BACHELOR'S THESIS

INTEL HIGH LEVEL SYNTHESIS COMPILER'S FEATURES

Tony Hänninen

Supervisor: Jukka Lahti

DEGREE PROGRAMME IN ELECTRICAL AND COMMUNICATIONS ENGINEERING

2019

Tony H. (2019) Intel High Level Synthesis Compiler's features. University of Oulu, Degree programme in electrical and communications engineering. Bachelor's thesis, 22 s

ABSTRACT

The increasing complexity of systems and applications increases workload and makes development cycles longer. High-Level Synthesis (HLS) tools have been developed to alleviate this by raising the level of abstraction from Register Transfer Level (RTL). This work introduces the basics of Field Programmable Gate Arrays (FPGA) and HLS. The Intel HLS Compiler and its features are studied in more detail. Intel HLS compiler is a commercial HLS tool, which takes in untimed C++ as input and generates production-quality RTL that is optimized for Intel FPGAs.

Key words: FPGA, HLS, Intel HLS Compiler.

Tony H. (2019) Intelin High Level Synthesis Compiler -ohjelman ominaisuudet. Oulun yliopisto, Elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 22 s

TIIVISTELMÄ

Laitteistojen ja systeemien kasvava kompleksisuus kasvattaa työmäärää ja pidentää suunnitteluvuohon kuluvaa aikaa. Korkean tason synteesin (HLS) työkalut ovat kehitetty automatisoimaan ja nopeuttamaan digitaalisuunnittelua nostamalla abstraktiotasoa korkeammalle rekisterinsiirtotasolta (RTL). Tässä työssä kuvataan FPGA-piirien ja HLS:n perusperiaatteet, jonka jälkeen Intel HLS Compiler -työkalua tutkitaan tarkemmin. Intel HLS Compiler on työkalu, joka generoi RTL-koodia C++-kielellä kirjoitetuista algoritmeista.

Avainsanat: FPGA, korkean tason synteesi, Intel HLS Compiler

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

ALM	Adaptive Logic Module
ASIC	Application Specific Integrated Circuit
CDFG	Control and Data Flow Graph
CLB	Configurable Logic Block
DLL	Delay-Locked Loop
DSP	Digital Signal Processing
ESL	Electronic System-Level
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
HLS	High-level Synthesis
HW	Hardware
LAB	Logic Array Block
LUT	Look-Up Table
MLAB	Memory Logic Array Block
PCIe	Peripheral Component Interconnect Express
PLL	Phase-Locked Loop
QoR	Quality-of-Results
RAM	Random Access Memory
RTL	Register Transfer Level
SoC	System on Chip
SRAM	Static Random Access Memory
IC	Integrated Circuit
II	Initiation Interval
IP	Intellectual Property

1. INTRODUCTION

The increasing complexity of systems and applications have urged the design community to raise the level of abstraction from Register Transfer Level (RTL). Electronic system-level (ESL) design automation has been identified as the next logical step to boost productivity in the semiconductor industry, where High-Level Synthesis (HLS) plays a central role. HLS enables designers to be more productive by leaving the implementation details to the design algorithms and tools. These details include the ability to determine the precise timing of operations, data transfer, and storage. [1]

There has been a greater push from Field-Programmable Gate Array (FPGA) designers to adopt HLS tools faster compared to Application-Specific Integrated Circuit (ASIC) designers for the following reasons: Modern FPGAs embed many predefined/fabricated Intellectual Properties (IP). These blocks can be modeled precisely ahead of time and as a result, it is possible for the HLS tool to apply a platform-based design methodology and achieve higher Quality-of-Results (QoR). In addition, recent advances in FPGAs have made reconfigurable computing platforms feasible to accelerate many high-performance computing applications, such as image and video processing, financial analytics, bioinformatics, and scientific computing applications. For application software developers, it is essential to provide a highly automated compilation/synthesis flow from C/C++ to FPGAs, since RTL programming is exotic for most of them. [2]

This work studies the Intel HLS Compiler, which is a commercial HLS tool that takes in untimed C/C++ as input and generates production-quality RTL that is optimized for Intel FPGAs. Intel HLS compiler speeds up verification time over RTL by raising the abstraction level for FPGA hardware (HW) design. As we can see from Table 1, algorithms and models developed in C/C++ are orders of magnitude faster than hand-coded RTL and require 80% fewer lines of code. [7]

Table 1. RTL vs untimed C++ functional verification times. [7]

Design	**RTL simulation time**	**C++ runtime**
Advanced Encryption System (AES)	22 minutes	46 ms
Huffman Encoding	13 minutes	52 ms
Complex FIR Filter	4.5 minutes	63 ms

In the following chapters, FPGAs and HLS are introduced and Intel's HLS design flow is studied in more detail.

2. FPGA TECHNOLOGY

FPGAs are semiconductor Integrated Circuits (IC) where a large majority of the electrical functionality inside the device can be reprogrammed. Normally FPGAs comprise of a matrix of Configurable Logic Blocks (CLB), programmable routing that connects these CLBs and Input and Output (I/O) blocks to make off-chip connections. CLBs include logic gates, Look-Up Tables (LUT) and Flip-Flops (FF). Today's FPGAs also include on-die processors, Random Access Memory (RAM) blocks, Digital Signal Processing (DSP) engines, and more. [3][4]

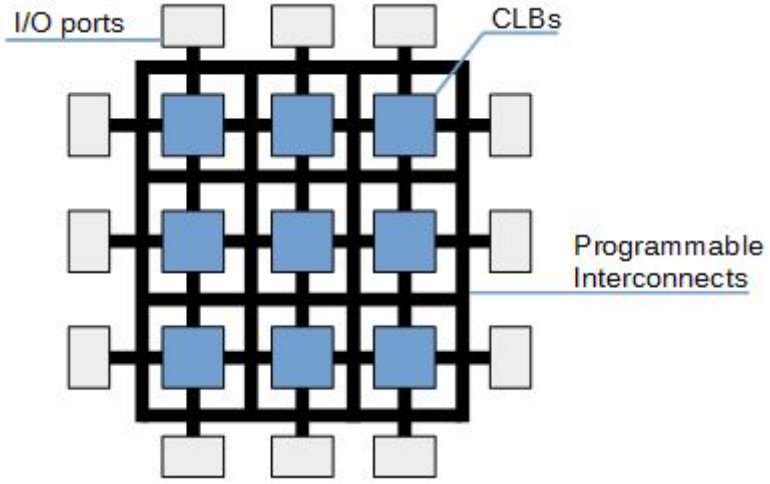

Figure 1. Overview of FPGA architecture.

Intel's Stratix IV is an example of a commercial architecture that uses a heterogenous mixture of blocks. Figure 2 shows the global architectural layout of Stratix IV. The logic structure of the device consists of Logic Array Blocks (LAB), memory blocks and DSP blocks. LABs and Adaptive Logic Modules (ALM) provide the basic logic capacity. They can be used to configure logic functions, arithmetic functions, and register functions. Each LAB consists of ten ALMs. A Memory LAB (MLAB) can be used a simple LAB or as a Static Random Access Memory (SRAM). [5]

The 9-Kbit M9K memory block can be used for general purpose memory applications and the 144-Kbit M144K memory block for processor code storage, packet buffering, and video frame buffering. The Stratix IV also has dynamically configurable Phase-Locked Loops (PLL), Delay-Locked Loops (DLL), high-speed transceivers and hard IP for Peripheral Component Interconnect Express (PCIe). [5]

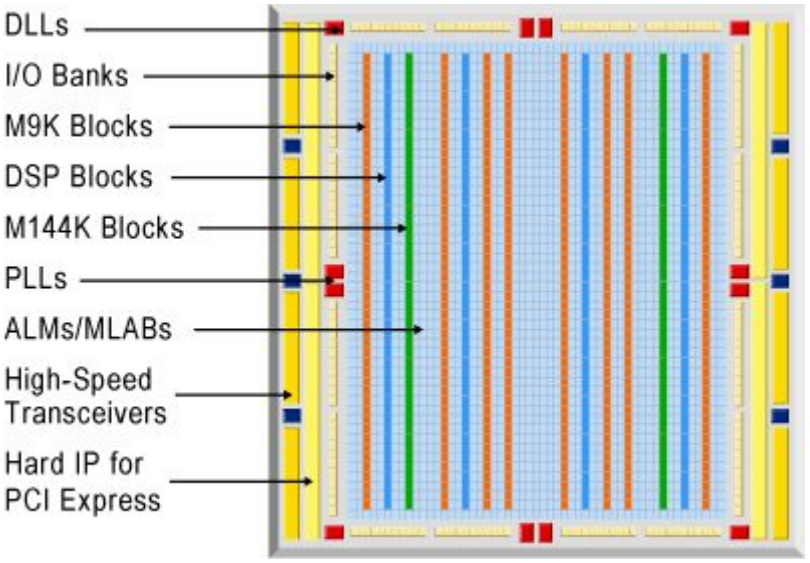

Figure 2. Intel Stratix IV architectural elements.

3. HIGH-LEVEL SYNTHESIS DESIGN STEPS

In a typical HLS design flow, seen in Figure 3, an untimed or partially timed high-level specification is turned into a fully timed implementation with a help of a HLS tool. HLS starts with compiling the functional specification into a formal model. This includes code optimizations, such as dead-code and false data dependency eliminations, and loop transformations. The formal model shows the data and control dependencies between operations and basic blocks. These dependencies are represented in a Control and Data Flow Graph (CDFG), which helps with optimizing the design. [6]

The next step in the HLS design flow is binding operations to allocated HW resources and scheduling the operations to clock cycles. The RTL model is generated, once these tasks are completed. Allocation defines the HW resources needed to satisfy the design constraints and are selected from the defined RTL component library. These resources include things, such as functional units, storage, and connectivity components. All operations in the specification model must be scheduled into clock cycles. The operations are scheduled to execute in parallel or after each other, depending on the data dependencies between them. Every operation is bound to a functional unit capable of executing it, such as arithmetic logic units, multipliers, and shifters. Variables are bound to memories and registers. Storage and functional units are connected with buses, tristate drivers, or multiplexers. These register-transfer components make up the RTL architecture, which is used for logic synthesis. [6]

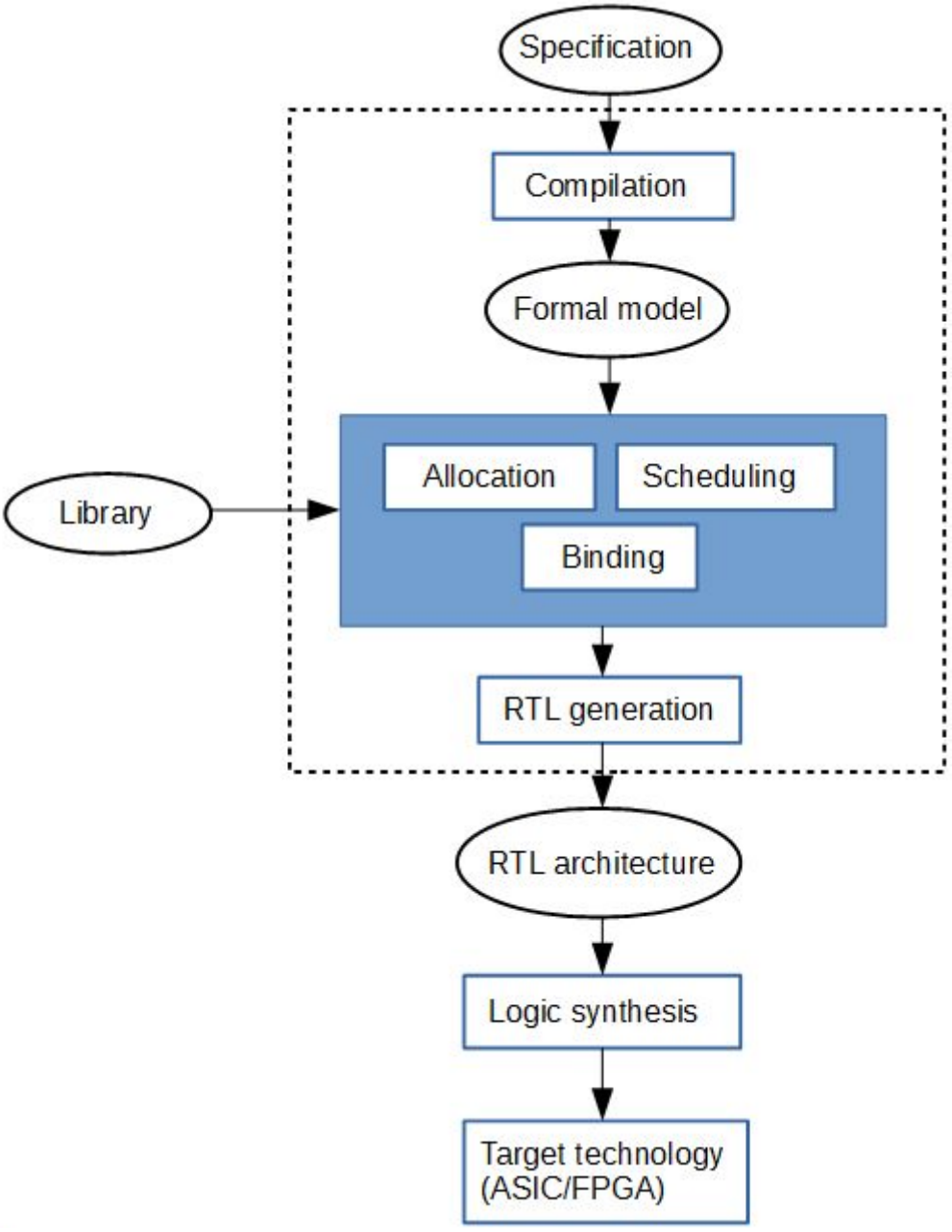

Figure 3. Typical HLS design steps.

Allocation, scheduling and binding are all interrelated but can be performed simultaneously or in a specific sequence, depending on the strategy and algorithms used. For example, allocation will be performed first when scheduling tries to minimize latency or to maximize throughput under a resource constraint. If scheduling tries to minimize the area under timing constraints, allocation is determined during scheduling.

4. INTEL HLS COMPILER

In this chapter, the Intel HLS Compiler design flow is examined and what options there are to optimize the designed component.

4.1. Intel HLS Compiler overview

In the Intel HLS Compiler design flow, seen in Figure 4, designers start with describing the behavior of the IP component using C/C++. This description is purely algorithmic and requires no timing, concurrency or target technology information. The synthesizable C/C++ design is modeled with either fixed-point, integer or floating-point arithmetic. [8]

Once satisfied with the algorithm and the functionality has been verified, the component can be compiled to RTL. Intel HLS Compiler generates a high-level design report that shows estimates of various aspects of how the component will be implemented in HW and can be used to further optimize and refine the design. The tool tries to provide maximum throughput whenever possible but with loop and memory directives it can be traded for smaller latency or area usage. [8]

After being satisfied with the predicted performance of the component, a longer HW synthesis can be performed with Intel Quartus Prime tool.
This tool generates more accurate area and performance estimates of the design. Platform Designer tool, included in Quartus Prime, is used to integrate the design into a system. [8]

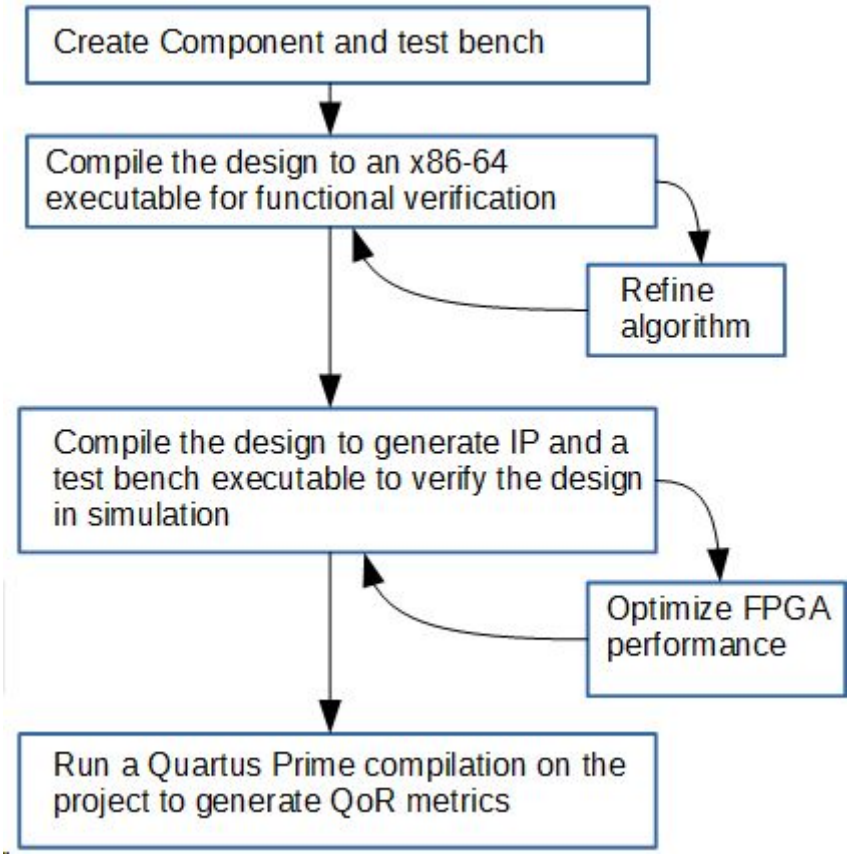

Figure 4. Coarse-grained progression of typical Intel HLS Compiler flow.

4.2. Compiling and verifying the functionality of the IP design

The first step in the design flow is compiling the component and testbench into an x86-64 executable for functional verification, which can then be debugged with a C++ debugger of choice. [9]

Compiling the design to an x86-64 executable is faster than compiling it to HW or HW simulation. This compilation time makes debugging and refining the component algorithm quicker before moving on to HW implementation. [9]

4.2.1. Supported C and C++ libraries and subset for component synthesis

In general, the compiler can synthesize functions that include classes, structs, functions, templates, and pointers. The Intel HLS Compiler has several synthesis limitations regarding the supported subset of C99 and C++. It cannot synthesize code for dynamic memory allocation, virtual functions, function pointers, and C/C++ library functions except the supported math functions, which come from the provided header files. [9]

The Intel HLS Compiler includes an extended math library for FPGA-specific definitions in addition to the standard math library. The compiler also supports Algorithmic C datatypes, which provide arbitrary width integer and arbitrary precision fixed-point support. These datatypes ensure that no data is lost by carrying out the operations in big enough size. However, data can still be lost if the location where it is stored is too small. [9]

4.3. Optimizing and refining the IP design

After verifying the functionality of the component, it can be compiled to RTL. The compiler generates a high-level design report after compiling the component, which can be used to further optimize the design. This report helps with analyzing various component aspects, such as area, loop structure, memory usage, and component pipeline. [9]

4.3.1. Interface synthesis

The design interface is how the component communicates with the rest of the world. The Intel HLS Compiler builds a correspondence between the arguments of the C/C++ function and the RTL module. This RTL module must have top-level ports, or interfaces, that allow the overall system interact with the HLS component. [9]

Once this is established, the compiler builds an interface based on the parameters and interface synthesis constraints defined in the code. With these constraints, the compiler can be forced to implement variables/arrays as registers or embedded memory. In addition, the width of the buses and data streaming can also be controlled. Figure 5 illustrates an RTL component created by the tool for a design written as follows:

```
component int dut(int a, int* b, int i) {
     return a*b[i];
}
```

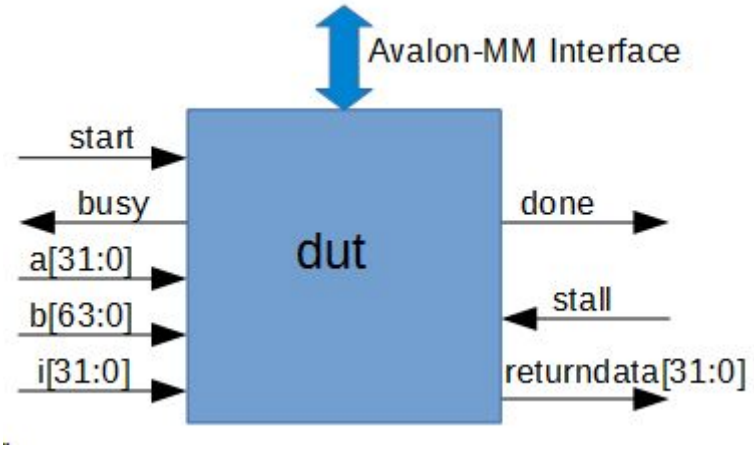

Figure 5. Block diagram of the interfaces and signals for component dut.

Each pointer, scalar, or reference argument in a component results in an input conduit. In addition to this input conduit, all pointers share a single Avalon Memory-Mapped master interface, which the component uses to access system memory. [9]

The Intel HLS Compiler allows you to explicitly declare Avalon Streaming interfaces and Avalon Memory-Mapped Master interfaces on components. These are protocols meant for connecting components in Intel FPGAs. Avalon Streaming interface supports unidirectional flow of data, including multiplexed streams, packets, and DSP data. Avalon Memory-Mapped Master interface is an address-based read/write interface typical of master-slave connections. [9][10]

4.3.2. Loop controls

Looping constructs are common in high-level designs, unlike RTL, where they are seldom used. By default, Intel HLS Compiler tries to pipeline loops to maximize throughput of the design. Throughput is the amount of data that can be processed in a single clock cycle. However, it is possible to manipulate loops with directives, or pragmas, to trade performance and area. All the loop pragmas are presented in Table 2. [1][9]

Table 2. Loop pragmas

#pragma ii *N*	Set loop Initiation Interval (II) to *N*

#pragma ivdep
safelen(*N*)
array(arrayname)
	Ignore local memory dependencies between iterations up to *N* iterations apart.

#pragma loop_coalesce *N*	Convert nested loops down to level *N* to single loop
#pragma unroll *N*	Unroll the loop into *N* copies. *N* is optional, and not specifying *N* fully unrolls the loop.
#pragma max_concurrency *N*	Specify the number of iterations of a loop that can execute simultaneously.

Loop pipelining enables the compiler to execute the next iterations of the loop in parallel. This means that multiple iterations of a loop execute concurrently. Figure 6 shows a basic loop with 3 stages and 3 iterations. A stage is how many operations occur in one clock cycle, so the loop in Figure 4 would have a latency of 9 cycles. Latency describes the time it take to data to pass from the input to the output of the circuit.

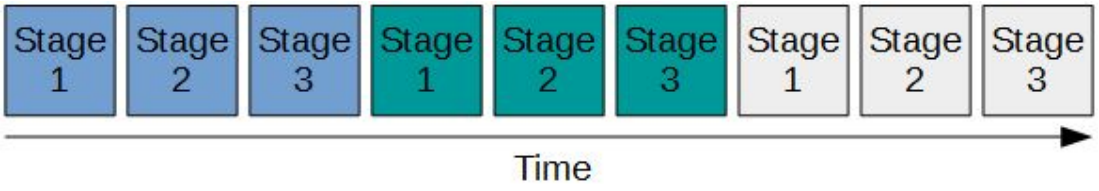

Figure 6. A loop without pipelining.

In Figure 7, a pipelined loop with an Initiation Interval (II) of 1 clock cycle is illustrated. An II of 1 means that there is a one clock cycle of delay between starting each successive loop iteration. The latency in the loop has now gone down to 5 cycles, thanks to pipelining.

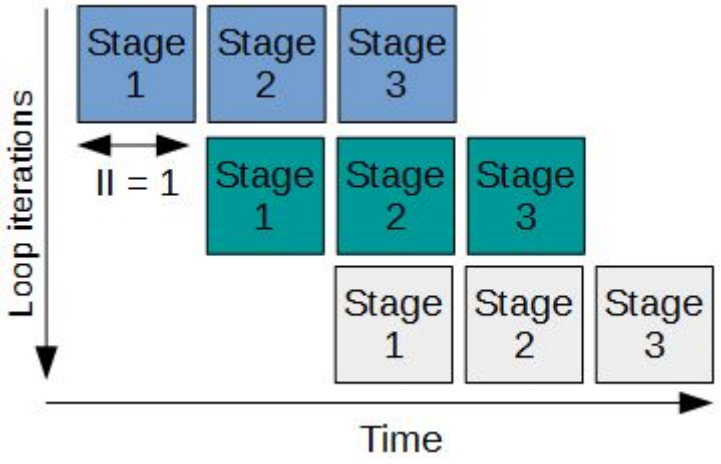

Figure 7. Pipelined loop with an Initiation Interval of 1 clock cycle.

However, not all loops can be pipelined as well as shown in Figure 7, particularly loops where each iteration of the loop depends on the value computed in the previous iteration. For example, if the value of Stage 1 depended on the value of Stage 3, the next iteration (green) could not start executing until previous iteration (blue) reached Stage 3 and would be pipelined with II = 3. This type of dependency is called loop-carried dependency. [9]

In some cases, the compiler assumes there are loop-carried dependencies to avoid any data hazards between load and store instructions. It is possible to tell the compiler that there are no dependencies between loop iterations with the ivdep pragma. This saves area and lowers the II of the loop because the hardware required for avoiding data hazards is no longer required. With the safelen(*N*) and array(array_name) clauses, it is possible to give further information about dependencies to the compiler. For example, safelen(16) tells the compiler that the 16th iteration is the closest iteration dependent on the current iteration. The array(array_name) clause specifies that a particular memory array inside a loop will not cause loop-carried dependencies. [9]

Fully or partially unrolling loops trades an increase in component area usage for a reduction in latency. When a loop is unrolled, the compiler replicates each iteration of the loop in hardware and executes it simultaneously if they are independent. Figure 8 shows what happens if the loop from Figure 6 is fully unrolled.

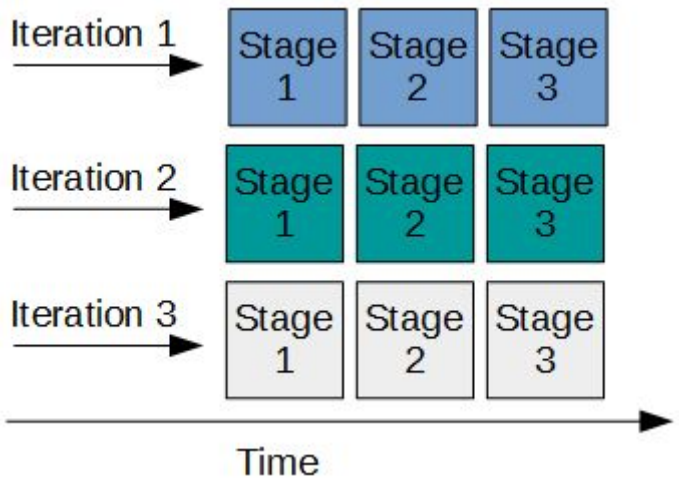

Figure 8. An unrolled loop with three stages and three iterations.

Three iterations of the loop are now completed in three clock cycles, bringing the latency from 9 to 3 cycles. However, three times as many HW resources are needed. [9][11]

The loop_coalesce pragma can be used to merge nested loops together without affecting the loop functionality. With loop merging, it is possible to decide if in the generated hardware the loops run in parallel (merging enabled) or sequentially (merging disabled). Merging nested loops reduces component area usage by reducing the overhead needed for loop control. Additionally, merging nested loops reduces the latency of the component, which could further decrease the component area usage. However, merging loops are not be suitable for all components, as it might lengthen the critical loop initiation interval path. The following example shows how the compiler handles a piece of code if the loop_coalesce pragma is used.

A simple nested loop is written as follows:

```
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

The compiler merges the two loops together so they run as a single loop written as follows:

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```
[1][9]

The max_concurrency pragma is used to control how many iterations of a loop can be in progress at one time. The Intel HLS Compiler tries to maximize concurrency of loops to maximize throughput of components. To achieve this, local memory has to be sometimes duplicated to break dependencies that prevent loops from being fully pipelined. With this pragma however, some performance can be traded for local memory savings. [9]

4.4. Verifying the IP with simulation and integrating it into a system

When compiling a design to a specific FPGA architecture, the Intel HLS compiler links the design C++ testbench with an RTL-compiled version of the component. This component runs in an RTL simulator, Mentor Graphics ModelSim, which has to be installed separately. To generate the verification executable, the compiler performs the following steps: Parses the design and extracts the functions and symbols necessary for component synthesis to the FPGA and compiling the C++ testbench; Compiles the testbench code to generate an x86-64 executable, which also runs the simulator; Compiles the code for component synthesis to the FPGA. This compilation generates RTL for the component and an interface to the x86-64 executable testbench. [8]

Signal logging can be turned on for debugging purposes. However, logging signals slows down the simulation and the waveform files can be very large. To see how well successive invocations of components can be pipelined, the compiler comes with functions for queueing inputs to the components with explicit interfaces. These blocking calls wait for a return value from the component before continuing execution. [8]

After being satisfied with the predicted performance, Intel Quartus Prime can be used to perform a longer hardware synthesis and is needed to integrate the Intel HLS Compiler generated IP into a target FPGA. The high level design report generated by Intel Quartus Prime contains more accurate data estimates for area and performance for the components. The Intel Quartus Prime synthesis can typically take minutes to hours, depending on the size and complexity of the components. [8]

5. DISCUSSION

The Intel HLS Compiler follows the basic HLS principles of taking an untimed algorithm written in a high-level language and synthesizing it into a fully timed HW implementation. The supported high-level language libraries, interfacing, and RTL optimizations, such as loop controls, are all typical to what you would expect from an HLS tool. The support for C++ algorithmic development enables much faster verification and simulation times compared to traditional RTL coding and cuts down development cycles significantly. In addition, the use of C++ is especially valuable for application software developers, as it enables them to develop IP for HW if they are not familiar with RTL.

The Intel HLS Compiler is a powerful tool if the chosen target platform is one of Intel's FPGAs. The automatic software testbench verification against the compiler generated RTL model, and the interactive analysis reports makes it easier for designers to implement and test new FPGA projects. However, Intel's Quartus Prime or Platform Designers is needed to implement the designed component into the target FPGA technology.

6. SUMMARY

This work presented the basic principles of HLS and FPGA technology. HLS greatly reduces development cycles by raising the abstraction level, as RTL-coding is much more time-consuming compared to a high-level language, such as C++. In addition, functional verification times are orders of magnitude faster on this abstraction level. Targeting FPGAs for synthesis makes it easier for HLS tools, since the available resources are known ahead of time. This allows the tool to apply a platform-based design methodology and achieve a higher QoR.

The aim of this thesis was to study the Intel HLS Compiler, which is a commercial HLS tool made to target Intel's FPGAs. The tool takes in untimed C++ and generates production-quality RTL. Production-quality RTL means that the code produced by the tool can be used for production and needs no further optimization. Key features, such as automatic RTL verification to C++, interactive analysis reports, and floating-point support makes the tool compelling for FPGA developers.

REFERENCES

[1] Coussy P. & Moraviec A. (2008) High-Level Synthesis: From Algorithm to Digital Circuit. Springer + Business Media B.V., Dordrecht, Netherlands, 297 p.

[2] Cong J., et al. (2011) High-Level Synthesis for FPGAs: From Prototyping to Deployment. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 30(4), 473-491 p-

[3] Intel FPGAs Resource Center (accessed 14.2.2019) URL: https://www.intel.com/content/www/us/en/products/programmable/fpga/new-to-fpgas/resource-center/overview.html

[4] Farooq U., Mehrez H. & Marrakchi Z. (2012) Tree-based Heterogenous FPGA Architectures. Springer Science + Business Media, New York, 186 p.

[5] Intel (2019) Stratix IV Device Handbook (Accessed 14.2.2019) URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-iv/stratix4_handbook.pdf

[6] Coussy P., Gajski D. D., Meredith M. & Takach A. (2009) An introduction to High-Level Synthesis. IEEE Design & Test of Computers, 26(4), 8-17 p.

[7] Intel HLS Compiler Product Brief (Accessed 14.2.2019) URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/products/hls/hls-production-brief.pdf

[8] Intel High Level Synthesis Compiler User Guide (Accessed 14.2.2019) URL: https://www.intel.com/content/www/us/en/programmable/documentation/ewa1457708982563.html

[9] Intel HLS Compiler Reference Manual (Accessed 14.2.2019) URL: https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html

[10] Avalon Interface Specification (Accessed: 18.2.2019) URL: https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html

[11] Intel HLS Compiler Best Practices Guide (Accessed: 4.3.2019) URL: https://www.intel.com/content/www/us/en/programmable/documentation/nml1505158467345.html