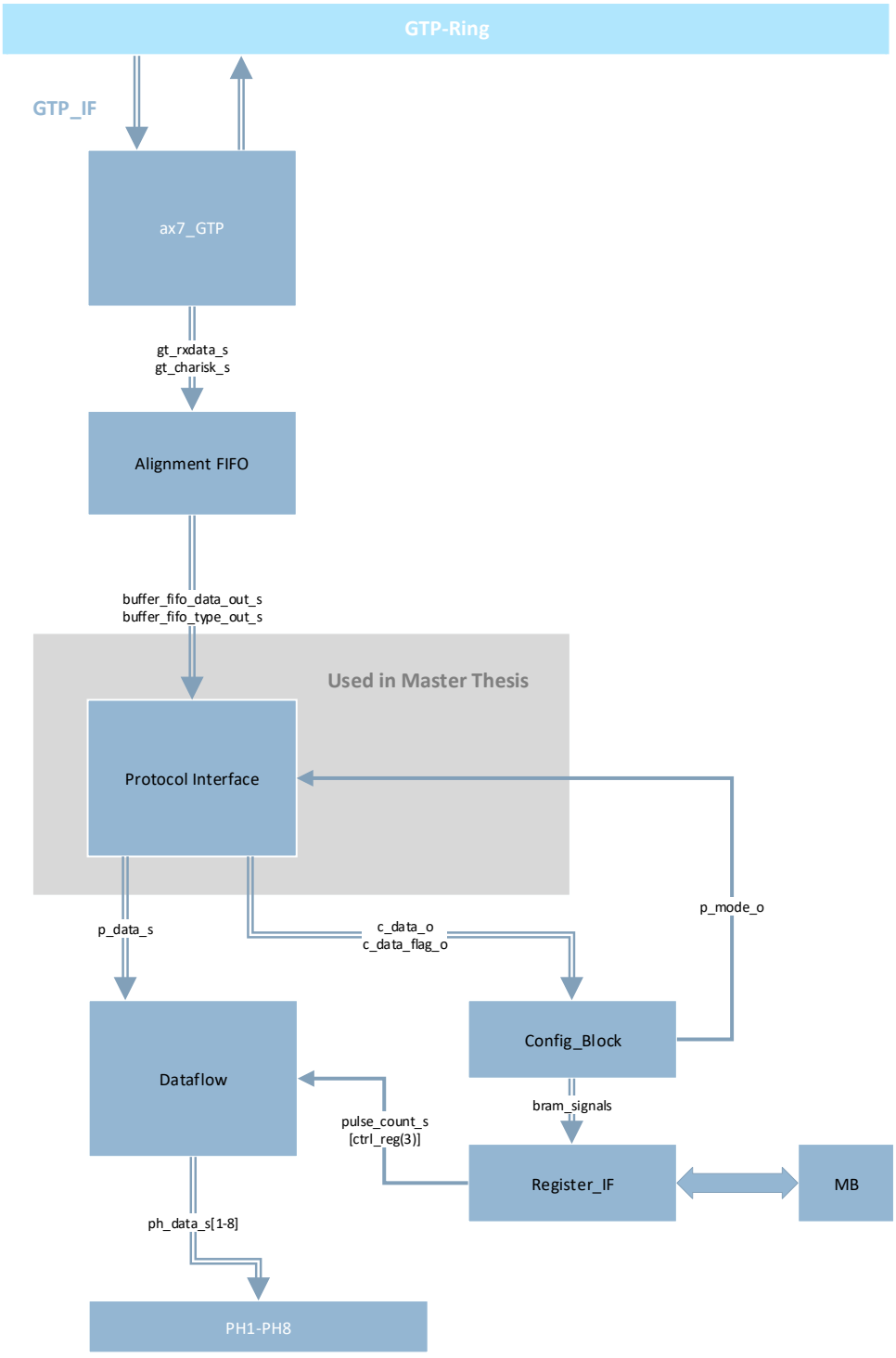


Figure 4.15: GTP Interface (data flow block diagram).

is the contrary to *Rx stream*. The principle, wherefore each control electronic needs a receiver and a transmitter, is explained in the following. The GTP interface is connected to a GTP ring. Therefore all the data is received by the first control electronic. Afterwards, for closing the GTP ring this electronic has to forward the data to the next GTP interface. Therefore, a *tx stream* part of this module is always necessary. Furthermore, *extracted data* includes the output ports for either print data or configuration data.

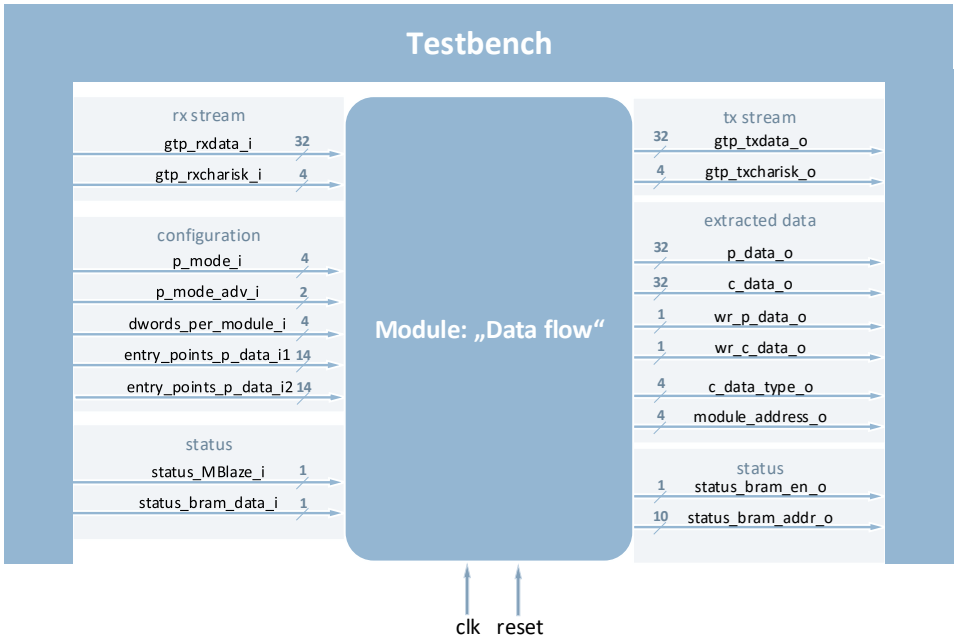


Figure 4.16: Module: "Data Flow" interface.

The verification of the implemented module is done by a testbench. It can be implemented with hardware description languages like VHDL or by using HLS. In this thesis, the design methodology for the testbench is HLS. So the behavior is implemented in C/C++ and afterwards it is synthesized automatically into a VHDL testbench. The implemented modules, either the conventional module with VHDL or the generated module by HLS, is connected with the testbench. Thus, these modules can be verified simultaneously and their output can be compared. More details about the implementation and about the verification is presented in the chapter 5. There, a comparison between the two modules is listed according to the utilization of resources, power consumption, etc.

4.6 Module: "Cryptography-Algorithm"

The second module for the comparison between HDL designs and HLS generated modules is a cryptography module. The preprocessing of images for the industrial printing systems requires a huge computational effort for the workstation of each system. Since the quality of images is increasing and so the computational complexity also raises, the

change of using FPGA boards for the preprocessing could be inevitable. Therefore, the module "Cryptography-Algorithm" presents the quality of using HLS versus the conventional design methodology using VHDL for CPU-intensive tasks. In addition, it is also necessary to get a knowledge of the design speed using HLS, because the best way is that already used algorithms for the preprocessing can be reused, partly modified and finally implemented on a FPGA. In this master thesis, a cryptography algorithm is used to show the design differences between these two design methods.

4.6.1 Principle of Ascon

The algorithm implemented in the "Cryptography-Algorithm" module is called Ascon and has been designed at Institute for Applied Information Processing and Communications of the University of Technology Graz for the current CAESAR competition. The CAESAR competition tends to develop a new standard, which includes all security aspects to become a very secure algorithm. Over the years, different types of competitions were started. The team, developing Ascon, is taking part of the competition for a new advanced encryption standard (AES). In the following paragraphs the principle of Ascon is described.

Encryption Procedure

Generally, the encryption and decryption procedure of Ascon is very similar. First, the encryption (figure 4.17) includes a state register with a data width of 320 bits, which includes a 64 bit IV data word (0x80400c0600000000), the 128 bit width key, and the 128 bit width nonce. Afterwards the round function is executed a times. A description about the round function follows after the decryption part. Last task of the initialization is a XOR operation of the key and the last 128 bits of the state register. Next part is the processing of the associated data. Therefor, the first 64 bits of the state and the first 64 bits of the associated data stream are executed by a XOR operation. Then the round function is run b times. Afterwards a new block of associated data is loaded. This is done until the last block of associated data (maximal 64 bits) is processed. Last part of this task is a XOR operation of the last bit of the state with the value 0x1. Thereafter, the main encryption part takes place. 64 bit width data words are loaded and a XOR execution is done with the first 64 bits of the state register. The result appears on the 64 bit width output data port (figure 4.20) and equates to the cipher text of the first 64 bits of plaintext. Next step is the execution of b times the round function. This part continues until the whole plaintext is encrypted. Keep in mind, that no execution of the round function follows to the last 64 bits of plaintext. Finally, a tag is generated in the finalization part of the encryption process. First task is a XOR operation of bits 65 to 192 from the state register and the key. Afterwards, the round function is executed again a times. Before the tag appears on the output port, the last 128 bits of the state and the key are loaded into the XOR operation again.

Decryption Procedure

The decryption procedure (figure 4.18) differs only in a few tasks. The initialization and the processing of the associated data have no distinctions. The main decryption part starts with a XOR operation of the first 64 bits from the state register and the 64 bit

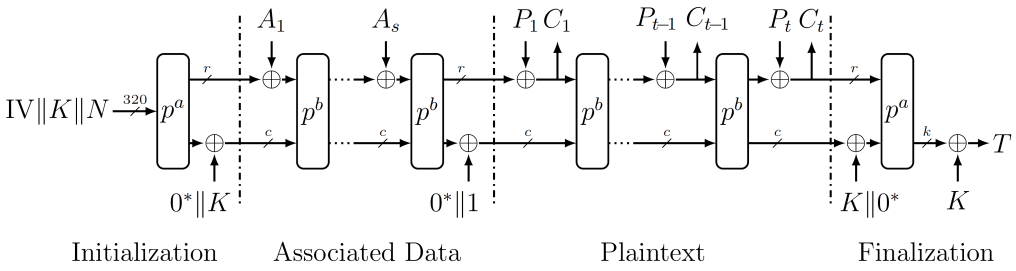


Figure 4.17: Block diagram of encryption. [4]

width cipher text block. The result appears at the output port and is the plain text of the corresponding cipher text. In addition, the first 64 bits of the state register are replaced by the cipher text. Afterwards, the round function is executed and then the same task is done again. This is the only part, which differs to the encryption procedure.

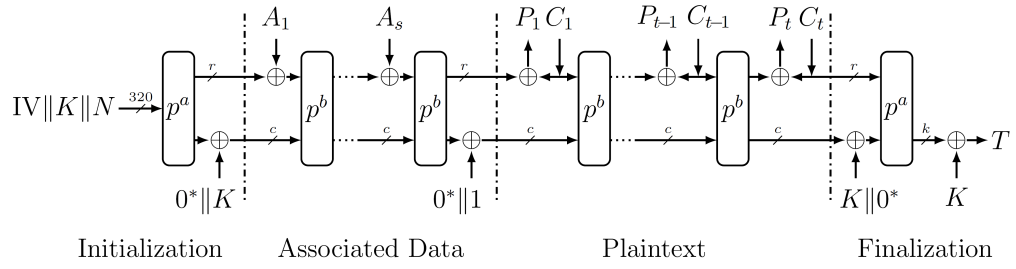


Figure 4.18: Block diagram of decryption. [4]

Round Function

The round function is one of the main parts of the encryption and decryption procedure. It consists of three tasks: linear diffusion layer, substitution layer and addition of constants.

The first task is the **linear diffusion layer**. The state register is split into five individual registers x_0 to x_4. A mathematical function including shift and XOR operations is applied on each register. The function for each register is listed in the following:

$$x_0 = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \tag{4.2}$$
$$x_1 = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \tag{4.3}$$
$$x_2 = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \tag{4.4}$$
$$x_3 = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \tag{4.5}$$
$$x_4 = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \tag{4.6}$$

Generally, the linear diffusion layer shifts the bits within the individual registers partitioned of the whole state register.

The second task of the round function is the **substitution layer**. This calculation is applied bit-slice. This means, that one bit is taken out of each individual register. So five

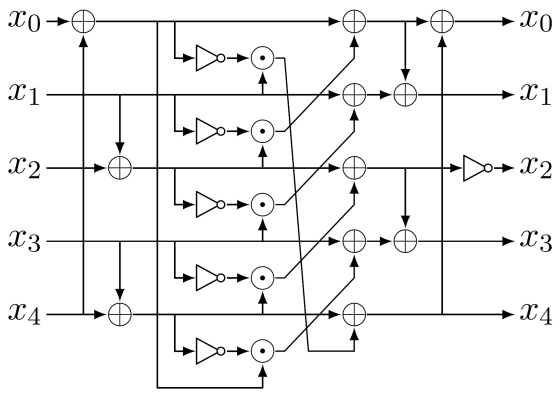


Figure 4.19: Process of substitution layer. [4]

bits conform one bit-slice. The mathematical substitution process is shown in figure 4.19. It consists of some XOR, AND, and NOT operations.

The last task is the **addition of constants**. For every round number a different constant is used. The addition means a XOR operation of the third individual register of the state register and the corresponding constant. The different constants are listed in the following table. More information about the algorithm and the security features of

p^{12}	p^8	p^6	Constant	p^{12}	p^8	p^6	Constant
0			00000000000000000f0	6	2	0	0000000000000000096
1			00000000000000000e1	7	3	1	0000000000000000087
2			00000000000000000d2	8	4	2	0000000000000000078
3			00000000000000000c3	9	5	3	0000000000000000069
4	0		00000000000000000b4	10	6	4	000000000000000005a
5	1		00000000000000000a5	11	7	5	000000000000000004b

Table 4.5: Addition of Constants. [4]

Ascon can be found in [4]. Additionally, the input data blocks like associated data and the plain text or cipher text are preprocessed. This means that the last data block has to be padded, but since this is part of the controlling processor of this module, no further details are described in this thesis. The description about padding can also be found in [4].

4.6.2 Module Interface

The definition of the interface ports of the module is a basic deliberation during the design process. Figure 4.20 depicts the input and output ports of the module "cryptography-algorithm". The input data port and output data port are defined as 64 bit width ports to achieve a high throughput design with consideration of a sensible amount of allocated area and utilized resources. Increasing the bit width of this ports would also raise the throughput, but the quality of this design is sufficient.

As already mentioned, the module includes an input data port and an output data

port. The input port is responsible for loading new data blocks like the associated data, plaintext, cipher text, the key, which has to be stored in the module during the processing, and the nonce loaded into the state register. The output port is used to write out the generated cipher text, or in case of decryption the plaintext as well as the tag, if the encryption or decryption is finished. In addition, the module have got some control signals: *start*, *instruction* and *length_last* signal. The *start* signal is used to initiate the encryption or decryption procedure. By using the *instruction* signal the processor, which is connected to the cryptography-algorithm module, can control this module. The different types of instructions is presented in chapter 5. The third port of the control signals is called *length_last*. It is used to indicate the last input data block and to indicate the length of the last block in bits (maximal value is 64), which is necessary to write out the correct corresponding cipher text or plain text. In addition, also three status signals are necessary to indicate the validation of the output data, the finishing of the instruction and the indication of the tag, whereby the port *out_is_tag_o* is two bit width, because the tag has 128 bits and the output data port has a data width of 64. So the tag has to be written out in two clock cycles and this port marks the least significant bits or the most significant bits. More information about the exact implementation is provided in chapter 5.

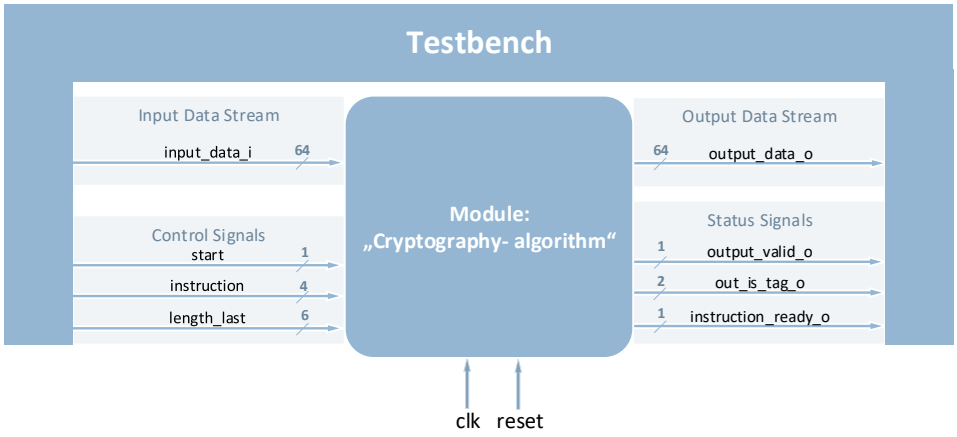


Figure 4.20: Module: "Cryptography-Algorithm" interface.

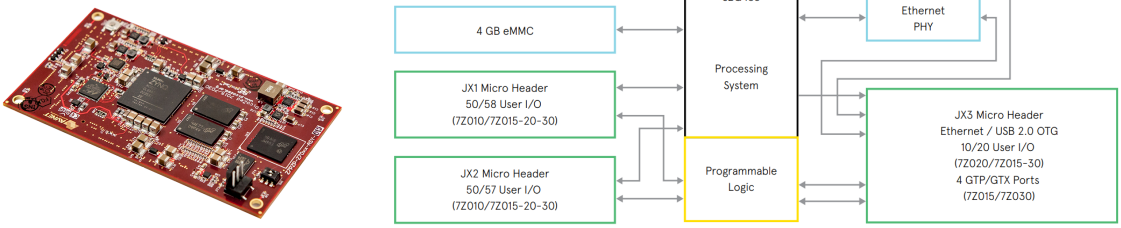
For the verification of this module a testbench is designed. It is written in C/C++ and transformed into a HDL testbench by using HLS.

4.7 Choice of the Evaluation Kit

The designed module have to be implemented on an evaluation kit, to get a precise report about the quality and a comparison of the HDL modules and the HLS generated implementations. Criteria according to the choice of the evaluation kit is the amount of available resources. This number has to correspond with the implemented designs. Since this thesis is a cooperation with the company *Durst Phototechnik Digital Technology GmbH* and they are using more often system-on-chip boards for their implementations, using an

evaluation board including a system-on-chip for this thesis is a sensible choice. So the benefits and drawbacks of using HLS in the design process can be shown directly by using a system-on-chip.

A very good and powerful evaluation kit is developed by Avnet, Inc. It is called PicoZed (figure 4.21), whereby different versions are available and for this thesis version 7Z015 is used. This board includes a system-on-chip called Zynq7015, which consists of a processing system and a programmable logic. The processing system (PS) can be utilized as a processor and the programmable logic (PL) corresponds to a simple FPGA, whereby this part is used for the implementation of the described modules. For the sake of completeness all further features a presented briefly. However, these are not necessary for the implementation. The PicoZed 7Z015 includes 148 user I/O. 135 pins are connected to the PL part and 13 pins to PS. These 148 user I/O pins are grouped into three connectors (JX1 - JX3) on the backside of this board. These connectors are responsible for the access of the ethernet, USB and JTAG interface as well as power and other control signals. Additionally, a 128MB quad serial peripheral interface (QSPI), 1GB DDR3 RAM, an interface for a 4GB embedded multi media card (eMMC), and an oscillator of 33.33 MHz are available. The mentioned three connectors JX1 to JX3 are used for the connection of the PicoZed 7Z015 with the corresponding PicoZed FMC Carrier Card V2.



a) PicoZed 7Z015 b) Block diagram of PicoZed

Figure 4.21: PicoZed. [11]

Figure 4.22 depicts a) the PicoZed FMC Carrier Card V2 and b) the corresponding block diagram. The FMC Carrier Card V2 provides all interfaces, which can be used by the PicoZed. For this thesis the JTAG interface and the USB-UART connection is necessary, whereby the JTAG is used to flash the system-on-chip and the USB-UART connection is used for debugging. Further features are listed below:

Memory

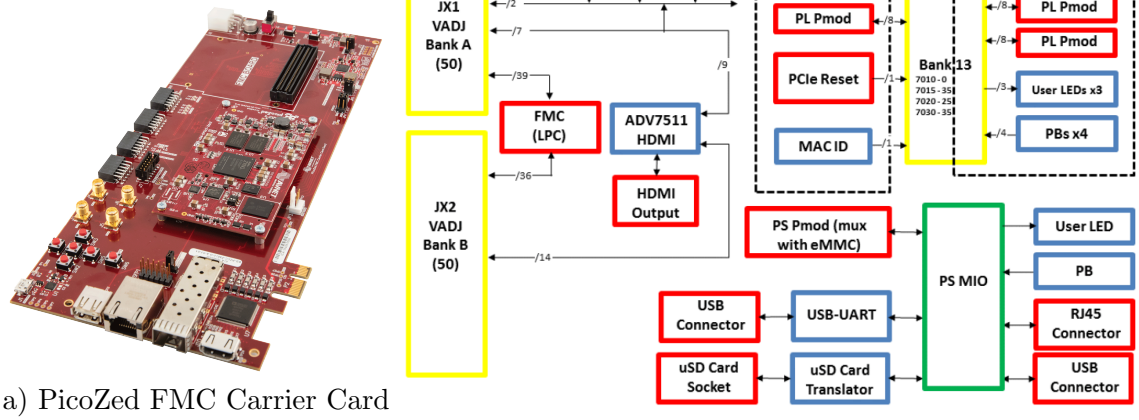
- MicroSD card socket with 8GB microSD Card
- Clock Synthesizer configuration EEPROM
- I2C MAC ID EEPROM
- 1-wire MAC ID EEPROM

Communication

- x1 PCIe Gen 2
- SFP+
- SMA port for GTX/GTP
- 10/100/1000 Ethernet connector
- USB 2.0 Type A connector
- USB UART

User I/O

- FMC (Low Pin Count)
- PS Pmod (Shared with SOM eMMC)
- PL Pmods #1,#2 (7015/20/30 only)
- PL Pmod #3 (7015/30 only)
- Software
- PetaLinux
- Wind River Pulsar Linux pre-loaded on microSD card



a) PicoZed FMC Carrier Card V2

b) Block Diagram

Figure 4.22: Picozed FMC. [11]

Chapter 5

Implementation and Results

This chapter deals with implementation details of the two different modules and their different design methods, whereby only parts of the implementation code are presented. The beginning of this chapter starts with the description of the module "data flow". It is split into the design on RTL and the HLS generated design. Afterwards, the "cryptography-algorithm" module is presented, whereby the two different design methods are covered individually. Finally, the comparison of the quality, power demand, design effort, etc. are published in this thesis.

5.1 Used Tools for Module Generation

Figure 5.1 shows the used tools for the implementation of the RTL model. The VHDL code is written by using Vivado Design Suite. This tool provides all steps for the generation of the bitstream, which is loaded onto the evaluation kit.

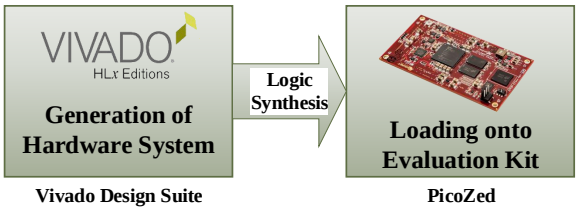


Figure 5.1: Used tools for the implementation - register-transfer level.

The design process of the HLS designed module includes one further tool. The high-level model is implemented in C/C++ by using Vivado HLS from Xilinx Inc. This tool enables the automatic synthesis to RTL. The further steps are equal to the design flow of the modeling on RTL. The output of the HLS is loaded into Xilinx Vivado Design Suite. Afterwards, the bitstream for the implementation on the evaluation kit is generated.

Figure 5.2: Used tools for the implementation - high-level synthesis.

5.2 Implementation of Module: "data flow"

The hardware module called "protocol interface", which is also mentioned as the "data flow" module in this thesis, is used to filter the incoming data stream. According to the received K-words on the input port called *rx_data*, the following data words are indicated as print data or configuration data. K-words have an 8-bit and a 10-bit representation and are used to recognize the clock out of the receiving data stream, because they achieve a DC-balance. So this is very useful for serial data transmissions. Additionally, they are usable to mark the data stream, because they are preceded to the data stream and in this way the receiver knows the type of the following data words. The table below includes all K-words of this module.

Code Name	8-bit representation	10-bit representation	hex representation
K23_7	11101 111	111010 1000	0xF7
K28_5	00111 101	001111 1010	0xBC
K28_6	00111 011	001111 0110	0xDC
K29_7	10111 111	101110 1000	0xFD
K30_7	01111 111	011110 1000	0xFE

Table 5.1: K-words abbreviations.

Generally, the "data flow" module consists of three state diagrams: Pre-Processing, Processing of Printing Data and Processing of Configuration Data. These are shown in figure 5.3. A detailed description of each state machine is done in the following paragraphs.

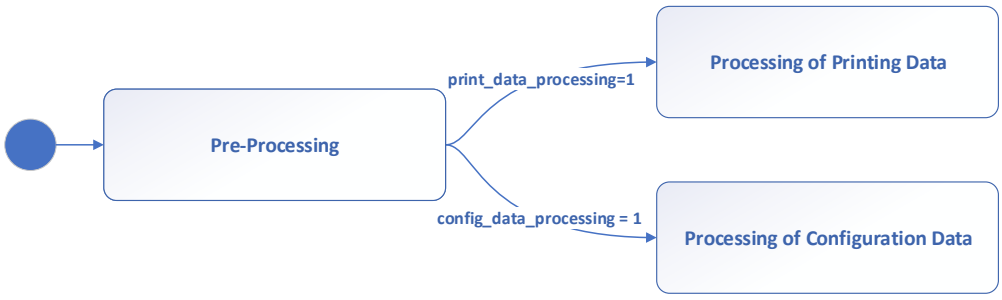


Figure 5.3: Implementation of module: "data flow".

The first state diagram depicts the pre-processing of the incoming data (figure 5.4). If the K-word called *K28_5* appears in the data stream, the following data equates to printing data. Otherwise, it is configuration data. Beyond the data receiving stream *rx_data*, the signal *rx_charisk* has to be High to indicate a K-word, because in some cases data words look like K-words. Regarding the type of the K-word, the signal *print_data_processing* or *config_data_processing* is set to logic one. As in figure 5.4 shown, the pre-processing state diagram consists of three states. The state *uninit* sets the control signals to zero. The idle state verifies the *rx_data* and the *rx_charisk* stream and sets the signal *print_data_processing* or *config_data_processing*. Finally the *set_flag* state resets the two control signals to zero. The whole implementation includes some additionally signals to get no errors during execution in this module, but the basic principle is explained above.

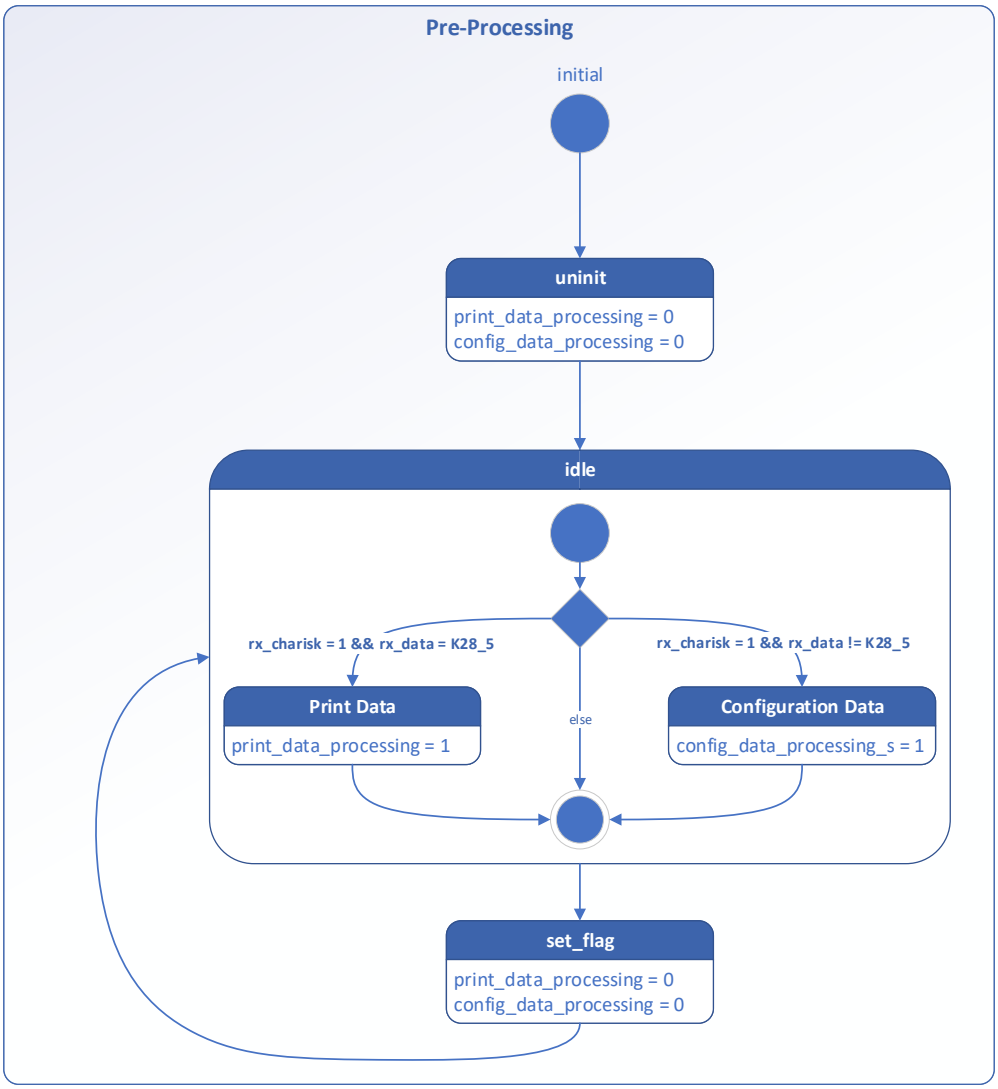


Figure 5.4: State diagram of the pre-processing.

The processing of the printing data is a further state machine in this module. It has two states: *idle* and *extract_data*. The default state is *idle*. If the signal *print_data_processing* is high, the state is changed to *extract_data*. So, the following receiving data is forwarded as print data to the next module (figure 4.15). The next change of state takes place after receiving "1111" on the *rx_charisk* port.

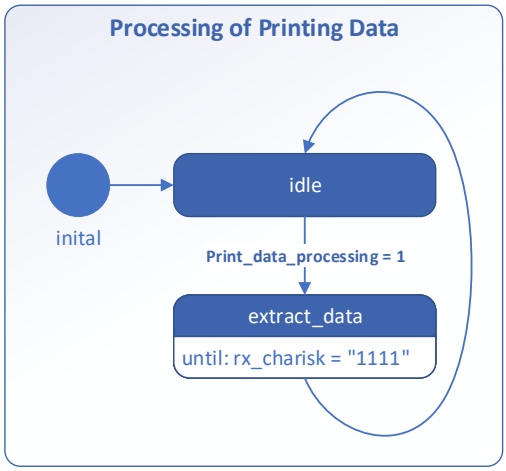


Figure 5.5: State diagram of the print data processing.

The third state machine is responsible for the processing of the configuration data and it is active after setting signal *config_data_processing* to logic one. This state machine consists of five states: *idle*, *addressing*, *check_address*, *extract_data* and *status_update*. Generally, the default state is *idle*. No special operations are executed in this state. If the K-word K29_7 is received, the state machine changes to *addressing* mode. This state assigns an address to the corresponding GTP interface and sends the next address via the transmitter ports (*tx_data* and *tx_charisk*) to the GTP ring respectively to the next GTP interface. This procedure enables the identification of each interface by an unique address. If the receiving data doesn't equate to the K-word K29_7, but either K28_6, K30_7 or K23_7, the next state is *check_address*. The receiving data is forwarded to the corresponding transmitter ports to send the same data to the next GTP interface on the GTP ring as well. In addition, this state also checks if the K-word equates to K23_7. In this case, the next state is called *status_update*. The state *status_update* is used to get information from the individual GTP interfaces, e.g. the assigned module address or firmware version number. Otherwise the data is extracted and forwarded as configuration data on the corresponding output ports of this module, which are connected to the following module as shown in figure 4.15.

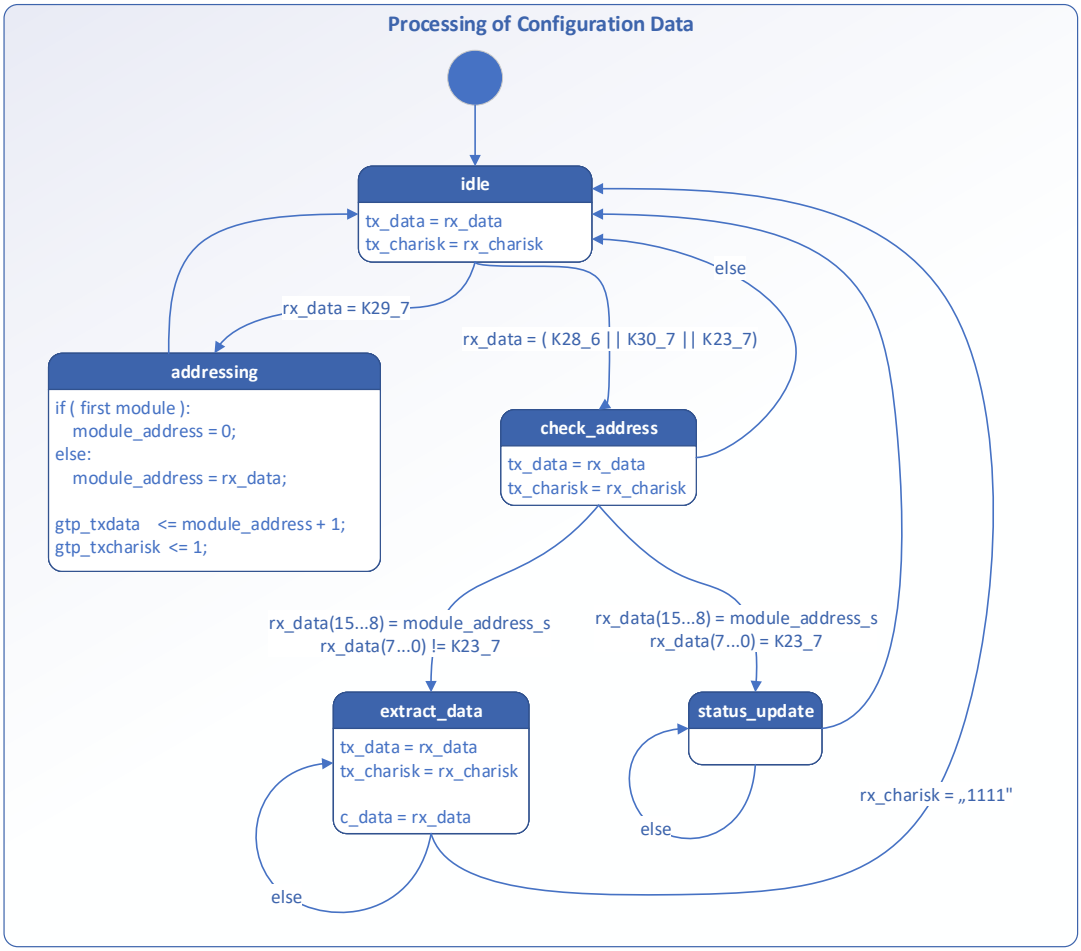


Figure 5.6: State diagram of the configuration data processing.

As already mentioned, this module is implemented in two different design methodologies, which are compared in this thesis. Therefore, an useful test system was designed, which is shown in figure 5.7. It consists of eight modules for the comparison of the two modules called ”data flow”, because they have to set all outputs identically to ensure the correct behavior at every clock cycle. In addition, the two modules are connected with these comparator modules and an automatically generated (by Xilinx Vivado) reset block is added. This test system is uploaded on the described evaluation kit. More details about the results will be presented in the chapter 5.4.

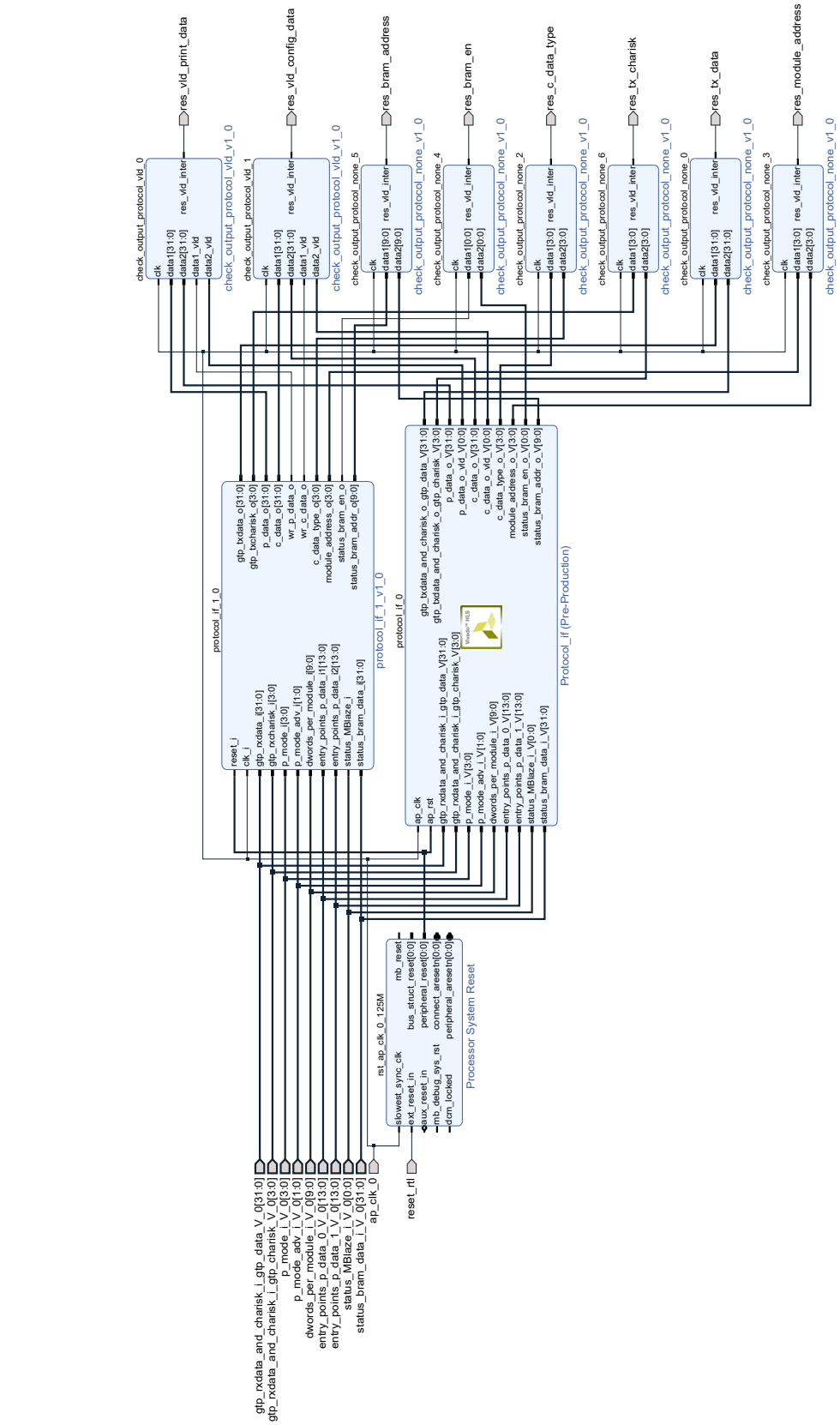


Figure 5.7: Implementation of module "data flow".

5.2.1 RTL Implementation

The entity of the module "data flow" designed on RTL is shown in listing 5.1. Generally, every hardware module with memory elements includes a reset and a clock signal. Firstnamed is used to reset the module to the default state. Further on, the data words are received on the receiver (rx) stream and sent to the next GTP interface via the transmitter (tx) stream. The data port width is 32 bits and the *rx_charisk* port is 4 bits width, whereby every bit refers to each eight bits of the data port. Principally, the GTP ring is a serial transmission. After 32 bits are received, they are forwarded over the *gtp_rxdata* port to the module. So the data processing is done by a 32 bit architecture.

The group of input ports called configuration is used to control this module. These signals are connected all to the configuration module (in figure 4.15 only the signal *p_mode_i* is depicted). The stored configuration data set the value of these signals and the module "data flow" is controlled by them. The signal *p_mode_i* and *p_mode_adv_i* is used to send information about a special mode to this module. The other three signals are used to inform the "data flow" module about the number of data words read from the GTP ring and processed in the module. *entry_points_p_data_iX* mark the first data word which should be forwarded to the next module and the signal *dwords_per_module_i* assigns the maximal number of data words. Additionally, these control signals are used for the processing of the printing data.

The group of signals called *extracted_data* includes output ports, which provide the processed data to the next modules. The first two signals are used to forward either the printing or the configuration data. *p_data_o* and *c_data_o* ports are used for forwarding of the received data. Additionally, the signal *c_data_type_o* indicate the type of the configuration data, e.g. live update of configuration data, whole configuration data file, status request, etc. Finally, the output port *module_address_o* is used for forwarding the information about actual GTP interface address. The bit width of this port is fixed to 4 bits, but it depends on the actual number of GTP interfaces on the GTP ring.

The last signal group is called *status* and it includes all signals to get information about the state of the *Config-Block* in figure 4.15. These are all necessary control signals to read out of a BRAM, i.e. an *address*, *data* and *enable* signal and a signal to indicate the availability of the BRAM.

```
1  entity protocol_if_1 is
2  port(
3    reset_i                 : in  std_logic;
4    clk_i                   : in  std_logic;
5    ---------- rx stream ------------------------------------------------
6    gtp_rxdata_i          : in  std_logic_vector(31 downto 0);
7    gtp_rxcharisk_i         : in  std_logic_vector(3 downto 0);
8    ---------- tx stream ------------------------------------------------
9    gtp_txdata_o          : out std_logic_vector(31 downto 0);
10   gtp_txcharisk_o         : out std_logic_vector(3 downto 0);
11   ---------- configuration -------------------------------------------
12   p_mode_i                : in  std_logic_vector(3 downto 0);
13   p_mode_adv_i            : in  std_logic_vector(1 downto 0);
14   dwords_per_module_i     : in  std_logic_vector(9 downto 0);
15   entry_points_p_data_i1  : in  std_logic_vector(13 downto 0);
16   entry_points_p_data_i2  : in  std_logic_vector(13 downto 0);
17   ---------- extracted data ------------------------------------------
```

```
18   p_data_o                : out std_logic_vector(31 downto 0);
19   c_data_o                : out std_logic_vector(31 downto 0);
20   wr_p_data_o             : out std_logic;
21   wr_c_data_o             : out std_logic;
22   c_data_type_o           : out std_logic_vector(3 downto 0);
23   module_address_o        : out std_logic_vector(3 downto 0);
24   ---------- status ----------------------------------------------
25   status_MBlaze_i         : in  std_logic;
26   status_bram_en_o        : out std_logic;
27   status_bram_addr_o      : out std_logic_vector(9 downto 0);
28   status_bram_data_i      : in  std_logic_vector(31 downto 0)
29 );
30 end protocol_if_1;
```

Listing 5.1: VHDL-Code of module "data flow".

The basic principle about the behavior and the usage is already mentioned. Further on, in the next section, a description of the same module designed with HLS is presented.

5.2.2 HLS Implementation

The second design method is the generation of the RTL model by HLS. The whole coding is done in C/C++ and automatically transformed into VHDL code. This section describes the most interesting parts of the code. The transformation is done with Xilinx Vivado HLS.

The following listing 5.2 depicts the interface definition of the "data flow" module in C/C++. The number and type of input and output ports equates to all ports of the RTL model. As already mentioned, Xilinx Vivado HLS provides some arbitrary precision data types. The post-posed number indicates the width of this data type, e.g. *p_data_o* is an unsigned integer with 32 bits data width.

```
1  void protocol_if(
2    gtp_in_out_stream gtp_rxdata_and_charisk_i,
3    uint4 p_mode_i,
4    uint2 p_mode_adv_i,
5    uint10 dwords_per_module_i,
6    uint14 entry_points_p_data[2],
7    uint1 status_MBlaze_i,
8    uint32 status_bram_data_i,
9    gtp_in_out_stream* gtp_txdata_and_charisk_o,
10   uint32* p_data_o,
11   uint1 *p_data_o_vld,
12   uint32* c_data_o,
13   uint1 *c_data_o_vld,
14   uint4 *c_data_type_o,
15   uint4 *module_address_o,
16   uint1 *status_bram_en_o,
17   uint10 *status_bram_addr_o);
18
```

Listing 5.2: Parts from C-Code of module "data flow".

One important keyword, which is very often used in C/C++ implementations for HLS, is *static*. Generally, after finishing the execution of a function, all variables loss their information. However, the implementation of state machines requires the storage

of the current state until the next call of this function. *static*-defined variables solve this problem, because the data is not volatile anymore. In this implementation, the keyword is used for the state of the pre-processing, state of the print data processing, state of the configuration data processing and for the incoming data samples on the receiver stream.

```
1  static state_processing state;
2  ...
3  static state_print state_p;
4  ...
5  static state_print state_c;
6  ...
7
```

Listing 5.3: "static" usage for variables.

Further on, the synthesis process needs some additional information about the behavior of the module. VHDL enables the programming of simultaneously running tasks. However, HLS provides the assigning of some directives to achieve the same behavior. These are shown in the following listing. A description about the type of the directives is already presented in chapter 4.4. The mainly used optimization directives in listing 5.4 define the type of the interfaces, except the directive "pipeline" and "latency". Pipelining means, that data is loaded into the module and provided at the output ports at every clock cycle and the latency define the number of clock cycles while the first data is processed.

```
1  ############################################################
2  ## This file is generated automatically by Vivado HLS.
3  ## Please DO NOT edit it.
4  ## Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
5  ############################################################
6  set_directive_array_partition -type complete -dim 1 "protocol_if"
       entry_points_p_data
7  set_directive_interface -mode ap_ctrl_none "protocol_if"
8  set_directive_interface -mode ap_none "protocol_if" p_data_o
9  set_directive_pipeline "protocol_if"
10 set_directive_interface -mode ap_none "protocol_if" c_data_o
11 set_directive_interface -mode ap_none "protocol_if" status_bram_addr_o
12 set_directive_interface -mode ap_none "protocol_if" status_bram_en_o
13 set_directive_interface -mode ap_none "protocol_if" module_address_o
14 set_directive_interface -mode ap_none "protocol_if" c_data_type_o
15 set_directive_interface -mode ap_none "protocol_if" gtp_txdata_and_charisk_o
16 set_directive_interface -mode ap_none "protocol_if" p_data_o_vld
17 set_directive_interface -mode ap_none "protocol_if" c_data_o_vld
18 set_directive_latency -min 2 -max 2 "protocol_if"
```

Listing 5.4: Directives for the automatic synthesis process.

5.2.3 Testbench: Generated by HLS

A testbench is used for the verification of the modules designed by two different methods. Since the testbench is not part of a whole system, the quality respective amount of utilized resources and power demand is not relevant. Therefore, it can be designed on RTL or by HLS, whereby the designing in VHDL is generally very time consuming, because of the complexity of this programming language compared to a high-level language.

The testbench assigns a data word to each input port of the function or module at every clock cycle. Thus, the receiving stream includes different K-words followed by a different number of data words to check every single state of the implementation. In further consequence, the testbench is connected to the system of figure 5.7.

5.3 Implementation of Module: "cryptographic-algorithm"

The second module, which is designed for the comparison between the two design methods (design on RTL, HLS generated design), deals with the cryptographic algorithm called Ascon. Figure 5.8 shows the input and output ports of this module. This algorithm is already explained in chapter 4.6.

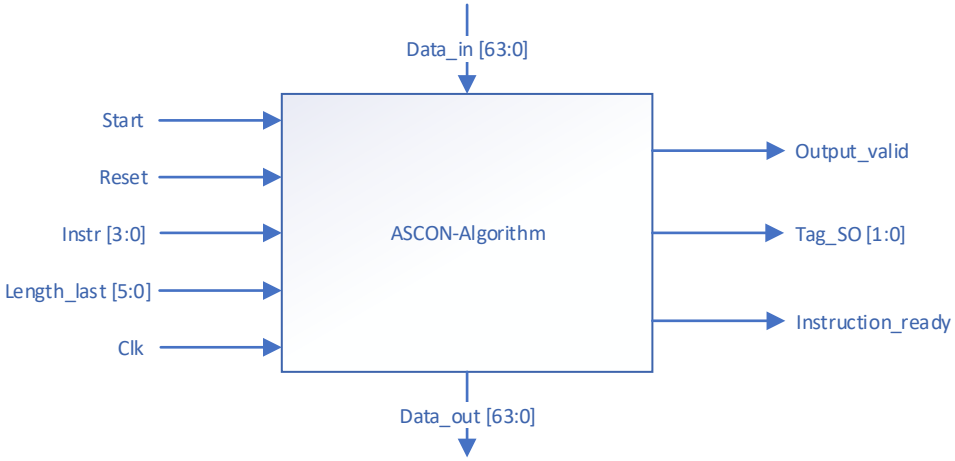


Figure 5.8: Module: "cryptography-algorithm".

The data flow graph of the implementation is depicted in figure 5.9. The data flow graph includes only the most important parts of the algorithm. This module is controlled via instructions. Altogether, twelve different instructions are available. All operations can be executed by them as described in chapter 4.6. The individual instructions are explained in the following paragraph. The input port *input_data* is used for the loading of key, nonce, associated data or plaintext/ciphertext. Port *output_data* provides the encrypted or decrypted text as well as the tag generated in the finalization. In addition, the module consists of different parts. The controller processes the incoming instruction and forwards the individual assignments to the other parts of the module. The state register includes five registers with a data width of 64 bits. Furthermore, the round function consists of three different parts: substitution layer, constant addition and shift operation layer. These are all implemented separately on RTL. Which part is active and what data are provided on the individual connections is determined by the controller. The last part is the XOR operation. It has a connection back to the state register and to the output port. According to the definition of the algorithm in chapter 4.6, the controller defines the data flow in this part.

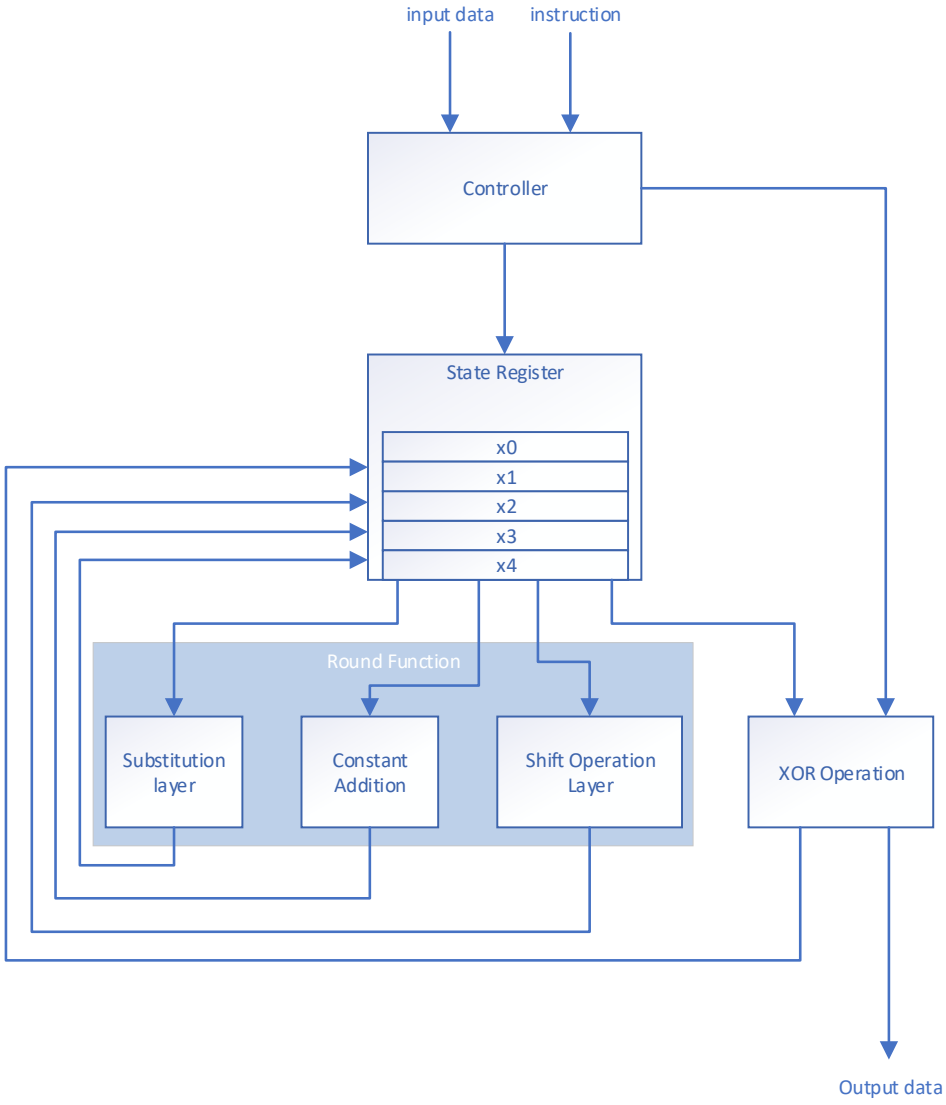


Figure 5.9: Data flow diagram of module: "cryptography-algorithm".

As already mentioned, twelve different instructions are available to control this hardware module. The following list describes all of them with their characteristic:

- „0000“ – Load IV: The IV is loaded into the first 64 bits of the state register x0.
- „0001“ – Load Key1: The first 64 bits of the key, which are provided at port *data_in*, are loaded into the second 64 bits of the state register (register x1). Additionally, these part of the key is also stored in the controller, because the key has to be available at a later point again.

- „0010“ – Load Key2: The second part of the key (64 bits) is read from the *data_in* port and stored in the third register x2 of the state register. As for the first part of the key, this data word is also stored in the controller to complete the storage of the key.
- „0011“ – Load Nonce1: In register x4 of the state register the first half of the nonce data word is stored, which is provided on the input data port.
- „0100“ – Load Nonce 2: The second 64 bits of the Nonce is stored in the fifth register x4. As before, it is read from the port *input_data*.
- „0101“ – Initialization: This instruction starts the initialization process. All operations are described in chapter 4.6.
- „0110“ – Load A: A 64 bit block of the associated data is loaded into the module and processed.
- „0111“ – Load last A: The last part of the associated data is provided at the input data port and is processed as described in chapter 4.6.
- „1000“ – Load P: A 64 bit width data word from the plaintext is read from the input data port and is processed. The Flag *Output_Valid* gets high after finishing the processing.
- „1001“ – Load last P: Since the processing of the last block is different, this instruction is used for the processing of the last plaintext-block. The signal *Output_Valid* is set to logic one after providing the result at the output data port.
- „1010“ – Load C: This instruction is similar to ”Load P”. A 64 bit data word of the cipher text is provided at the input data port. After the processing, the result appears at the output data port and the corresponding flag is set to High.
- „1011“ – Load last C: The last cipher block is processed as described in chapter 4.6. The result is provided at the output data port and the *Output_Valid* gets high.
- „1100“ – Finalization: The last step of encryption or decryption is the finalization. It generates the tag, which is provided at the output data port in two cycles, because it is 128 bit width and the output data port has only a width of 64 bits. Corresponding to the tag, the flag *Output_Tag* is set to ”10” for indicating the MSB-Bits of the tag or ”01” for the indicating of the LSB-Bits of the tag.
- „1101“ – NOP: no operation
- „1110“ – NOP: no operation
- „1111“ – NOP: no operation

The test system loaded on the evaluation kit is depicted in figure 5.10. The same input data is forwarded to both modules: the RTL design and the HLS generated design. Different from the "data flow" module, the output ports of these two modules are not compared at the same clock cycle, because they include a cryptography algorithm, and it is unnecessary and impossible to design them completely identically.

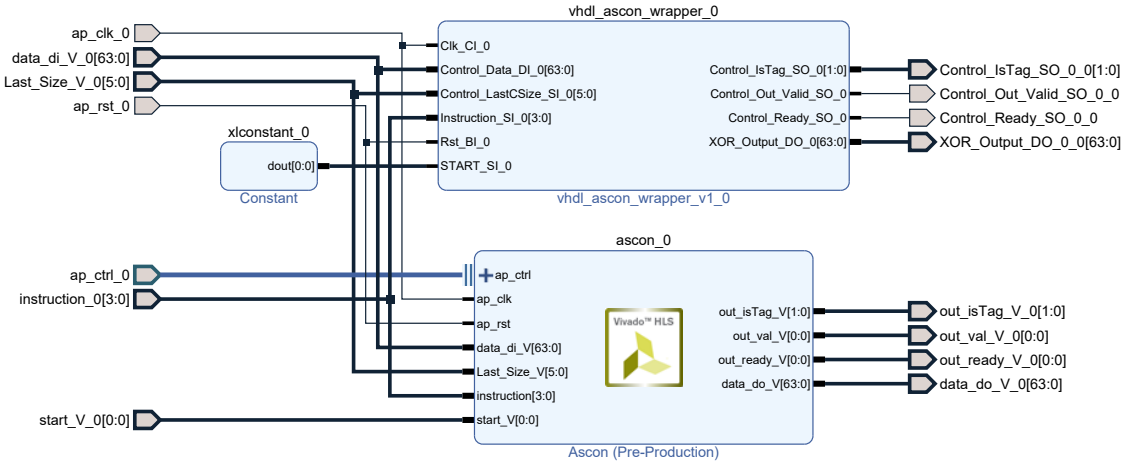


Figure 5.10: Comparison of module: "cryptography-algorithm".

5.3.1 RTL Implementation

As described before, the RTL implementation consists of different parts: Controller, State Register, Substitution Layer, Constant Addition, Shift Layer and XOR Operation. In addition to the data flow graph in figure 5.9, all control signals between the mentioned parts of this module are shown in figure 5.11.

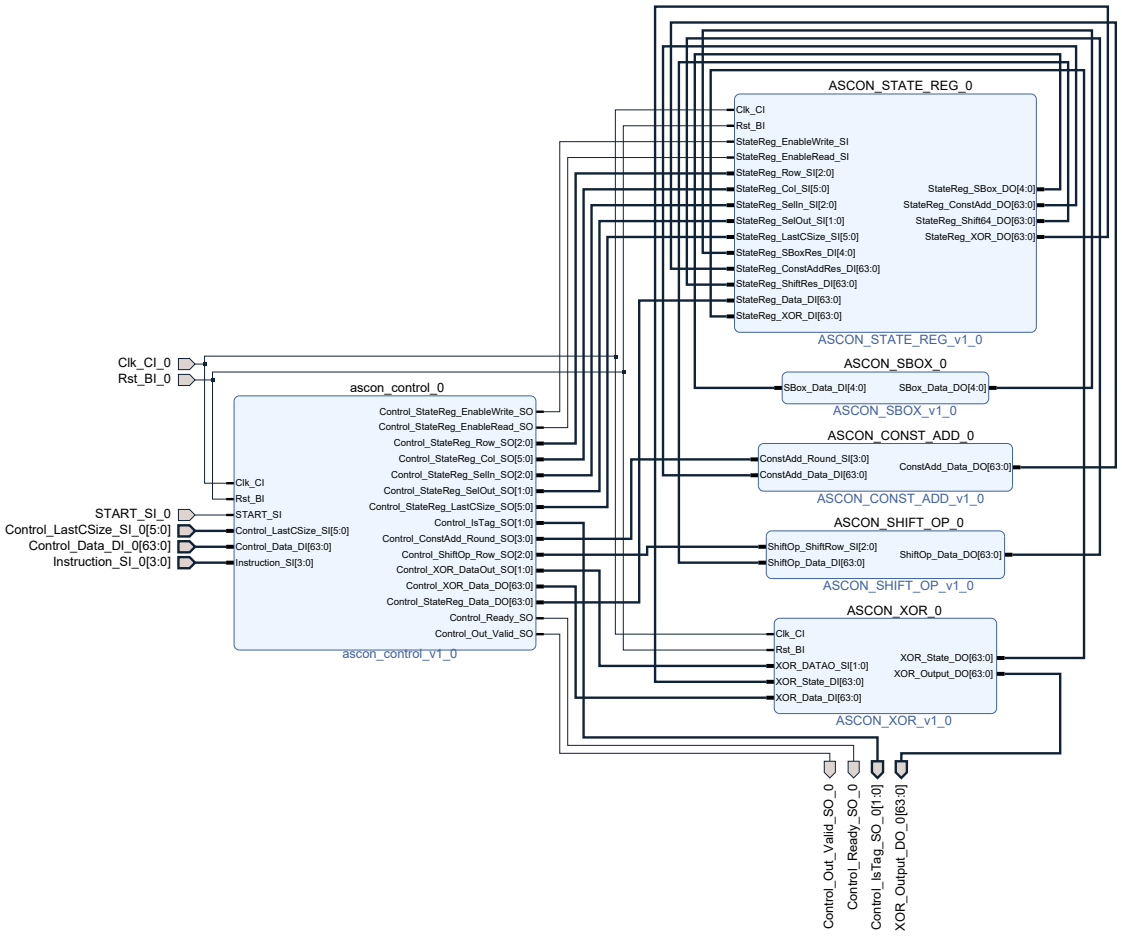


Figure 5.11: RTL implementation of module: "cryptography-algorithm".

5.3.2 HLS Implementation

The C/C++ code for the HLS of this module looks a little bit different to the described data flow graph. All instructions of this module are defined as a *switch-case* statement. So, no control signals between the individual parts of this module has to be added. Therefor, the module design is more clear and less effort is needed. But more information about the quality and complexity is described in chapter 5.4.

```
1  ...
2  void ascon(uint64 data_di, uint6 Last_Size, instruction_type instruction,
3      uint1 start, uint2* out_isTag, uint1* out_val, uint1* out_ready,
4      uint64* data_do) {
5
6    switch (instruction) {
7    case WAITING:
8      ...
9      break;
10   case LOAD_IV:
11     ...
```

```
12      break;
13    case LOAD_KEY1:
14      ...
15      break;
16    case LOAD_KEY2:
17      ...
18      break;
19    case LOAD_NONCE1:
20      ...
21      break;
22    case LOAD_NONCE2:
23      ...
24      break;
25    case INITIALIZATION:
26      ...
27      break;
28    case LOAD_A:
29      ...
30      break;
31
32    ... // further instructions are implemented
33    }
34  }
```

Listing 5.5: C/C++ implementation of the cryptographic-algorithm module.

Directives have to be added for the synthesis process like described for the "data flow" module. Mainly, the ports are defined as *ap_none*, which defines no additional signals for the corresponding port, e.g. no valid signal. In addition, the interface type of the function is *ap_ctrl_chain*, whereby signals like *start*, *reset* or *finished* are generated automatically. Further on, the for-loop for calling the round function is defined as a pipeline. The state register is split into individual 64 bit width registers, because in the C/C++ code it is implemented as an array.

```
1  ############################################################
2  ## This file is generated automatically by Vivado HLS.
3  ## Please DO NOT edit it.
4  ## Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
5  ############################################################
6  set_directive_interface -mode ap_none "ascon" data_di
7  set_directive_interface -mode ap_none "ascon" Last_Size
8  set_directive_interface -mode ap_none "ascon" instruction
9  set_directive_interface -mode ap_none "ascon" start
10 set_directive_interface -mode ap_none "ascon" out_isTag
11 set_directive_interface -mode ap_none "ascon" out_val
12 set_directive_interface -mode ap_none "ascon" out_ready
13 set_directive_interface -mode ap_none "ascon" data_do
14 set_directive_interface -mode ap_ctrl_chain "ascon"
15 set_directive_pipeline "ascon/ascon_label0"
16 set_directive_array_partition -type complete -dim 1 "ascon" state_reg
17 set_directive_pipeline "round_function"
```

Listing 5.6: Directives for the automatic synthesis process of cryptographic module.

5.3.3 Testbench: Generated by HLS and Matlab

The test bench is also written in C/C++ code and translated into RTL by HLS. The testbench includes the correct order of instructions to achieve the described behavior of this algorithm. The test data is generated by Matlab and is stored in text files. The C/C++ testbench can load this text files and uses this data for their instructions, e.g. executing of instruction *load_key1* splits the 128 bit width key stored in textfile key.txt and provides the first 64 bit at the input data port. The same procedure is done for the other possible instructions. Figure 5.12 depicts schematically this procedure.

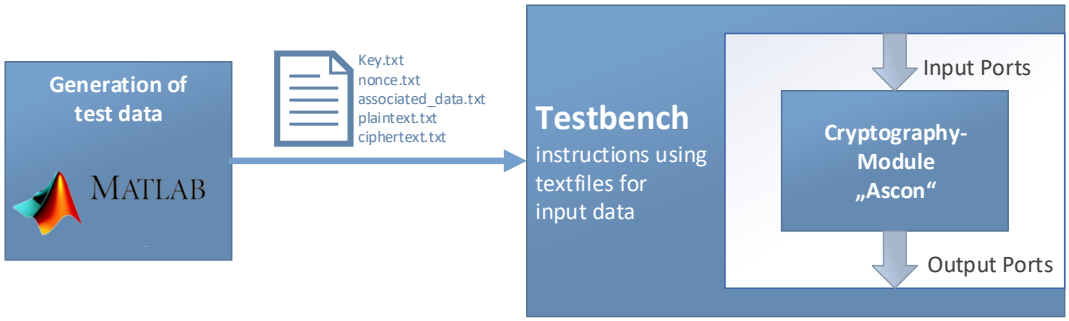


Figure 5.12: Testbench of module: "cryptography-algorithm".

After the synthesis process of the C testbench, the test data is stored in the corresponding VHDL files of the testbench. This procedure for achieving a test system for several hardware modules is very time-saving and less complex compared to generate the whole testbench on RTL.

5.4 Results and Comparison of Designs

First of all, the behavior of the two different generated modules are compared. The first part of this chapter deals with figures showing the behavior. Finally, the report about the quality of the designs is presented.

5.4.1 Module: "data flow"

The testbench provides data to the individual input ports of the module. Generally, the receiver data stream (ports *rx_charisk* and *rx_data*) consists of a K-word followed by some data, whereby all possible K-words appear once in the test procedure. So every state is reached during the execution of the testbench and the behavior can be checked. Since the output of both modules with different design methodology is connected to comparators, the identical behavior of them can be additionally checked. Figure 5.13 shows the output of the comparators (signal name starts with *res_...*). The output is set to logic high during the whole test, except at the beginning some glitches appear, because the initialization wasn't finished. But these are not relevant for the whole test.

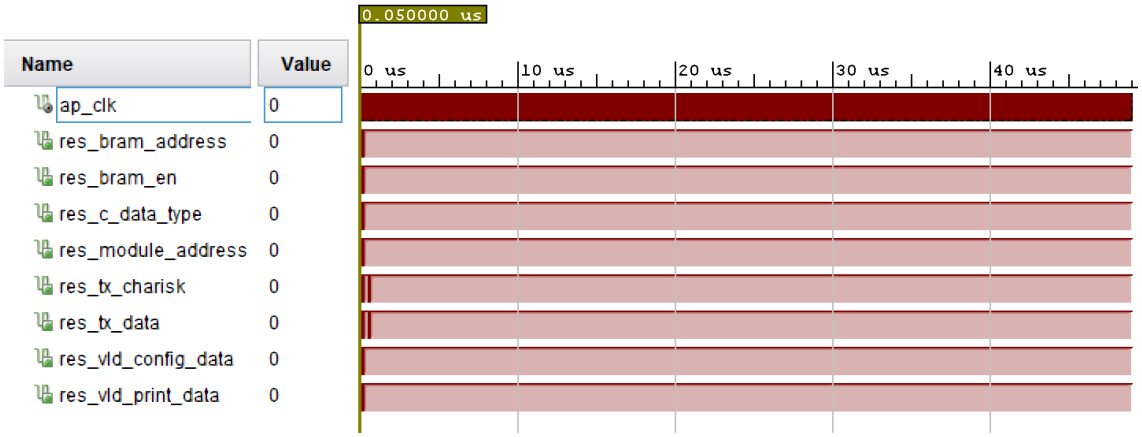


Figure 5.13: Check of the complete behavior.

As already mentioned, special K-words control the behavior of the module, if the following data stream is print data or config data. Figure 5.14 shows an excerpt of a print data stream. Starting point is the K-word K28_5 or in hexadecimal code "0xbc". The module has a latency of five clock cycles. The data appears at the output port *p_data_o* of the RTL design and at *p_data_o_V* of the HLS generated module. In addition, the corresponding valid signal is set to logic high. Further on, the received data stream is

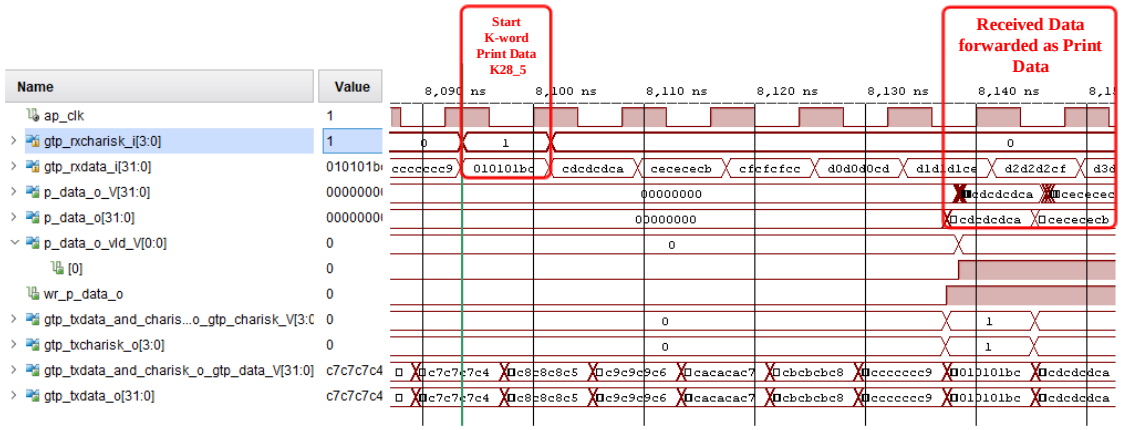


Figure 5.14: Comparison of print data.

forwarded to the transmitter output ports, because several GTP interfaces include one module like this and are connected together over the GTP ring. Sending data from one to another GTP interface requires the forwarding of the received data at the transmitter port.

Receiving configuration data is indicated by one of the K-words of table 5.1, except K-word K28_5. In figure 5.15 K28_6 appears in the data stream. According to the design description for the processing of configuration data mentioned in chapter 5.2 all output ports are controlled. In addition, the transmitter output ports forward the received data as well.

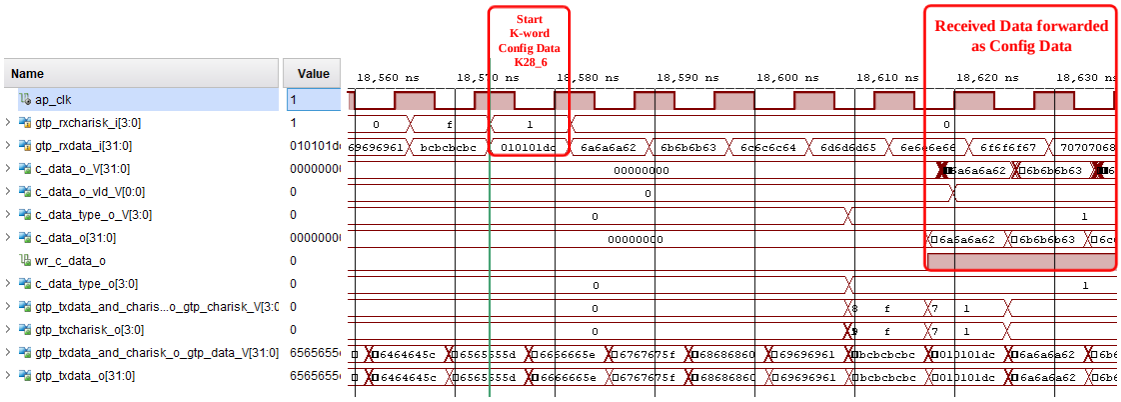


Figure 5.15: Comparison of config data.

5.4.2 Module: "cryptography algorithm"

The cryptography-algorithm module is also verified by a test bench generated with HLS. It includes the correct order of the required instructions. After the finishing of an instruction, the next is executed. The input data is provided by Matlab. Matlab translates a random text into 64 bit width data words. According to the number of data words for e.g. plaintext, the same number of *Load_P* instructions are executed.

Figure 5.16 shows an example of this instruction (instruction 8: *Load_P*).The output data or the encrypted text is provided at the output port *data_do_V_0* from the HLS generated module or at *XOR_Output_DO_0_0* of the corresponding RTL module. It is possible to see, that the data latency for the RTL module is higher. It takes two clock cycles, whereby the HLS generated module provides the data at the same clock cycle like the instruction is loaded into the module. The data itself is equal on both output data ports. Therefore, the behavior has also to be the same. Additionally, this result equates to the output data, which was also generated in Matlab.

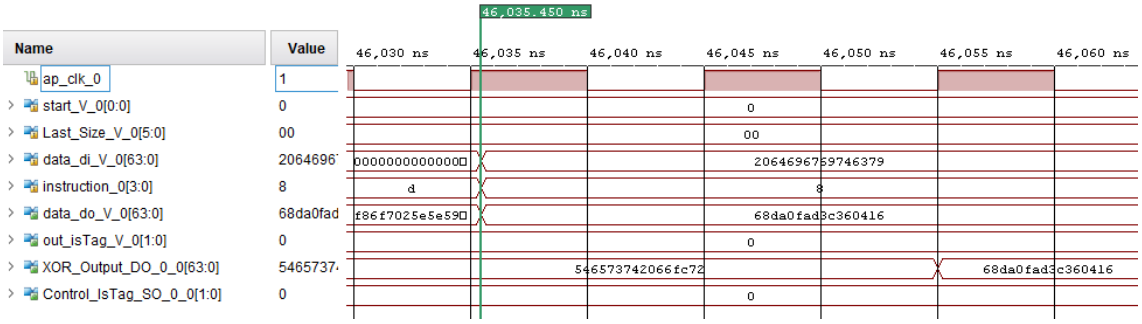


Figure 5.16: Check of instruction *Load_P*.

The second example for the verification is the result of the tag. The calculation of the corresponding tag for the encrypted text takes a little bit longer for both modules. Figure 5.17 depicts, that the tag is provided using the HLS generated module after about 25 clock cycles. The RTL module needs much more clock cycles, after about 8000 ns the result

of the Tag appears, whereby one clock cycles equates to 10 ns. As obvious in figure 5.17, the implementation of the control signal *Control_isTag_SO_0_0* and *out_isTag_V_0* is a little bit different, because the control signal *Control_isTag_SO_0_0* is shifted by one clock cycle. This is a design error which will be solved in future work.

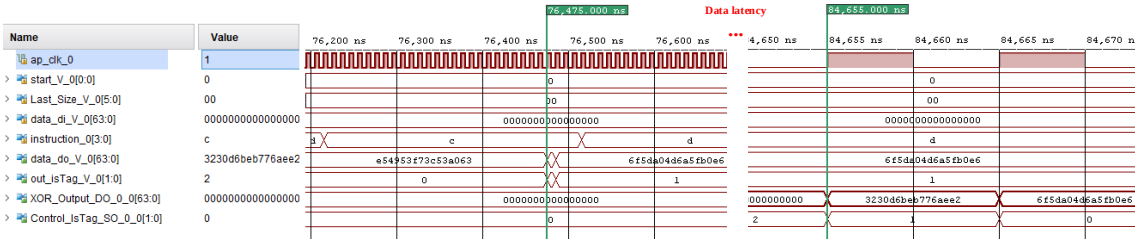
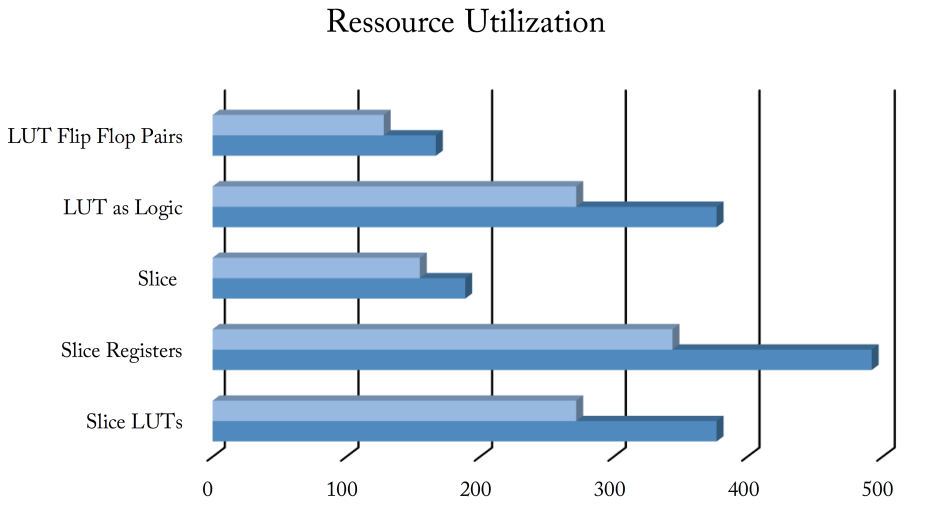


Figure 5.17: Check of instruction *Finalization*.

5.4.3 Allocated Area

The important part of the whole thesis is the resulting quality of the different modules. The goal is to achieve the same number of resource utilization for HLS generated designs as for the corresponding RTL module. In addition, also the power consumption of the individual modules is considered and finally the reduction of the overall design time of each module is listed and compared among themselves.

The resource utilization for the "data flow" module is shown in figure 5.18. Generally, the number of required resources are lower for RTL design. The number of slice LUTs is

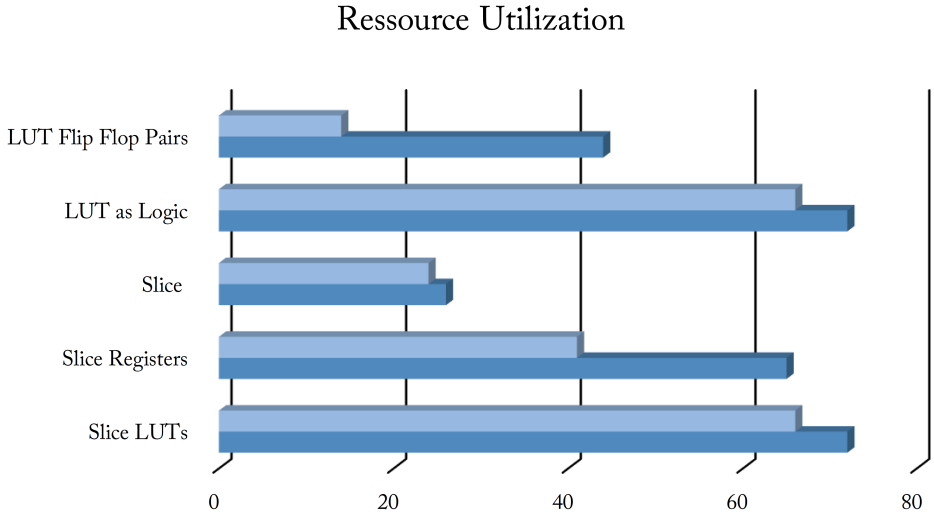


	Slice LUTs	Slice Registers	Slice	LUT as Logic	LUT Flip Flop Pairs
RTL design	272	344	155	272	128
HLS design	377	492	189	377	167

Figure 5.18: Comparison of resource utilization for module: "data flow".

about 28% lower as for HLS designs. Other types of resources like slice registers, slices, and LUT Flip Flop Pairs are 30%, 18%, and 23% lower.

The same type of results for the "cryptography-algorithm" module are depicted in figure 5.19. The number of utilized resources is also lower for the RTL module as for the HLS generated design.



	Slice LUTs	Slice Registers	Slice	LUT as Logic	LUT Flip Flop Pairs
RTL design	66	41	24	66	14
HLS design	72	65	26	72	44

Figure 5.19: Comparison of resource utilization for module: "cryptography-algorithm".

In summary, it can be stated that HLS generated designs require a higher number of resources to achieve the same behavior as RTL modules. If a module has to provide high quality according to area and resource utilization, designers have to choose the traditional design method on RTL.

5.4.4 Power Consumption

The second parameter, which is compared to each other is the power consumption. In mobile applications, the single available power source is often a battery pack. So it is very significant to keep it as low as possible. The following two pictures show the comparison between the power consumption of RTL designed and HLS generated modules.

Figure 5.20 shows the results of power consumption of the "data flow" module. Since no sophisticated calculations are done, the power consumption is very low. A comparison makes no sense.

The comparison of power consumption for the "cryptography-algorithm" module is more expressive. An algorithm includes much more switching tasks for the complex operations. These tasks also require much more energy. Thus, the power consumption for the RTL design is seven times higher as before for the "data flow" module. The HLS

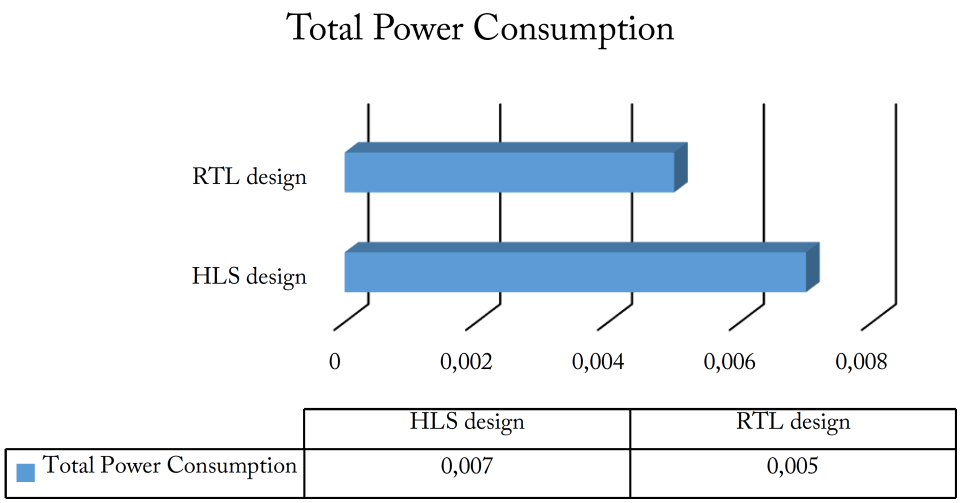


Figure 5.20: Total power consumption of module: "data flow". (Power consumption in W)

generated design requires far more power - about 30 times more power. However, keep in mind that the smaller data latency on the output data port for HLS designs affects this result additionally.

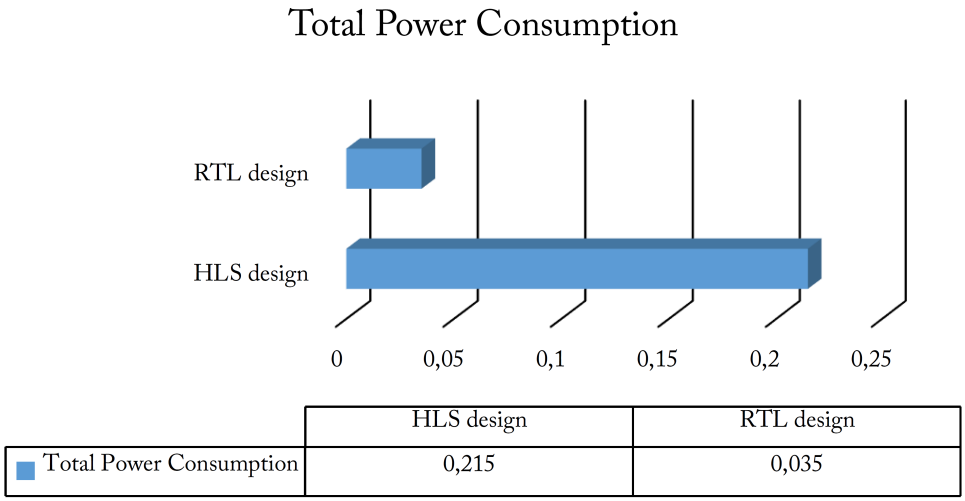


Figure 5.21: Total power consumption of module: "cryptography-algorithm". (Power consumption in W)

Generally, for very low power applications designers should prefer the traditional designing on RTL, because it's possible to define the behavior of a design more accurate. HLS does many definitions by itself.

5.4.5 Design Effort

The last parameter for comparison of RTL designs and HLS generated designs is the number of source lines of code (SLOC). Generally, that way shows the effort of spending time for each module. So it is one of the best way to compare each module for a meaningful evaluation. In the following two tables the associated number of code lines for each module is listed.

First, the comparison of module "data flow" is presented. The RTL module and the HLS generated module have almost a similar number of code lines. The RTL module is designed with 23% more code lines. The HLS generated module consists of a header file *.h and a *.cpp file, on the other hand the RTL module is defined in one single *.vhdl file.

Design Effort: Module "data flow"			
RTL		HLS	
files	# of code lines	files	# of code lines
*.vhdl	404	*.h	63
		*.cpp	265
overall	**404**	**overall**	**328**

Table 5.2: Number of source lines of code for module: "data flow".

The second module dealing with the cryptography algorithm called "Ascon" presents a huge difference between the two design methods. The module designed on RTL is written in 1214 code lines, whereas the HLS generated design is created writting 320 code lines.

Design Effort: Module "cryptography-algorithm"			
RTL		HLS	
files	# of code lines	files	# of code lines
ascon_const_add.vhdl	41	*.h	42
ascon_shift_op.vhdl	45	*.cpp	278
ascon_sbox.vhdl	94		
ascon_control.vhdl	464		
ascon_state_reg.vhdl	270		
ascon_xor.vhdl	59		
ascon_top.vhdl	241		
overall	**1214**	**overall**	**320**

Table 5.3: Number of source lines of code for module: "cryptography-algorithm".

So, this table depicts the big advantage of HLS. For very large designs like different algorithms the design time can be reduced significantly. HLS can decrease the number of source lines of code, which are necessary for a cryptography algorithm, by about 73%. Finally, the biggest benefit of using HLS is the much lower time exposure against traditional designing methods on RTL.

Chapter 6

Conclusion

As already mentioned, this thesis is done in cooperation with co. *Durst Phototechnik Digital Technology GmbH* in Lienz, Austria. It is a company, which designs huge printing systems for industrial applications. All of the images loaded into these systems have to be pre-processed. Image processing has a very high computational effort, whereby this effort is rapidly increasing since the size of images and the complexity of processing algorithms raises. Therefore, this company is willing to transfer the computational effort from standard processor to hardware processing systems running on FPGAs, but the adoption should not utilize to much time or effort. So the usage of HLS is verified and compared to the conventional design method on RTL. The output of this thesis presents the benefits or disadvantages of HLS in relation to the use of it for hardware development.

Generally, this thesis includes also a description about the basics of HLS for a better understanding. It is beneficial to be familiar with HLS, because the quality of the evaluation for this two design methods (HLS generated design and design on RTL) gets more validity. During the research for this thesis, also several papers about this comparison was discovered. The results were presented in chapter 3 and are likened in chapter 5.4. The whole design process includes a description for generation of designs on RTL and HLS generated designs.

Generally, the comparison shows, that designs generated by HLS enable the same behavior of any hardware as it is possible to design on RTL. The "data flow" module was designed completely equal for both methods. The output ports provide the same data after a specific number of clock cycles. So this module is very useful for an exact comparison. This module shows, that the number of utilized resources is about a third higher for HLS designed modules than for the design on RTL. The power consumption is approximately equal for both design methods. The second module deals with a cryptography algorithm called "Ascon". The module designed on RTL is not identical to the HLS generated design, because it's to complex to achieve completely the same behavior. According to the resource utilization, the HLS generated module performs worse than the RTL model. Additionally, the power consumption is very bad using HLS for the design process. However, the design effort, which is measured in number of source lines of code, is significant lower using HLS than for the designing on RTL. This is the most expressive argument for using HLS in the design process. Although the previous aspects signifies the RTL design technique, the aspect of reducing the time effort for designing a hardware model is significant. Therefore it is sensible to use HLS for the generation of hardware modules for complex algorithms.

This results conforms also to the results of the presented works in chapter 3. HLS is very useful for complex systems, because the time exposure can be reduced, but the quality of the resulting hardware is a little bit worse according to resource utilization.

The result of this thesis could be an impulse for the adoption of HLS in the hardware design process. For the co. *Durst Phototechnik Digital Technology GmbH*, this means that the transfer of the execution of image processing algorithms from standard processors or from the workstation to FPGA units could be solved by HLS very easily and time-saving. The algorithms can be generally reused without many modifications and synthesized into hardware description. In further consequence, this is able to be loaded on a FPGA or a system-on-chip.

Appendix A

Terminology

A.1 Definitions

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
BRAM	Block Random-Access Memory
CAPH	Language for Implementing Stream-Processing Applications
CDFG	Control- and Data-Flow Graph
DFG	Data-Flow Graph
DSP	Digital Signal Processor
FF	Flip-Flop
FIFO	First-In, First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GTP	Gigabit Transceiver (defined by Xilinx Inc.)
HDL	Hardware Description Language
IC	Integrated Circuit
LUT	Look-Up Table
HLS	High-Level Synthesis
REG	Register
RTL	Register-Transfer Level
SLoC	Source Code of Lines
SoC	System-on-Chip
uC	Microcontroller
VLSI	Very-Large-Scale Integration

Bibliography

[1] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. The Springer International Series in Engineering and Computer Science. Springer US, 2012. URL: https://books.google.at/books?id=yTbSBwAAQBAJ.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers*, 26(4):8–17, July 2009. doi:10.1109/MDT.2009.69.

[3] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq Book: Embedded Processing Withe ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC*. Strathclyde Academic Media, 2014. URL: https://books.google.at/books?id=9dfvoAEACAAJ.

[4] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon. *Institute for Applied Information Processing and Communications*, v1.2, 2016.

[5] J. Erickson and M. Warren. Modern system on chip challenges demand development of new skills in electronic engineering graduates. In *2013 3rd Interdisciplinary Engineering Design Education Conference*, pages 32–35, March 2013. doi:10.1109/IEDEC.2013.6526755.

[6] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin. *High — Level Synthesis: Introduction to Chip and System Design*. Springer US, 2012. URL: https://books.google.at/books?id=1BTaBwAAQBAJ.

[7] D. D. Gajski and R. H. Kuhn. Guest Editors' Introduction: New VLSI Tools. *Computer*, 16(12):11–14, Dec 1983. doi:10.1109/MC.1983.1654264.

[8] habeebq. Writing a 2x2 Matrix Multiplier in VHDL. http://habeebq.github.io/writing-a-2x2-matrix-multiplier-in-vhdl.html, 2016.

[9] E. Homsirikamol and K. G. George. Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 120–127, Dec 2017. doi:10.1109/FPT.2017.8280129.

[10] Ekawat Homsirikamol and Kris Gaj. Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study. In

Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 217–228, Cham, 2015. Springer International Publishing.

[11] Avnet Inc. *Picozed*. AVNET Reach Further, 2017. URL: `http://picozed.org/product/picozed`.

[12] Xilinx Inc. Vivado Design Suite: VivadoHLS. [online], 2013. URL: `http://www.xilinx.com/products/design-tools/vivado/index.htm`.

[13] Xilinx Inc. *Vivado Design Suit User Guide, High-Level Synthesis, UG902 (v2017.1)*, 2017. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf`.

[14] Books Llc, S. Wikipedia, and B. Group. *Hashing: Hash Functions, Hash Table, Pearson Hashing, Hmac, Collision, Rabin-karp String Search Algorithm, Bloom Filter*. General Books, 2010. URL: `https://books.google.at/books?id=sB2ZSQAACAAJ`.

[15] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009. `doi:10.1109/MDT.2009.83`.

[16] J. Michael and M. Chinnadurai. High Level Synthesis Tools-an Overview from Model to Implementation. *Middle East Journal of Scientific Research*, 22:241 – 254, 01 2014.

[17] D. Micheli. *Synthesis & Optimization Of Dig. Circuits*. McGraw-Hill series in electrical and computer engineering. McGraw-Hill Education (India) Pvt Limited, 2003. URL: `https://books.google.at/books?id=aNoSUQiCD8MC`.

[18] 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*, 35(10):1591–1604, Oct 2016. `doi:10.1109/TCAD.2015.2513673`.

[19] R. Niemann and P. Marwedel. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Springer US, 1998. URL: `https://books.google.at/books?id=ZHdId5u0GCYC`.

[20] Staff of Berkeley Design Technology. *BDTI High-Level Synthesis Tool Certification Program (HLSTCP)*. Berkeley Design Technology, Inc. URL: `https://www.bdti.com/Services/Benchmarks/HLSTCP`.

[21] Staff of Berkeley Design Technology. The AutoESL AutoPilot High-Level Synthesis Tool. Technical report, Berkeley Design Technology, Inc., 2010. URL: `https://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf`.

[22] Staff of Berkeley Design Technology. The Synopsys Synphony C Compiler. Technical report, Berkeley Design Technology, Inc., 2010. URL: `https://www.bdti.com/Resources/BenchmarkResults/HLSTCP/Synphony`.

[23] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry. Design Productivity of a High-Level Synthesis Compiler versus HDL. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 140–147, July 2016. doi:10.1109/SAMOS.2016.7818341.

[24] S. Sinha and T. Srikanthan. High-Level Synthesis: Boosting Designer Productivity and Reducing Time to Market. *IEEE Potentials*, 34(4):31–35, July 2015. doi:10.1109/MPOT.2013.2292957.

[25] J. Sérot, editor. *A lightweight implementation of dependently sized types for a functional hardware description language*. Workshop on Trends in Functional Programming, 2015. URL: http://caph.univ-bpclermont.fr/papers/articles/tfp2015.pdf.

[26] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, Dec 2012. doi:10.1109/TCSVT.2012.2221191.

[27] P. Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL*. Elsevier Science, 2015. URL: https://books.google.at/books?id=RuH5AwAAQBAJ.

[28] Michael D. Zwagerman. High Level Synthesis, a Use Case Comparison with Hardware Description Language. Master's thesis, Grand Valley State University, 2015. URL: https://scholarworks.gvsu.edu/theses/755/.