OULUN AMMATTIKORKEAKOULU

Jaakko Ropponen

FEASIBILITY OF USING HIGH-LEVEL SYNTHESIS IN FPGA DESIGN

Evaluating the Capabilities of Intel High-Level Synthesis Compiler

FEASIBILITY OF USING HIGH-LEVEL SYNTHESIS IN FPGA DESIGN

Evaluating the Capabilities of Intel High-Level Synthesis Compiler

Jaakko Ropponen
Bachelor's thesis
Autumn 2021
Information technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Device and Product Design

---

Author: Jaakko Ropponen
Title of the thesis: Feasibility of Using High-Level Synthesis in FPGA Design – Evaluating the Capabilities of Intel High-Level Synthesis Compiler
Thesis supervisor: Timo Vainio
Term and year of thesis completion: Autumn 2021 Pages: 59 + 5 appendices

---

Field Programmable Gate Arrays (FPGA) have become vital in high-performance Digital Signal Processing (DSP) applications in embedded systems, but the development process of application-specific hardware is long and requires expertise on FPGA design and Hardware Description Languages (HDL). In recent years, High-Level Synthesis (HLS) has risen in popularity. It shortens the development time and simplifies the design work significantly by increasing the amount of abstraction between written code and the resulting hardware.

The aim of this thesis was to evaluate the feasibility of using HLS as an alternative to traditional, Register Transfer Level (RTL) FPGA design. Since Intel is one of the world's largest FPGA manufacturers along with Xilinx, studying the capabilities of the Intel HLS Compiler was chosen as the primary subject of the study.

To perform an accurate comparison between RTL and high-level FPGA design, two nearly identical components for performing a matrix-matrix multiplication were developed. The other component was developed using the Verilog hardware description language and Intel Quartus FPGA design tools, while the other component was developed using C++ and Intel HLS Compiler. Then, the resource consumption and performance of the two different component implementations were compared. To provide more versatile results, comparison data was collected with different input data bit widths and different number of parallel multiply-accumulate (MAC) operations.

It was discovered that in most cases, the HLS implementation used a lot more resources than the RTL implementation. With parameter values that the RTL implementation was optimised for, the results of the HLS implementation were significantly worse. On the other hand, the HLS implementation provided more consistent performance results with different parameters and even came ahead of the RTL implementation in some cases.

It was concluded that while traditional FPGA design can achieve superior results in terms of both performance and resource consumption, the implementations might only be optimised for a narrow use case. Optimising an RTL design written with an HDL for different applications can take a significant amount of time and effort. Therefore, the Intel HLS Compiler may be an appropriate choice for projects that require a quick development of a medium-performance DSP component. The Intel HLS Compiler was deemed not suitable for implementing control logic or for very high-performance applications, such as high-throughput physical layer components.

Keywords: Field-Programmable Gate Array, High-Level Synthesis, Digital Signal Processing

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tieto- ja viestintätekniikka, Laite- ja tuotesuunnittelun suuntautumisvaihtoehto

---

Tekijä: Jaakko Ropponen
Opinnäytetyön nimi: Feasibility of Using High-Level Synthesis in FPGA Design – Evaluating the Capabilities of Intel High-Level Synthesis Compiler
Työn ohjaaja: Timo Vainio
Työn valmistumislukukausi ja -vuosi: Syksy 2021 Sivumäärä: 59 + 5 liitettä

Ohjelmoitavat järjestelmäpiirit (FPGA) ovat suosittuja sulautetuissa järjestelmissä, joissa tarvitaan tehokasta digitaalista signaalinkäsittelyä. Suunnittelutyö on kuitenkin hidasta ja siinä tarvitaan laajaa tietämystä laitteistonkuvauskielistä (HDL) ja ohjelmoitavista järjestelmäpiireistä. Viime vuosina suosiotaan kasvattanut korkean tason synteesi (HLS) pyrkii helpottamaan FPGA-kehitystyötä. Se yksinkertaistaa ja nopeuttaa kehittämistä lisäämällä abstraktiota kirjoitetun koodin ja sen pohjalta luodun laitteiston välillä.

Tämän opinnäytetyön tarkoituksena oli arvioida HLS:n käyttämistä FPGA-suunnittelussa perinteisen rekisterinsiirtotasolla (RTL) tapahtuvan suunnittelun sijaan. Arvioitavaksi HLS-työkaluksi valittiin Intel HLS Compiler, sillä Intel on Xilinxin rinnalla yksi maailman suurimmista FPGA-valmistajista.

Jotta RTL- ja HLS-suunnittelua voitiin vertailla todenmukaisesti, molemmilla suunnittelutavoilla luotiin mahdollisimman samankaltaiset komponentit, joista kumpikin suorittaisi kahden matriisin välisen kertolaskun. RTL-suunnittelussa käytettiin Verilog-laitteistonkuvauskieltä sekä Intel Quartus -kehitystyökaluja, kun taas HLS-suunnittelussa käytettiin C++-ohjelmointikieltä ja Intel HLS Compiler -käännöstyökalua. Kehitystyön päätyttyä komponenttien resurssienkäyttöä sekä suorituskykyä vertailtiin keskenään. Niille syötettävän datan leveyttä biteissä ja rinnakkaisten pistetulo-operaatioiden määrää vaihdeltiin, jotta tuloksia saataisiin mahdollisimman monipuolisesti.

Tuloksia tarkastelemalla huomattiin, että HLS-komponentti käytti useimmissa tapauksissa enemmän resursseja. Tietyillä, RTL-komponentille otollisilla parametreillä HLS-komponentti hävisi huomattavasti sekä resursseissa että suorituskyvyssä. Toisaalta HLS-komponentin suorituskykyä kuvaavat tulokset heittelivät huomattavasti vähemmän verrattuna RTL-toteutukseen ja joissain tapauksissa HLS-toteutus jopa voitti suorituskyvyssä.

Lopulta pääteltiin, että perinteisellä FPGA-suunnittelulla voidaan kehittää huomattavasti suorituskykyisempiä ja resursseja vähemmän kuluttavia komponentteja, jotka saattavat kuitenkin olla optimoituja vain tiettyihin käyttötarkoituksiin. Laitteistonkuvauskielellä kirjoitetun komponentin uudelleenoptimointi ja muokkaus voi olla hyvin haastavaa ja hidasta. Tulosten perusteella todettiinkin, että Intel HLS Compiler saattaa sopia projekteihin, joissa on tarve kehittää signaalinkäsittelyyn tarkoitettu komponentti mahdollisimman nopeasti ja joissa suorituskyvyn sekä resurssien suhteen voidaan tehdä kompromisseja. Todettiin myös, että Intel HLS Compiler ei sovi ohjauslogiikan toteuttamiseen tai todella suurta suorituskykyä vaativiin sovelluksiin.

Asiasanat: Ohjelmoitava järjestelmäpiiri, korkean tason synteesi, digitaalinen signaalinkäsittely

ABBREVIATIONS

AC	Algorithmic C
ALM	Adaptive Logic Module
ASIC	Application Specific Integrated Circuit
CE	Clock Enable
CLI	Command Line Interface
DSP	Digital Signal Processing
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
IC	Integrated Circuit
II	Initiation Interval
IP	Intellectual Property
LAB	Logic Array Block
LUT	Lookup table
MAC	Multiply-Accumulate
MLAB	Memory Logic Array Block
MM	Memory-Mapped
MUX	Multiplexer
QoR	Quality of Results
ROM	Read-Only Memory
RTL	Register Transfer Level
SRAM	Static Random Access Memory
ST	Streaming

CONTENTS

1 INTRODUCTION

1.1 Background

1.1.1 High Level Synthesis

High Level Synthesis (HLS) is a method of increasing abstraction between written code and the hardware generated from it. Traditionally, Field Programmable Gate Arrays (FPGAs) have been programmed using Hardware Description Languages (HDLs). HDLs have a high learning curve especially from a software developer's point-of-view, since HDLs are used to describe the actual physical hardware instead of software instructions. HLS allows the designer to write the component functionality using a high level language, such as C++, or even a graphical environment, such as LabVIEW, and the HLS tools will generate the HDL code which can be synthesised into the hardware required to perform the functionality. The main goal of HLS is to simplify and accelerate the FPGA design process, while also bringing down the cost of development. (1, p. 1; 5.)

Different HLS tools have been available for several years and some tools were already released in the early 2010's (1, p. 2). The leading FPGA manufacturers, Intel and Xilinx, have also developed their own HLS tools: the Intel HLS Compiler, the Xilinx Vivado HLS and the Xilinx Vitis HLS (2; 3; 4). As the number of different tools increases along with their maturity, a study on the feasibility of using HLS alongside or in place of traditional FPGA design is in order.

1.1.2 Etteplan

This thesis work was commissioned by Etteplan Embedded Finland. Etteplan is a global engineering company which provides software, embedded, industrial and plant engineering solutions to customers. Among embedded solutions development Etteplan also offers FPGA design. (7.)

The curiosity towards further FPGA design abstraction has increased within Etteplan, too. Reducing the development time and the number of designers required for FPGA development can be advantageous, especially if HLS reduces or removes the need for in-depth knowledge of FPGAs. Some successful trials of using HLS have already been made by Etteplan, but a more

comprehensive study and documentation of some of the tools and their capabilities was requested. (8.)

1.2 Scope and Objectives

This thesis focuses only on the Intel Quartus FPGA design environment and the Intel HLS Compiler due to the limited time available for the study. Furthermore, the Intel HLS Compiler has not yet been thoroughly studied by Etteplan, making it a viable subject for this thesis.

The objective is to study and document the possibilities of using the Intel HLS Compiler to develop FPGA designs by comparing the Quality of Results (QoR) of a component designed with a traditional FPGA design flow using an HDL and Intel Quartus Prime, and the QoR of a component designed with a high level design flow using C++ and the Intel HLS Compiler. The results of the study should not only help designers choose between a traditional FPGA design flow and HLS, but also to help them decide if the Intel HLS Compiler is the right HLS tool for their use case. Internal documentation produced by the study will also include instructions and recommended practices of the Intel HLS Compiler.

1.3 Structure of Thesis

After the introduction, chapter 2 explains the physical structure and operation principle of FPGAs as well as introduces HDLs and describes how they can be used to program FPGAs. Also, the FPGA design verification process and its importance is explained. Chapter 3 introduces the design tools used in this thesis. In chapter 4 the development process of both design flows is presented and the design decisions are explained. In chapter 5 the results of the study are presented and evaluated. Finally, in chapter 6, the thesis is concluded with final thoughts of the entire work process and the usefulness of the results discovered by this study.

2 FPGAS

An FPGA is a digital integrated circuit (IC) containing programmable logic cells and interconnects that connect the logic cells to each other. By configuring these cells and interconnects, an FPGA can form a vast number of different digital circuits, limited by the available resources in the FPGA and how those resources can be linked together. (9, s. 1–2.)

FPGAs are often presented as an alternative to Application Specific Integrated Circuits (ASICs), which too are usually designed for a single, manufacturer-specific application. Unlike FPGAs though, ASICs are physically constructed to provide the desired functionality instead of being programmed. Compared to an ASIC, an FPGA offers significantly lower cost and time-to-market, however with the cost of size, possible complexity and performance. The major downside of ASICs is that they cannot be modified after production, which means that any bugfixes or updates cannot be performed on existing devices. Most FPGAs on the other hand are reprogrammable and could even be updated in devices already deployed to customers. FPGAs are therefore optimal for developing smaller batches of custom digital circuitry where the very highest performance of ASICs is not necessary. FPGAs can also be used to prototype ASICs, but the low prices and increasing complexity of FPGAs make them a compelling option even for final products. (9, s. 2–4.)

2.1 Physical Structure

There are several existing technologies for implementing the underlying memory elements of the configurable portions of FPGAs, but the majority of modern FPGAs are based on Static Random Access Memory (SRAM). SRAM is reprogrammable but volatile, which means FPGAs based on SRAM will lose their configuration on power-down. It is therefore necessary to store the configuration on an external device, such as on-board memory, in order to reprogram the FPGA on power-up. (9, p. 14–15.)

The FPGA series targeted in this thesis is the Cyclone V from Intel, and the following sections will show examples of the architecture of this particular device as well as some general examples.

2.1.1 LUTs

There are different ways to implement the configurable logic blocks of the FPGA using the aforementioned memory elements. The two fundamental methods are multiplexer (MUX) and lookup table (LUT) based logic blocks, of which the LUT based implementation is explained here. (9, p. 19.)

LUTs can be used to define the truth table of any logic operation possible for a given number of inputs. For example, a 2-input NAND gate has a truth table shown in TABLE 1, where A and B are the inputs of the gate.

TABLE 1. The truth table of a 2-input NAND-gate

A	B	**Output**
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

This 2-input NAND gate can be implemented with a LUT, as shown in FIGURE 1. This example implementation consists of transmission gates controlled by the inputs A and B of the LUT. The gates with a circle on them are active low, while the gates without a circle are active high. When the gates are active, they pass the logic values from their inputs to their outputs, otherwise their outputs are high impedance and do not affect the signal path their outputs are connected to (10; 9, p. 21). The data controlled by the gates is input from the programmable SRAM cells on the left, which are configured according to the desired logic function (9, p. 20–21).

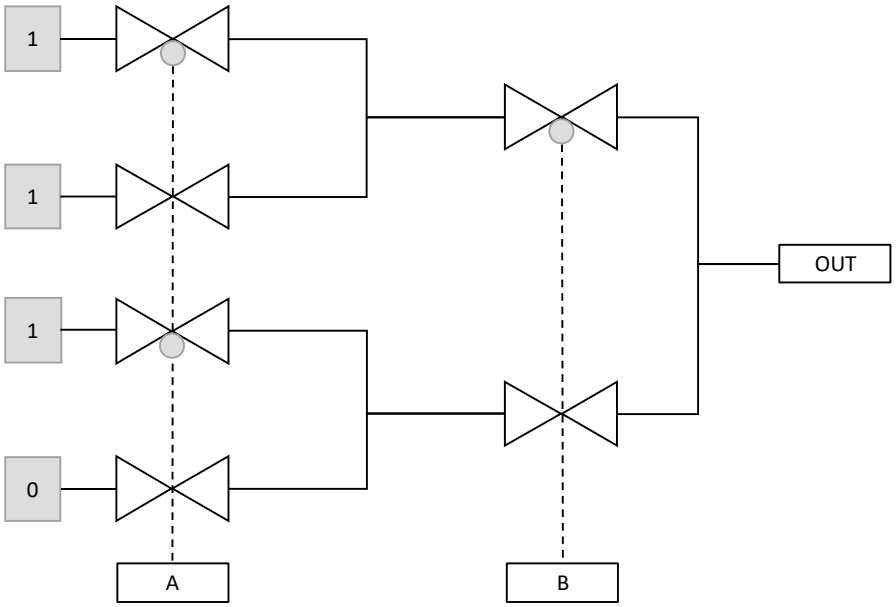

FIGURE 1. A transmission gate-based LUT with SRAM cells on the left, inputs on the bottom and output on the right

By examining FIGURE 1 it can be seen that with this configuration of the SRAM cells on the left, the output is indeed only 0 when A and B are both 1, otherwise the output is 1. This satisfies the truth table requirements of TABLE 1 and therefore implements a 2-input NAND gate. By changing the contents of the SRAM cells on the left, a different logic gate could be defined.

2.1.2 ALMs and LABs

The next level of FPGA hierarchy is what Intel refers to as Adaptive Logic Modules (ALMs). ALMs can contain multiple LUTs, multiplexers, registers, logic gates and even adders. These elements allow ALMs to be used in various different configurations, hence the word ‘adaptive’ in their name. The Intel Cyclone V FPGA uses an ALM structure depicted in FIGURE 2, with a total of 6 LUTs, 2 adders and 4 output registers, among other components. The ALMs in Cyclone V devices can be used in Normal, Extended LUT, Arithmetic or Shared Arithmetic modes by wiring the elements differently. (11.)

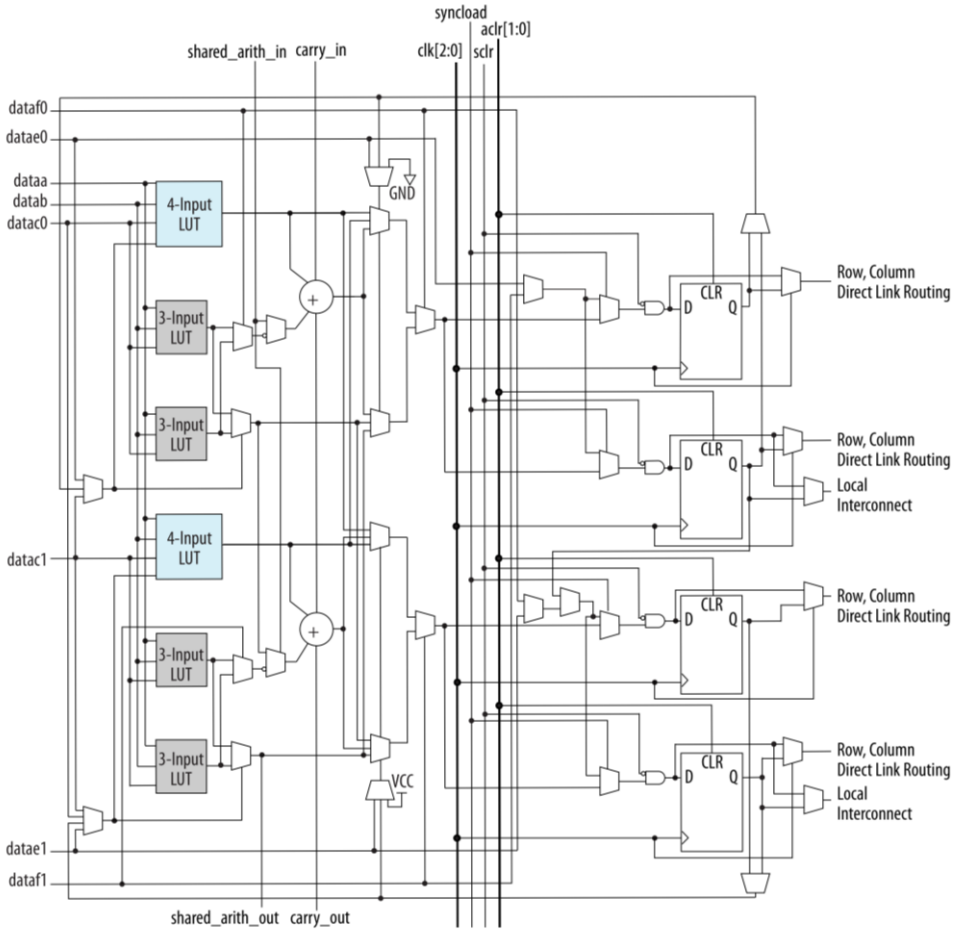

FIGURE 2. The ALM structure in Intel Cyclone V devices (11)

ALMs in turn are confined in Logic Array Blocks (LABs), which have dedicated logic to control the signal flow to the numerous ALMs inside those blocks (11). In Cyclone V devices, one LAB contains 10 ALMs but one LAB can drive the ALMs in adjacent LABs as well, allowing one LAB to control 30 ALMs. (11.)

FPGAs usually allow some of the configurable logic to be used as relatively small memory, commonly referred to as distributed RAM (9, p. 23). In the Cyclone V, a quarter of its LABs can be configured as Memory LABs (MLABs) so that the ALMs within that LAB are used as LUT based memory instead of performing logic operations (11).

It should be noted that different manufacturers use different terms of these levels of hierarchy. Xilinx for example, uses the term Configurable Logic Block (CLB) of their equivalent of LABs, even though there are differences in their structure (9, p. 25). ALMs on the other hand could be compared to Xilinx's Slices, however it can be argued that it is futile, or at least very difficult, to compare these

elements by name, since their definition and resource allocation varies even between device families of the same manufacturer (12; 13; 14; 15, p. 1–4).

2.1.3 DSP Blocks

In addition to the generic logic blocks such as ALMs, many FPGAs contain special digital signal processing (DSP) oriented sections, called DSP blocks by Intel and DSP Slices by Xilinx (9, p. 27; 16; 17). DSP blocks may contain hardwired multipliers and adders which can be used to perform commonly used arithmetic operations quicker and with less resources than would be possible using generic logic blocks (9). Similar to its ALMs, the DSP blocks in Cyclone V devices can be configured differently depending on the desired operation (11).

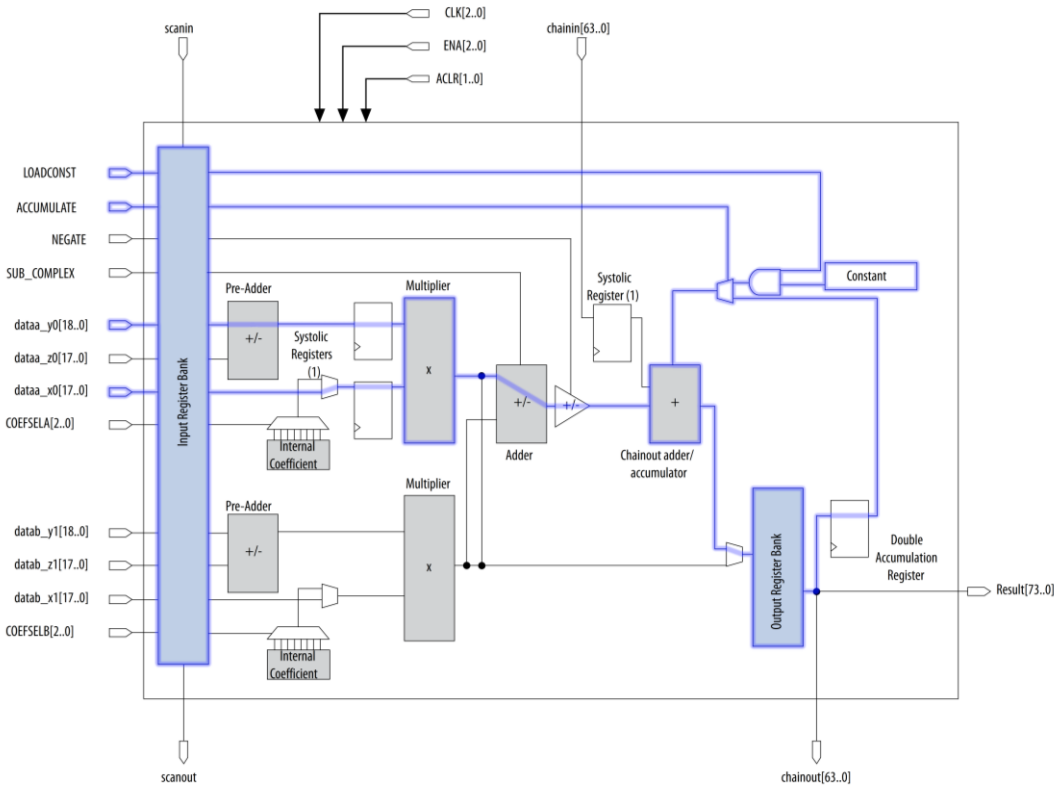

FIGURE 3. The Cyclone V DSP block with MAC operation data path highlighted. Registers and other parts not highlighted are ignored (11.)

One common operation used in DSP is multiply-accumulate (MAC). In a MAC operation, inputs are multiplied together and added to a register which keeps accumulating the result until it is reset. In

Cyclone V devices, there is a dedicated structure in the DSP blocks, highlighted in FIGURE 3, to allow a MAC operation to be performed by a single DSP block, instead of having to route a feedback loop through an ALM and back to the input of the multiplier (11). The MAC operation is important for this thesis, as will be discussed later.

2.1.4 Block RAM

Even though standard logic blocks can be used as memory elements, they are not optimised for area usage and using them as memory is only practical for small, local needs (5). For bigger memories FPGAs include dedicated RAM blocks which can be used to store larger amounts of data efficiently (5; 9, p. 27).

Depending on the device, there can be different ways to configure a RAM block for different applications. In Cyclone V devices RAM blocks can be configured as single-port RAM, simple dual-port RAM, true dual-port RAM, shift registers, read-only memory (ROM) or first in first out data buffers (FIFOs). For this thesis, the most important ones to understand are the single and dual-port RAM modes. In single-port mode, only one read or write operation can be performed at a time and only to one memory address. In true dual-port mode, you can perform any combination of reads and writes to two different addresses at a time. (11.)

2.2 Designing for FPGAs

2.2.1 Hardware Description

HDLs are the traditional method of configuring FPGAs. They were developed to ease the process of designing hardware, as circuitry became increasingly complex and the schematic design flow was no longer viable. Some HDLs have evolved to include parts of nearly all levels of abstraction described below, the most relevant languages of today being Verilog and VHDL. (9, p. 89–100.)

The abstraction of HDLs can be split into 3 different levels as shown in FIGURE 4: the structural, functional and behavioural levels. (9, p. 89.)

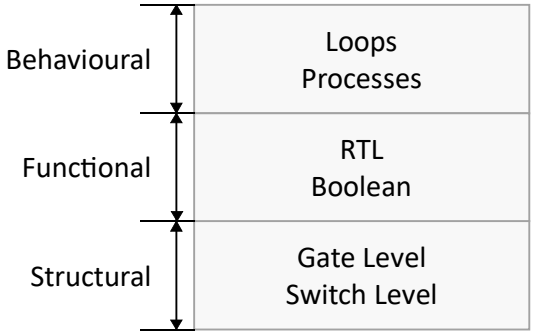

FIGURE 4. Abstraction levels of hardware description

The structural level describes the hardware rather directly and can be split into switch and gate levels. The switch level is the lowest level, which describes how individual electronic components ,such as transistors, are connected. The gate level hides these constructs and depicts the circuits as a collection of primitive logic gates. (9, p. 90.)

The functional level is the next level up, and it includes register transfer level (RTL) representations. To describe a register on the RTL, you only have to specify a condition for a clock signal, inputs and outputs; there is no need to construct the register out of primitive components. The functional level also allows the use of Boolean equations to describe combinational logic, such as multiplexers. (9, p. 90.)

An even further abstraction is the behavioural level, which describes the high level behaviour a design should exhibit. No considerations to wiring, components needed or anything internal to the design are done in a purely behavioural bit of code (19, p. 151–166). A behavioural abstraction could be as simple as using an arithmetic operator to resemble an adder, since on a functional or gate level, a full adder would require several logic gates to be described in order to provide the same functionality (9, p. 91; 18, p. 367 – 370).

The HDL used in this thesis, Verilog HDL, features behavioural, functional and structural levels of abstraction (9, p. 97). In other words, it is possible to instantiate (place into design) individual logic gates in one part of a Verilog design, while on another part, an arithmetic operator can be used to infer a large multiplier structure.

In this thesis, the term RTL is used to refer to the HDL implementation of a component to avoid confusion, even though it may include different levels of abstraction.

2.2.2 Synthesis and Design Considerations

The concept of logic synthesis for FPGAs today essentially means constructing a LUT/CLB (LUT/LAB) level netlist from the HDL code. This netlist can then be further passed into the place-and-route tools of specific FPGA vendors, which will optimise the physical placement of the elements described in the netlist. (9, p. 166–167).

Since the HDL code is synthesised into actual hardware, some consideration needs to be done when writing code to optimise the hardware that is generated. Making false assumptions of the architecture of the FPGA used can lead to a worse performance and more consumed hardware resources. A good example of this is the reset signal. A reset can only be asynchronous on some devices, while others also include synchronous resets. Some FPGA constructs on the other hand might not have any dedicated reset logic at all. Implementing a reset in a way that is not built-in to the register constructs would cause extra reset logic to be generated and could even lead to timing issues. (20, p. 262.)

Another important aspect to consider in FPGA design is timing. Since synchronous logic depends on having valid data in the input of registers at the rising or falling clock edge, it is crucial that data signals arrive on time to the inputs of registers.

To ensure a predictable behaviour, there are two main constraints to meet in synchronous logic; setup and hold time, as depicted in FIGURE 5. Setup time t_{su} is the minimum time the data signal has to be stable before the clock edge in the input of a register. Hold time t_h is the minimum time the data signal has to be stable after the clock edge in the input of a register. If either of these is violated, the output of the register cannot be guaranteed to have the value of the input. (27, p. 3–4; 35).

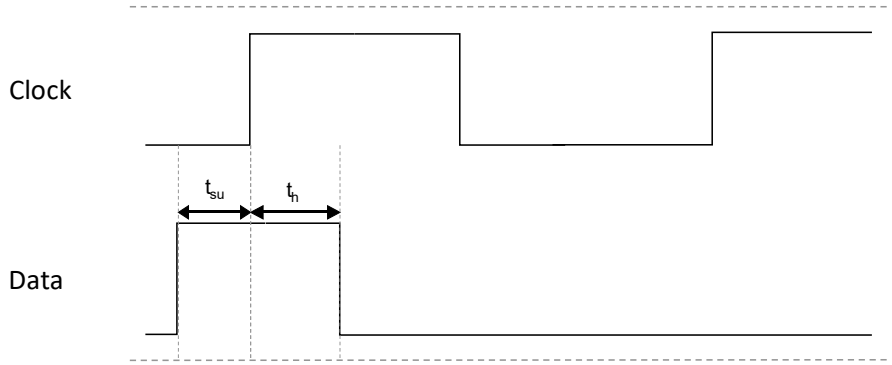

FIGURE 5. Setup and hold times

The process of evaluating the delays in the physical paths of the design is called a timing analysis. In a timing analysis, all the gate and track delays of a path are summed together to determine the total delay of a path between nodes, for example from a register output to the next register's input. Any hold or setup time violations will cause an error in timing analyser tools to ensure the correct behaviour of registers and latches. The largest delay value found during a timing analysis defines the maximum clock frequency at which the design can be guaranteed to work as expected. This frequency is called the f_{MAX}. (9, p. 169–170.)

A design can also have multiple clock domains which allow some of the logic to operate at a greater frequency than the input clock. These different clock domains can be established by deriving a new clock signal from the input clock signal fed through a clock pin on the FPGA, or by having multiple clock signals fed into different clock pins. Although the f_{MAX} is used to represent the maximum frequency of the top-level input clock, it can also be affected by delays in other clock domains derived from that clock. (29.)

2.2.3 Verification and Simulation

Verification is the process of ensuring that the designed component behaves according to its specification. Logic simulation is a part of the verification process and it is often performed on the HDL files of an FPGA design. In simulation, the outputs of the component in response to different stimuli can be observed and debugged. Simulation results can be observed visually as a graph, as a list of results or as only the final fail or pass results, depending on the complexity of the design and the needs of the designer or verification engineer. (21, p. 83–84.)

There are three levels of simulation; RTL, functional and gate level simulations. An RTL simulation only uses the HDL files and does not consider timing as a part of the simulation. A functional level simulation is performed on the netlist generated by the synthesis tool and ensures that the functionality remains the same after synthesis. A gate level simulation is the most accurate and is performed on the gate level netlist files. It takes timing into account by knowing the actual delays between design nodes and therefore allows the detection of possible timing issues. (21, p. 86–87.)

Some HDLs, such as VHDL and Verilog, can be used to create simulation test benches. This is beneficial due to HDL test benches being highly compatible with different simulation programs (21, p. 92). The designer needs to be careful with these languages however, since both of them include elements that can be executed by simulation tools but will not be synthesised into hardware by synthesis tools (20, p. 43; 21, p. 92).

3 DESIGN TOOLS

Several different tools were used in the design process. The most important ones for both design flows are explained in this chapter.

3.1 Intel Quartus Prime

Intel Quartus Prime is the main design software for Intel FPGAs. It is a full development environment which contains nearly all tools necessary to develop for Intel FPGAs, from writing the RTL code to actually programming the device.

Intel Quartus includes several graphical tools to aid the design and verification process. The ones utilised in this thesis are described here.

3.1.1 RTL Viewer and Technology Map Viewer

The RTL Viewer offers a graphical representation of the netlist generated after the Analysis & Elaboration phase, but before synthesis and fitting optimisations. In other words the graphical representation provided by the RTL Viewer is a very close model of the original code written in an HDL, but not the actual hardware implementation. It is a very helpful tool to help visualise the design, especially in case of any issues noticed during the RTL simulation. (22.)

The Technology Map Viewer looks very similar to the RTL Viewer, and it offers a graphical representation of the netlist generated after fitting or Analysis and Synthesis. It includes the fitting optimisations and can be used to review the design schematic as it will be implemented on the FPGA. It does not provide any data on the physical locations of the elements however, which is what the Chip Planner is designed for. (22; 23.)

3.1.2 Chip Planner

The Chip Planner offers a graphical representation of the physical placement of the logic elements on the FPGA. It can be used to display extensive information, e.g. routing details, timing issues and

resource usage. Thus it provides insight on possible routing and placement related issues of a design. It is also useful for examining the actual implementation of a design portion inside ALMs and DSP blocks for any undesired or non-optimal constructs. For example, a synchronous reset could be implemented with separate logic elements instead of an ALM-integrated reset construct, or a DSP block could be placed into a non-optimal configuration. These issues can be identified in Chip Planner and corrected in HDL if necessary. (23.)

3.1.3 Timing Analyzer

Intel provides the Timing Analyzer tool to perform timing analysis and to help debug possible timing issues (27, p. 3). An easy way of finding and improving the worst-case path (path with most delay) of a design is to use the Report Top Failing Paths command and then report the worst-case path. It can be used to go through the worst-case path node-by-node, so that the largest delays in that path can be detected and mitigated.

Another practical feature of the Timing Analyzer is its integration with other Quartus tools such as the Chip Planner. If a path seems properly constructed in HDL code but has a large delay value, Chip Planner can be called from Timing Analyzer to display the physical placement of that path. This way timing issues related to long distances between nodes can be easily detected and solved. (27, p. 31–32.)

Timing Analyzer can also be used to set timing constraints in a Synopsys Design Constraints (SDC) format using a graphical interface. Timing constraints describe how an FPGA design should behave timing-wise and they include things such as target clock frequency, clock uncertainty and external device timing specifications. It is also possible to set false paths to ignore timing analysis for paths that are known not to affect the performance of the final design. (27; 30.)

3.2 Intel HLS Compiler

The Intel HLS Compiler is a High-Level Synthesis tool that generates Intel FPGA optimised RTL code from untimed, algorithmic level C++. It can also be used for creating a test bench in C++ for verification of both the C++ implementation and the RTL implementation of the component. Currently, the HLS Compiler conforms to C++17 and does not support files conforming to newer C++ standards. The Intel HLS Compiler version used in this thesis is 21.2. (2; 33, p. 3, 4.)

3.2.1 Command-Line Interface

The Intel HLS Compiler is mainly operated from a Command Line Interface (CLI) with the `i++` command. The Intel HLS Compiler is command line compatible with the GNU C++ compiler, g++. C++ code written for HLS Compiler can thus be compiled with g++ by defining the paths to HLS Compiler header files for g++. (31, p. 11–13.)

As an example of the CLI, here is the command to compile a design file called 'example.cpp' for a Cyclone V FPGA:

```
i++ -march=”CycloneV” -o “output_name” “example.cpp”
```

After a successful execution of the above command, the compiler will have created an executable file containing the test bench, a debug symbol file for the executable, and a project folder containing reports, RTL files, simulation files and Quartus project files.

The HLS Compiler has several command line options to alter the tool behaviour. The `-ghdl` option for example enables the creation of a waveform file from the RTL simulation, allowing for a visual verification of the behaviour of the component. The `--quartus-compile` option on the other hand automatically performs a Quartus compilation of the component, which allows for more accurate estimations on resource usage and f_{MAX}. Some options affect the implementation of the component, for instance the `--dsp_mode` command option can be used to set the preferred implementation method of math operations on the FPGA. (31, p. 6–10.)

These commands and options can be automated through the use of scripts, but currently no graphical user interface exists for the Intel HLS Compiler.

3.2.2 HTML Report

The Intel HLS Compiler creates an HTML report file which contains information about the resource usage, performance and functionality. The report has several different graphical views to provide insight to the internal operation of the component and it can help the developer to further optimise the design.

The front page of the report lists basic information about the component, such as compile commands, program version numbers as well as the estimated component resource usage and

f_{MAX}. If the `--quartus-compile` option was used, the estimates are automatically transferred from Quartus into the report.

There are more detailed sections in the report, such as the System Viewer. It shows the operations performed by the component as a graph and also provides detailed information about the operations and their actual implementation in the generated HDL code. The report also includes the Area Analysis and Loop Analysis views. The former can be used to find the resource usage of the different sections and even individual operations of the component, while the latter can be used to analyse the performance of loops.

The report also has the Function Memory Viewer section, which is used to visualise the actual implementation of the memories declared inside the component. It can be useful if the developer wants to self-optimise memories by changing their properties using the Intel HLS Compiler's memory attributes. In this thesis the Intel HLS Compiler is relied on to create the optimised implementations of the memory, so the Function Memory Viewer is hardly used.

3.2.3 Verification and Debugging

In source files written for the Intel HLS Compiler, the test bench is contained in the `main` function, while the components are described in functions marked with the `component` keyword. By calling the component functions from the main function, the behaviour and outputs of the components can be verified. There are essentially two verification flows for the Intel HLS Compiler depending on the phase of development.

In the first verification flow the compiler is configured to compile an x86-64 executable of the design for high-level verification. The executable behaves like any other C++ program and can be executed and debugged using any C++ compatible debugger in an x86-64 environment. As the Intel HLS Compiler does not come with a debugger, it is up to the developer to acquire one. High-level verification is efficient, since the compilation of an x86 program only takes a short amount of time. However, high-level verification does not guarantee correct functionality for the RTL implementation of the component. (33, p. 8; 31, p. 7.)

In the second flow, once the functionality has been verified on a higher level, the compiler is instructed to compile for a specific FPGA line-up, such as the Cyclone V. Now the compiler will synthesise the component functions into HDL code, which means that their actual RTL

implementation can be verified. The test bench is still defined in the `main` function, but now an inter-process communication library is used to pass data between the x86 test bench and the RTL simulator. Intel calls this verification method Co-Simulation. (34.)

3.3 ModelSim

ModelSim is an HDL simulation software from Siemens EDA (former Mentor Graphics) (24; 26). Intel has their own adaptation of this tool called the ModelSim - Intel FPGA Edition which they offer with the Quartus design software (25). ModelSim supports mixed-language simulation, which means that it can be used to simulate a design incorporating both VHDL and Verilog (26). It supports RTL and gate-level simulations, so the timing of the optimised design can be analysed as well (26). ModelSim and certain other third party simulation tools can easily be linked to execute an HDL test bench from Quartus using the NativeLink feature (32).

ModelSim has a graphical waveform viewer, which can be used for visual verification of the simulated design. Any signals in any module of the design can be added into the waveform and inspected, allowing thorough debugging and verification. The waveform viewer also has helpful tools such as bookmarks and cursors. Another helpful tool is the signal search, which can be used to find the number of positive or negative edges or certain values of a signal in a specified time region.

The free version of Intel's adaptation of ModelSim used in this thesis is called the ModelSim Intel FPGA Starter Edition. Although it has some limitations compared to the full version, such as decreased simulation performance, it performed well enough for a project of this size. (27.)

4 DESIGN PROCESS

In order to study the differences between HLS and RTL design flows, a suitable design was to be chosen for implementation. As one of the aims was to compare resource usage, the chosen design should utilise DSP and RAM blocks as well as regular ALMs.

Multiplication of two matrices was deemed suitable for this purpose, since it requires the use of DSP blocks for multiplication and RAM blocks for storing large matrices. The multiplication would be done on signed integers, with negative numbers represented as two's complement. Fixed-point numbers were chosen for this thesis instead of floating-point numbers to save development time, due to fixed-point arithmetic being easier to implement.

4.1 Matrix Multiplication

To perform a matrix multiplication operation, the input matrices need to be defined. Take for example two input matrices, A and B. When the matrix A has 2 rows and 3 columns and the matrix B has 3 rows and 2 columns, the multiplication of these matrices can be defined with EQUATION 1.

$$AB = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{bmatrix}$$

EQUATION 1 (37.)

The matrix B must have a number of rows equal to the number of columns in the matrix A, but the column number of matrix B is not limited (36).

Calculating an element in the output matrix is a multiply-accumulate operation, as can be observed from EQUATION 1. Thus, the resource consumption and performance for matrix multiplication on an FPGA will partly depend on how well the MAC operation can be implemented in the FPGAs DSP blocks. The configurability of DSP blocks was described earlier in chapter 2.

4.2 Design Specification

To achieve consistent and comparable results, a high-level specification of the components common to both implementations was necessary before starting the development work. In addition to the matrix multiplication operation, various data flow, control signal and storage (RAM) options had to be considered in order to form a component specification achievable with both design flows.

The main reason for having to create a common specification is the limitations caused by the high level of abstraction of the HLS design flow. For example while HDLs can be used to create virtually any kind of interfaces for components, the HLS Compiler only has a finite number of pre-determined interface options. This limits the specification to conform with the interfaces available in the HLS Compiler.

After studying the possibilities and limitations of the HLS Compiler, it was possible to define the high-level specification as a starting point for the development work. The specification is shown in FIGURE 6.

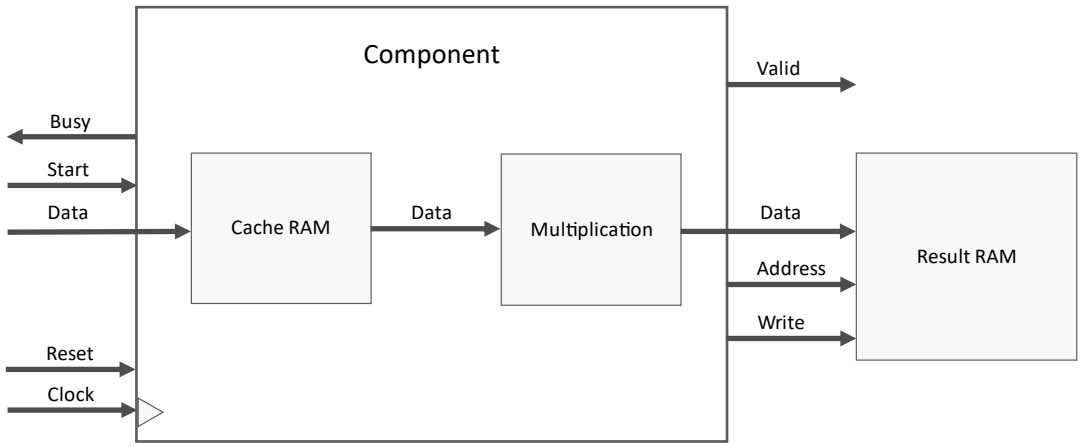

FIGURE 6. The high-level specification of the designed components

For the data flow, a simple streaming interface with handshake signals was chosen. On the input side (on the left in FIGURE 6) it consists of busy, start and data ports. The start signal set high tells the component that there is valid data on the data input port. The busy output signal, when high, tells the upstream component that the component is not ready to receive data and it will not accept

new data even if start is raised. On the output side, the valid signal is high when new valid data has been written into the result RAM.

Since the elements of the input matrices are accessed multiple times during the multiplication process, a temporary cache memory is needed to store the matrices for the duration of the operation. The cache RAM would be managed by the component and would not be accessible from the outside. The results on the other hand were to be stored in RAM outside the component, accessible to other components in the system.

As for the multiplication, it was decided that the component would perform the multiplication on fixed-size matrices of a certain data width known at compile time. For the sake of simplicity the component would perform multiplication only on matrices where the number of rows in the first matrix equals to the number of columns in the second matrix, and the number of columns in the first matrix equals to the number of rows in the second matrix.

4.3 RTL Implementation

4.3.1 Design

The Verilog implementation of the matrix multiplier was written first. Since designing with an HDL requires more elaborate design work on the internal functions of the component, a more detailed specification was crafted to better perceive the necessary constructs.

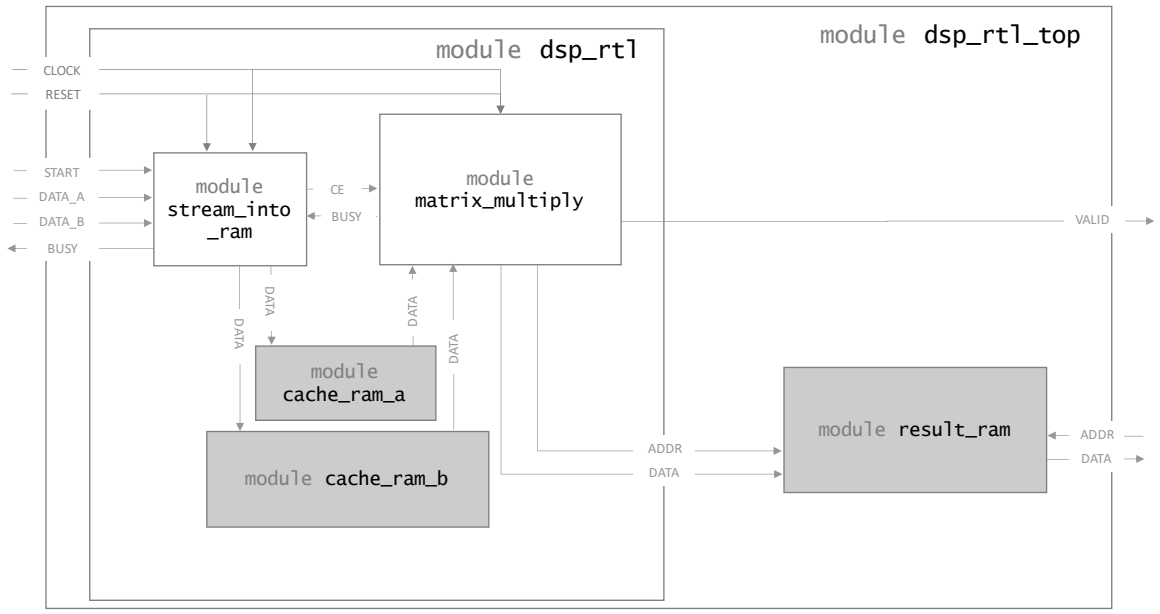

FIGURE 7. Initial specification of the Verilog component

Verilog designs are enclosed in modules. Each module can contain any amount of logic and can instantiate (place into design) other modules as well, creating a hierarchical structure. As can be seen from FIGURE 7, there are four main modules in this design in addition to the three RAM modules. The top-level `dsp_rtl_top` and the `dsp_rtl` modules merely act as containers for connecting the other modules to each other and contain very little logic, which is why they are not thoroughly explained here.

Streaming to Cache RAM

The design was started from the `stream_into_ram` module. The purpose of this module was to stream the incoming matrices into the cache RAMs, element-by-element. It would also be responsible for stalling (preventing from feeding data) the upstream component for the duration of the multiplication after receiving all the input matrix elements. Implementation of the `stream_into_ram` module was rather simple with a small amount of logic for the data flow control and a counter for addressing the RAM.

Both of the cache RAMs and the output RAM were implemented with Intel's own 2-port RAM Intellectual Property (IP) cores. They allow the designer to include device-optimised functions in their design by only defining some configurable parameters (38). The IP core implementation was

chosen for the memories because the method of RAM implementation was not the primary subject of the study, instead a more higher-level perspective of embedded memory usage was needed. Furthermore, the use of IP blocks cut the development time significantly.

The Verilog code of the `stream_into_ram` module is listed in Appendix 1.

Matrix Multiplier

Most of the design effort went into writing the `matrix_multiply` module, since it contains all of the arithmetic logic, the majority of control logic and output RAM addressing. To better understand the constructs needed, a more detailed specification of this module was created (FIGURE 8).

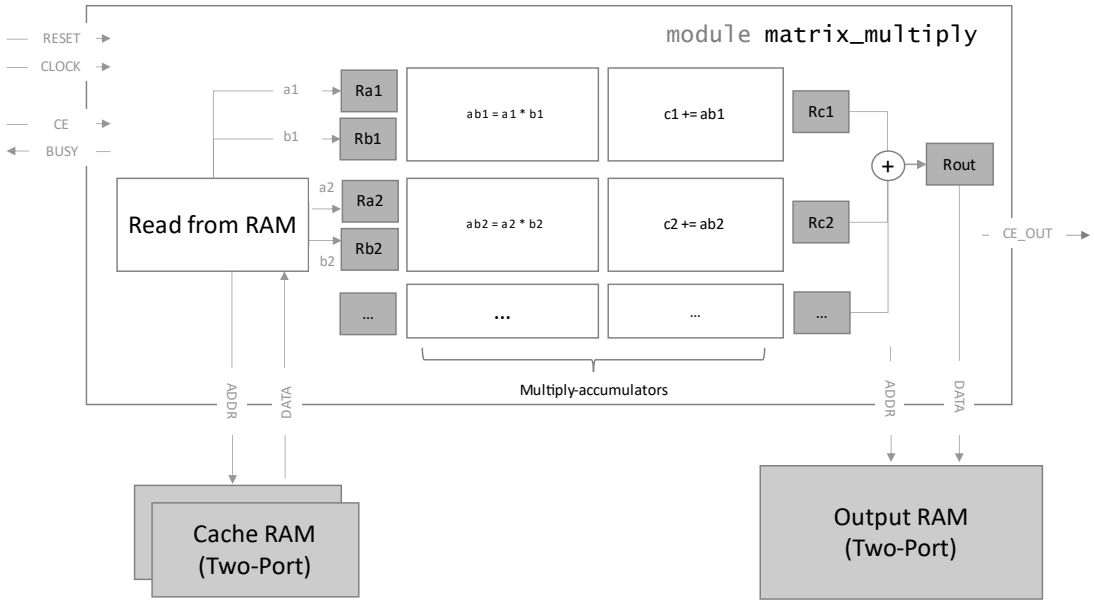

FIGURE 8. A detailed specification for the matrix_multiply module

The `matrix_multiply` module, listed in appendix 2, is triggered by the Clock Enable (CE) signal which indicates that there is new data to be read at the next input RAM address. If CE is set low during data feeding, the multiplication operation will stop and will resume as soon as CE is set back high. Also, since the input matrices have to be read several times during the multiplication, the `matrix_multiply` module must stall the streaming module from overwriting the data in the cache RAMs until it has finished the multiplication process. This is what the BUSY signal is for. It will be

raised high for the duration of the multiplication process and once it is set low, the module is ready to perform another multiplication. The BUSY signal is connected to the `stream_into_ram` component, which in turn will tell the upstream component not to feed new input matrices until BUSY is set low. The BUSY signal behaviour can be seen from a waveform capture later in FIGURE 12.

A special feature of the component is parallel MAC operations. It allows the simultaneous calculation of multiple terms of an output matrix element, allowing the output to be generated in less clock cycles. In FIGURE 8 the parallel MACs are shown in the middle with their output registers (Rc1, Rc2) leading to an adder which calculates the final output matrix element.

However, there is a slight complication to the parallel MAC operation. The component was designed to start the multiplication process as soon as there is at least one pair of input elements available, allowing a shorter time to complete the operation since the component does not have to wait until all the elements of the input matrices have been stored in cache RAM. This means that when there are several parallel MAC operations taking place at the same time, the data also needs to be input at a faster rate. This was solved by multiplying the input data port widths by the number of parallel MAC operations, allowing the inputs to receive multiple elements of the matrices during one clock cycle.

For instance, if the input data width is 20 bits and there are two parallel MAC operations, then the input data ports have a width of 40 bits. FIGURE 9 shows the hexadecimal representations of the numbers at the input ports data_a and data_b. In the case of two parallel MAC operations (the rightmost waveform of FIGURE 9), the first of the two values that are being fed to the inputs needs to be shifted left by a number of bits equal to the data width, while the second value is placed on the lower, rightmost bits of the input. The values are then internally separated by the `matrix_multiply` module and fed to their respective MAC modules.

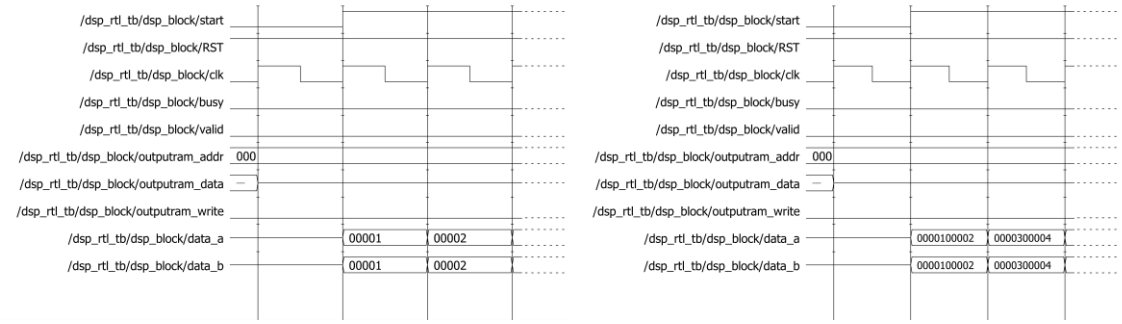

FIGURE 9. The effect of parallel MAC operations to data input port width

The parallel MAC support was implemented because it was known that the Intel HLS Compiler has a similar parallelisation support called loop unrolling. Comparing these two methods was intriguing, since parallel computation is one of the biggest advantages FPGAs can offer. It provides a significant performance benefit compared to purely sequential computation and can greatly reduce data throughput bottlenecks in a system.

Multiply-Accumulators

The MAC operation was enclosed into a separate module called `multiply_acc` to improve modularity. The number of parallel MAC operations in the `matrix_multiply` module is adjusted with the parameter COMP_MULT. The Verilog language includes a structure called `generate`, which makes it possible to instantiate modules several times without creating duplicate code. To generate a configurable number of instances of the `multiply_acc` module, the following `generate` structure was written:

```
// Generate multiplier-accumulators
genvar i;
generate
    for(i = 0; i < COMP_MULT; i = i + 1)
    begin: MACCGEN
        multiply_acc macc(
            .clk(clk),
            .CE(macc_CE_reg),
            .sload(macc_sload_reg[2]),
            .a(in_a_wires[i]),
            .b(in_b_wires[i]),
            .c(out_c_wires[i])
            );
            defparam
            macc.DATAWIDTH = DATAWIDTH,
```

```
                macc.ADDITIONS = FIRSTMATRIXCOLS / COMP_MULT;
    end
endgenerate
```

The outputs of the generated MACs are summed in another part of code and then fed to output RAM through output registers once the operation is complete. The `multiply_acc` module can be found in appendix 3.

Status and flow control

Data flow and internal status control were implemented as a combination of a Finite State Machine (FSM) and several internal control and status registers within the `matrix_multiply` module. Depending on the state of the FSM, different operations are performed by the component.

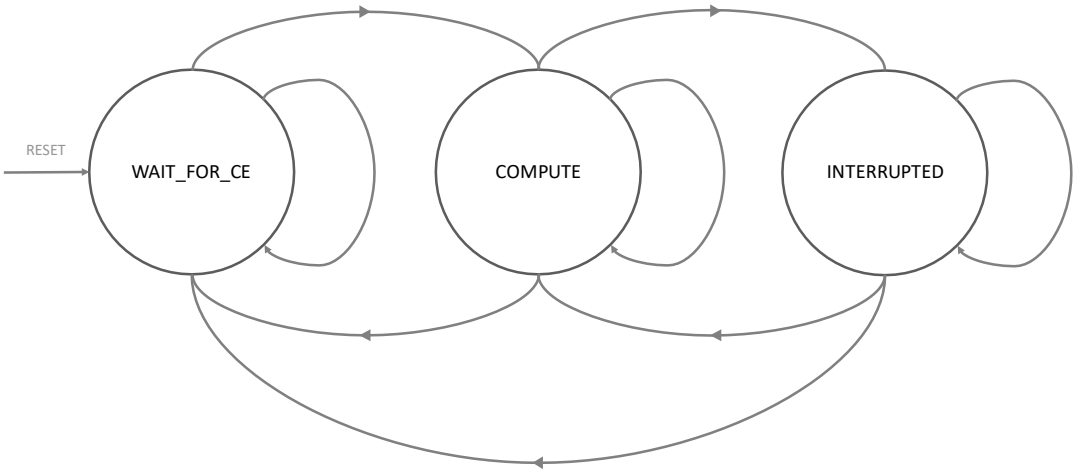

FIGURE 10. Finite State Machine for internal state control

In FIGURE 10, the WAIT_FOR_CE state is the idle state of the component where it is ready to begin a completely new matrix multiplication process as soon as CE is set high. The COMPUTE state is the active state where the component is performing calculations on every clock cycle. INTERRUPTED is the state where data feed has been interrupted before all input matrix elements could be stored in cache RAM. An asserted reset signal or the completion of a matrix multiplication sets the component to its idle state, WAIT_FOR_CE.

As mentioned before, several internal status registers determine the state of the component and the FSM. Certain logic however, such as raising the output RAM write signal if there is valid data to be written to output RAM, is active regardless of the FSMs state.

Determining the right conditions and assuring the correct operation when the data feed is interrupted was not a trivial task and, together with the simulation, it took a large portion of the module's development time.

4.3.2 Verification

Test bench

For verification, a test bench module was written in Verilog. Its purpose was to test not only the validity of the results of the matrix multiplication algorithm, but also to ensure that the component behaves as expected in relation to its control signals. Since it was specified that the data feed could be interrupted by setting the START signal low, the test bench had to ensure the validity of results after such interrupts.

The created test bench module is listed in appendix 4. It performs two matrix multiplication operations consecutively without a reset in between in order to verify the proper self-reset of the internal state of the component. The test bench also interrupts the data feed twice by lowering the START signal: once in the middle of calculations and once at a critical spot where an output element has just been calculated and needs to be written to output RAM. The latter is depicted as a simulation waveform in FIGURE 11.

The component is fed input bit vectors converted from a file containing hexadecimal representations of fixed-point integers and the component output is compared to the known, correct results loaded from a similar file. If the results produced by the component do not match the results loaded from the file, an error message is displayed indicating the index of the mismatched result as well as the expected and gotten values. The expected and gotten results are also written into a file for later inspection, regardless of the validity of the results.

The input and expected output data was generated in Octave software. Octave makes matrix multiplication a single-line operation and was an obvious choice for generating known results in a quick manner (38).

Simulation in ModelSim

Once the test bench had been written it was possible to use it to verify the design. Since the test bench notified of possible false results, it was quite difficult to miss any fatal flaws in the design. Whenever there was an issue, however, the cause of the false results had to be found by manually inspecting the waveform.

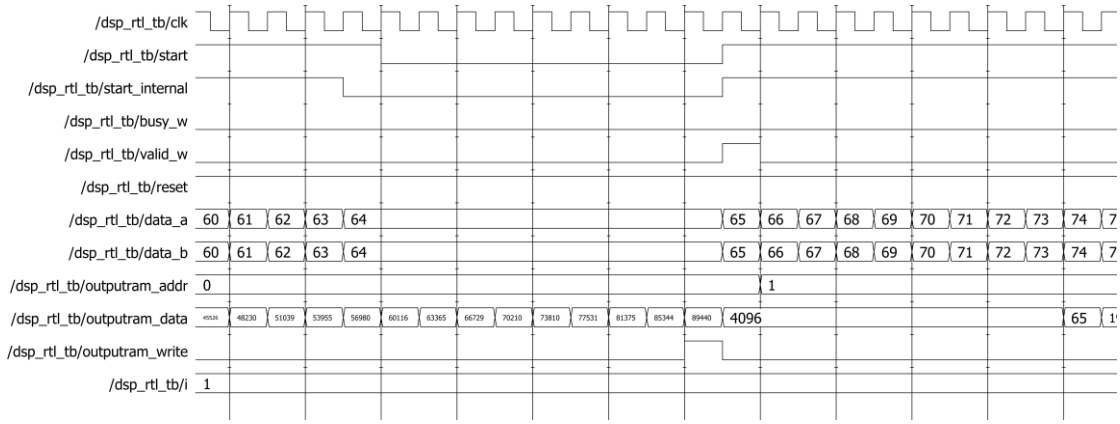

FIGURE 11. A waveform capture of an interrupted data feed before writing to output RAM

A captured waveform of a data feed interrupt is shown in FIGURE 11. As it can be noticed, the component is able to continue receiving data after the interrupt and writes the output into RAM successfully. No errors were printed, so the component worked as expected in this case.

The multiplication operation defined in EQUATION 2 can be seen performed by the component in FIGURE 12, demonstrating the behaviour of the control signals START, BUSY and VALID as well as proving the correctness of the results. In this simulation no interrupts were performed for clarity.

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 7 \\ 2 & 8 \\ 3 & 9 \\ 4 & 10 \\ 5 & 11 \\ 6 & 12 \end{bmatrix} = \begin{bmatrix} 91 & 217 \\ 217 & 559 \end{bmatrix}$$

EQUATION 2.

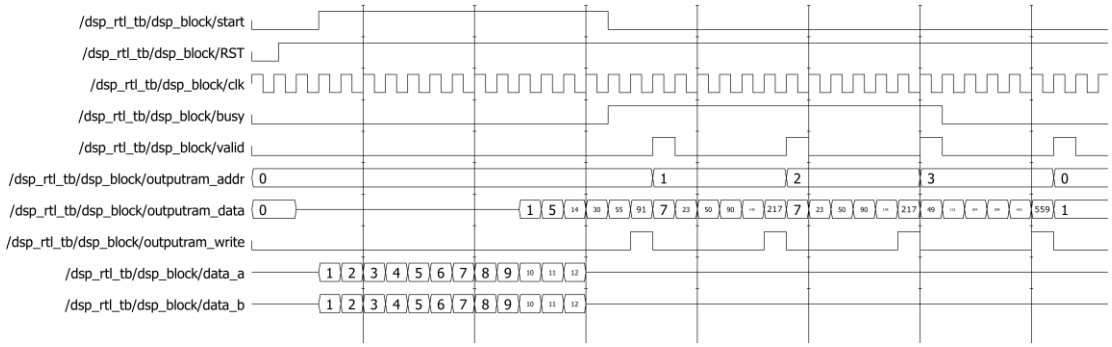

FIGURE 12. A successful multiplication of two matrices with the RTL component

Simulations were performed throughout the development process whenever a new functionality had to be tested. Being able to view signals of any module on any level of a design was essential for debugging any anomalies during development, since the causes could rarely be found by inspecting only the top-level signals.

Optimisation & Timing Analysis

Having initially verified the functionality and behaviour of the component, the design was then further optimised for the resource usage and f_{MAX}. Using the Intel Timing Analyzer together with the Chip Planner, it was easy to locate the bottlenecks present in the design.

One of the major bottlenecks in the beginning was the implementation of the MACs in the FPGAs DSP blocks. As mentioned in chapter 2, the DSP blocks in the targeted Cyclone V device can be configured to perform MAC operations using only one DSP block per MAC. Although functionally correct, the first implementation of the MAC module did not infer the DSP blocks in their correct configuration, leading to excess resource usage and less than ideal f_{MAX}.

The ideal Verilog code for implementing a MAC in the Cyclone V devices was found from Intel's web page. Now the DSP blocks were configured correctly (FIGURE 13), which lead to greatly increased f_{MAX} and only one used DSP block per MAC.

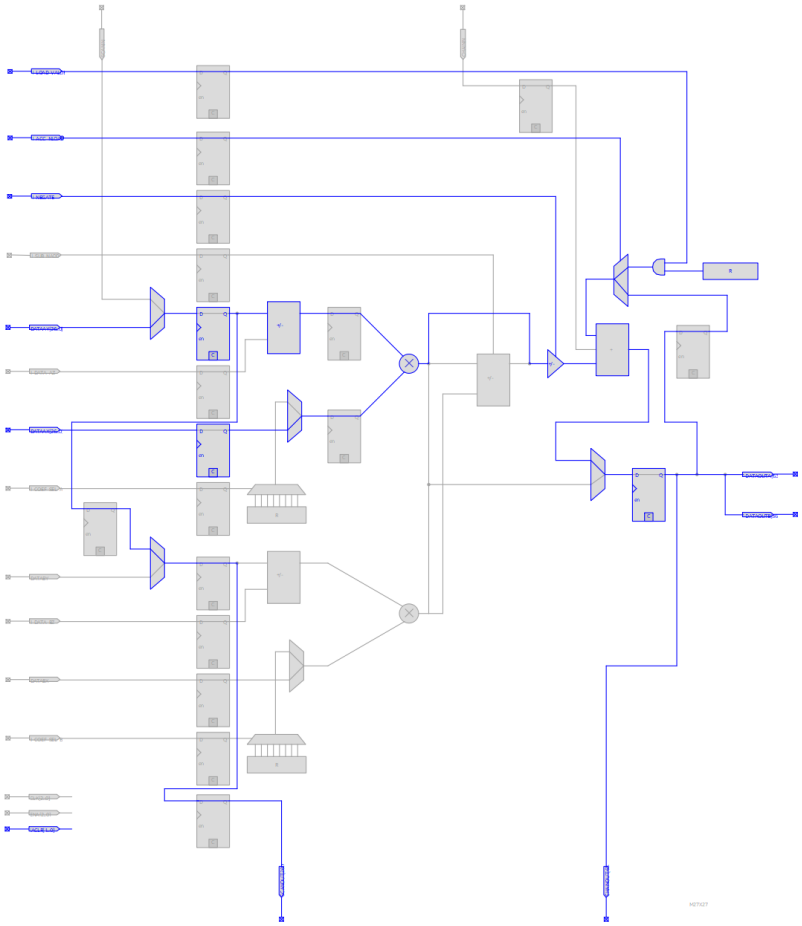

FIGURE 13. A DSP block in its MAC configuration viewed in Quartus

Due to hardware limitations, however, the DSP block MAC configuration is limited to input data widths between 20 and 27 bits (11). To allow the use of the MAC configuration of the DSP blocks with input widths less than 20 bits, a sign extension operation was added into the Verilog code. Since signed integers were used with negative numbers represented as two's complement, the sign extension was performed by padding the extra bits of the extended bit vector with the value of the most significant bit of the original number. For example, if the input width was 16 bits, the 4 extra bits would be filled with the 16th bit of the original number. This preserves the sign of the original number and thus the result remains correct. The sign extension was implemented in Verilog as follows:

```
genvar n;
genvar k;
generate
    for(n = 0; n < COMP_MULT; n = n + 1)
    begin: MACWIREGEN
```

```
        // Assign wires for MACs by separating input elements fr
om cache RAM output registers:
        assign in_a_wires[n][DATAWIDTH - 1:0] = in_a_reg[DATAWID
TH * COMP_MULT - DATAWIDTH * n - 1:DATAWIDTH * COMP_MULT - DATAW
IDTH * n - DATAWIDTH];
        assign in_b_wires[n][DATAWIDTH - 1:0] = in_b_reg[DATAWID
TH * COMP_MULT - DATAWIDTH * n - 1:DATAWIDTH * COMP_MULT - DATAW
IDTH * n - DATAWIDTH];

        // Sign-
extension for MACs if datawidth is less than 20 bits:
        for(k = 0; k < MACC_INPUT_WIDTH - DATAWIDTH; k = k + 1)
        begin: SIGNEXTGEN
            assign in_a_wires[n][DATAWIDTH + k:DATAWIDTH + k] =
in_a_wires[n][DATAWIDTH - 1:DATAWIDTH - 1];
            assign in_b_wires[n][DATAWIDTH + k:DATAWIDTH + k] =
in_b_wires[n][DATAWIDTH - 1:DATAWIDTH - 1];
        end
    end
endgenerate
```

Another major bottleneck was the physical length of the paths between ALMs and DSPs. The long paths increased the delay between registers, which in turn lead to a worse f_{MAX}. The delays were mitigated by adding more registers to the input and output data paths of the MAC module.

4.4 HLS Implementation

4.4.1 Design

The initial High-Level design was written in a relatively short amount of time. The high level of abstraction and not having to deal with control signals made designing the component very time-efficient. No additional low level specification was needed since the HLS Compiler would make the decisions for the underlying hardware structure. There are multiple options available in the HLS Compiler for component interfaces and memory implementation though, and some control regarding the optimisation of the generated hardware is given to the user as well.

Datatypes

Due to C++ being a programming language, it mostly uses consistent data widths for calculations. For example, standard C++ compilers usually perform arithmetic operations on narrower standard datatypes as 32-bit operations, even though an operation of a smaller width would suffice (31, p. 82–83). On an FPGA this would lead to excess hardware being generated unless the promotion is circumvented. Furthermore, it is not necessarily the best choice to use standard datatypes when designing for FPGAs if the values can be stored in smaller containers.

To simplify the implementation of narrower datatypes, the HLS Compiler includes header files that define arbitrary-precision datatypes partly based on Algorithmic C (AC) datatypes. The AC-based datatypes are `ac_int`, `ac_fixed` and `ac_complex`. The HLS Compiler also includes custom floating point datatypes that include mantissa and exponent bit width customisations, but they will not be explained further since they are not utilised in this thesis.

To make the implementation as efficient as possible, the `ac_int` datatype was used in the design. A 20-bit signed integer datatype for example can be declared as follows:

```
typedef ac_int<20, true> fixed_input_t;
```

Any data ports and arithmetic operations using this datatype would be implemented in the minimum data width possible. For example a sum of two `fixed_input_t` numbers would lead to a 21-bit integer, since it is the minimum width the maximum possible result can fit into without overflow. (31, p. 83.)

Interfaces

The Intel HLS Compiler has several options for interfacing with the component. Due to the specified simple streaming interface, there were only two suitable options for the component inputs: the pass-by-value interface and the Avalon Streaming (ST) interface.

A pass-by-value interface is the simplest interface for component invocation in the Intel HLS Compiler. With it, on every clock cycle the input VALID signal is high and the BUSY signal is low, a new component invocation is performed and data is read from the input ports. Below is an example of a pass-by-value interface:

```
component int dsp_hls(fixed_input_t a, fixed_input_t b)
```

The above implementation would lead to top-level interfaces as shown in FIGURE 14. Unlike the 20-bit, arbitrary width input ports, the output port here is for a standard integer datatype leading to a 32-bit wide data path. If the component function was defined to return a number of the `fixed_input_t` datatype, the output would also be only 20 bits wide.

FIGURE 14. Top-level ports of a pass-by-value interface

By comparing FIGURE 14 with FIGURE 6, it can be noticed that this implementation is very similar to the top-level specification and seems suitable for the purpose, with the start signal indicating data validity and the busy signal stalling the upstream component. Since the right-side stall signal is not needed in this case, it can be omitted by placing the `hls_stall_free_return` attribute before the component definition. When simulating this interface option, however, it was noticed that the Initiation Interval (II) of the component was very poor. II indicates that the minimum number of clock cycles there has to be between invocations of a structure, in this case the entire component. The poor II value lead to a slow data input rate, seen in FIGURE 15, which in turn increased the number of clock cycles it would take to complete the matrix multiplication. Output data signals have been removed from the waveform as irrelevant.

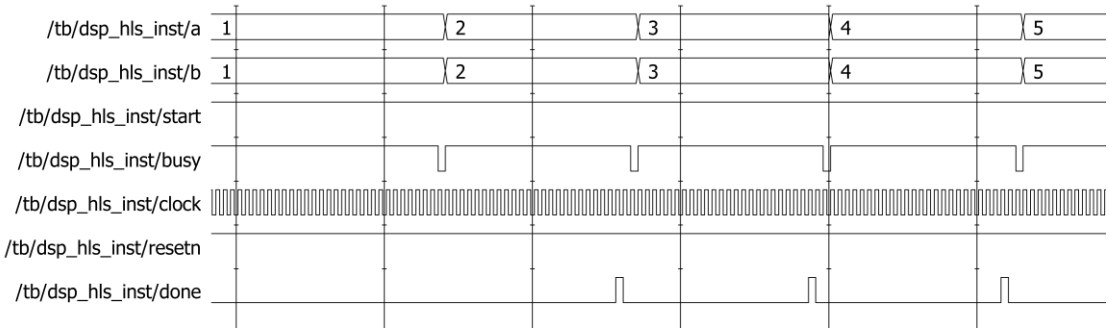

FIGURE 15. Pass-by-value interface data input waveform.

To bypass this issue, another input interface option, called the Avalon ST interface, was considered. The Avalon interface family by Intel defines several different interfaces including streaming, Memory-Mapped (MM) and interrupt interfaces. The Avalon ST interface is suitable for high-bandwidth, low-latency data transfer and it is, at a minimum, controlled by ready and valid signals similar to the busy and valid signals in the component specification in FIGURE 6. (62, p. 4, 40–42.)

The difference in the component behaviour is such that the component start signal only needs to be set high once at the start of the component invocation, and after that, the data validity is indicated by the individual valid signals of the Avalon ST interface ports.

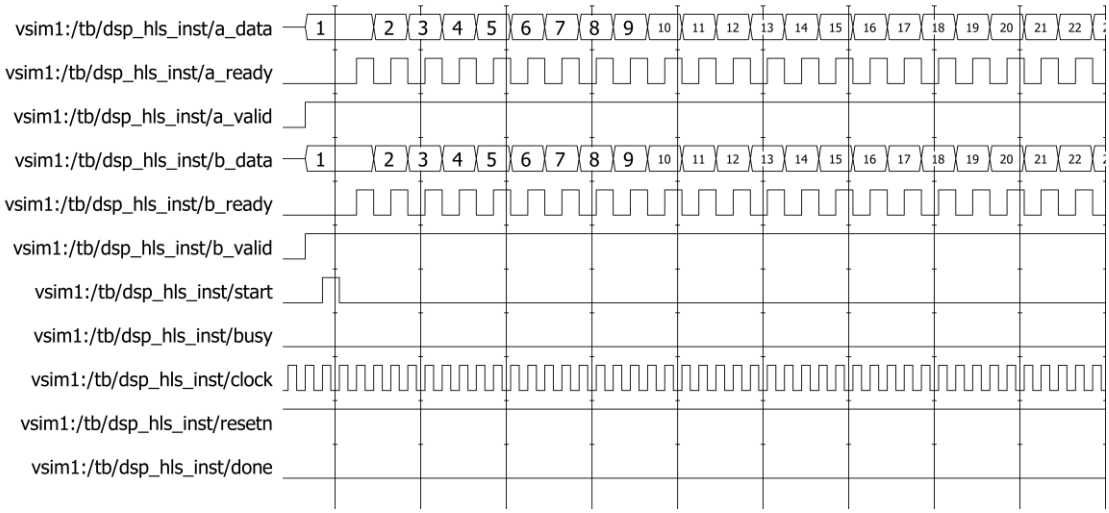

FIGURE 16. Avalon ST interface data input waveform

The waveform in FIGURE 16 shows that the data can be input on every second clock cycle in this case, which provides a significantly better data input rate than the pass-by-value interface. Although

the validity of the input data is now controlled by the separate valid signals, the behaviour is still very similar to the specification.

The component was chosen to output data via an Avalon ST interface as well. An output RAM is also needed to store the results and it would be implemented separately in Verilog along with a counter for addressing the RAM. This was necessary because the Avalon ST interface by itself provides no possibility for addressing the RAM. The final, top-level implementation of the interfaces and output RAM is shown in FIGURE 17.

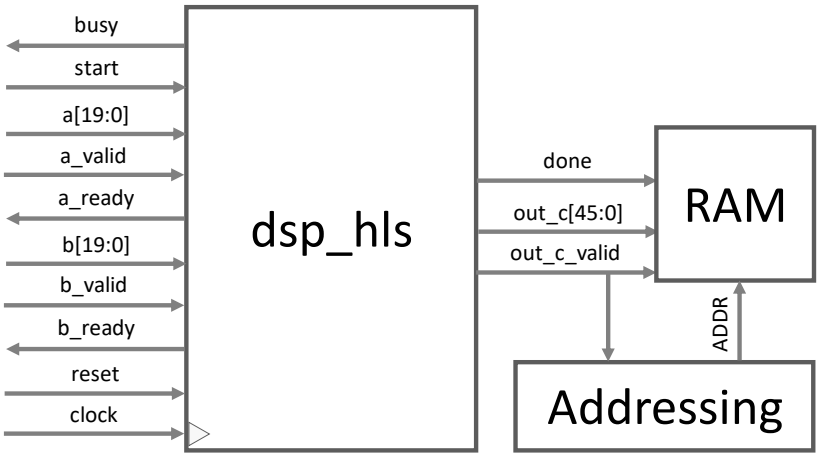

FIGURE 17. The final top-level structure of the HLS component

It would also have been possible to implement an Avalon MM interface to the output and let the HLS Compiler handle the addressing of the RAM. Due to the extra generated control logic and worse performance however, the MM interface was not used.

Although the Avalon ST interfaces fit well to this use case, they exhibit a significant flaw in the current version of the HLS Compiler according to an Intel employee on the Intel Community forums: the width of a symbol on the Avalon ST bus has to be a power of two (42). As the data was to be transferred in one symbol, the symbol width also defines the width of the bus in this application. Thus, the design was forced to use 32-, 64- or even 128-bit wide output streams to fit the results of different widths, increasing the number of logic elements and making the timing closure more difficult due to the increased number of routes.

Memory Implementation

Since the output RAM was chosen to be implemented manually by instantiating an IP block in Verilog, the only memories implemented in C++ were the cache RAMs.

Memory implementation is determined automatically by default. If a large enough array is declared in the component code, the HLS Compiler will place it in the embedded memory blocks of the FPGA. Smaller arrays on the other hand may be implemented as registers in MLABs. This automatic implementation can be bypassed with the `hls_memory_impl` attribute if necessary.

Writing the input matrices to the cache RAMs is done with two nested for loops, with the innermost loop reading the input streams and writing them to their respective memories. Unlike the RTL component, the HLS component does not start the multiplication process before all input matrix elements have been stored in cache RAM. This design decision was made because a code structure, which would have allowed the multiplication to start earlier, would have hindered the ability of the HLS Compiler to optimise the design, especially on the loops' part. Delaying the start of multiplication also eliminates the need to widen the data input ports when using parallel MAC operations and further simplifies the HLS component design.

The Intel HLS Compiler Recommended Practices document states that avoiding nested loops would be beneficial to the performance of the component (41, p. 66). Nested loops with a known iteration count can be automatically coalesced to a single loop, however, with the pragma `loop_coalesce`. Coalescing the following loop brought some performance improvements in the form of increased f_{MAX}.

```
#pragma loop_coalesce
for(uint6 input_row_counter = 0; input_row_counter < (uint6) fir
st_matrix_rows; input_row_counter++){
    for(uint7 input_col_counter = 0; input_col_counter < (uint7)
 first_matrix_cols; input_col_counter++){
        a_mem[input_row_counter][input_col_counter] = a.read();
        b_mem[input_row_counter][input_col_counter] = b.read();
    }
}
```

Manually coalescing the loop was also tried in the form of the code below, but the performance difference was negligible. The nested loop was implemented due to better readability and slightly better performance in the cases recorded in this thesis.

```
uint6 input_row_counter = 0;
uint7 input_col_counter = 0;
while(input_row_counter < first_matrix_rows){
    a_mem[input_row_counter][input_col_counter] = a.read();
    b_mem[input_row_counter][input_col_counter] = b.read();

    input_col_counter++;
    if(input_col_counter == first_matrix_cols){
        input_col_counter = 0;
        input_row_counter++;
    }
}
```

Multiplication Algorithm & Loop Unrolling

Implementing a matrix multiplication operation is rather straightforward in C++. It can be done using three nested loops, with the outermost loop iterating over the rows of the first matrix and the second loop iterating over the columns of the second matrix. The innermost loop iterates both, the columns of the first matrix and the rows of the second matrix. This implementation was found performing best overall, and it was the easiest to optimise using the pragmas supported by the HLS Compiler.

Since C++ is traditionally run in a sequential manner, the parallel nature of FPGAs does not directly fit into the C++ coding style. But as mentioned before, the HLS Compiler has a feature called loop unrolling to help the implementation of parallel operations. It can be used to 'unroll' for loops by a factor known at compile time, causing the execution of some (or all) iterations of the for loop in parallel instead of their normal, sequential execution. In the matrix multiplier, the innermost loop can be unrolled by a known factor to reduce the number of clock cycles it takes to calculate an output matrix element.

The created matrix multiplier structure is shown below. If `comp_mult` would be defined as 2, the structure would have its innermost loop unrolled by a factor of two. Instead of having to perform iterations equal to the number of columns in the first matrix, it performs two loop iterations in parallel per clock cycle, effectively halving the number of clock cycles it takes to write a result into the output stream. In the RTL implementation this is equivalent to the MAC parallelisation written using the `generate` construct of the Verilog language.

```
#pragma loop_coalesce 2
for(uint6 a_row = 0; a_row < (uint6) first_matrix_rows; a_row++)
{
```

```
    for(uint6 b_col = 0; b_col < (uint6) first_matrix_rows; b_co
l++){

        fixed_output_t c = (fixed_output_t) 0;
        #pragma ii 1
        #pragma unroll comp_mult
        for(uint7 count_primary = 0; count_primary < (uint7) fir
st_matrix_cols; count_primary++){
            c += a_mem[a_row][count_primary] * b_mem[b_col][coun
t_primary];
        }

        out_c.write(c);
    }
}
```

The `loop_coalesce` pragma is also used here. The number after it defines how many nested loops it will coalesce, and in this case it will only coalesce the two outermost loops. The innermost loop has also the `ii` pragma. It is used to force the II of the loop to be 1 using pipelining techniques, so that a new iteration of the loop can be started in every clock cycle, even if one iteration would take longer than a single cycle. This behaviour is depicted in FIGURE 18 and FIGURE 19.

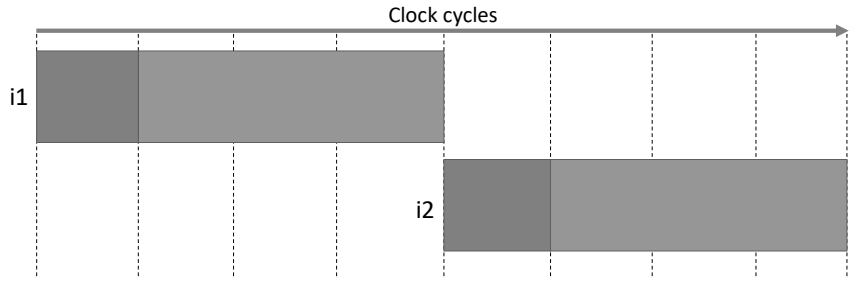

FIGURE 18. A loop which has an II equal to the number of clock cycles an iteration takes

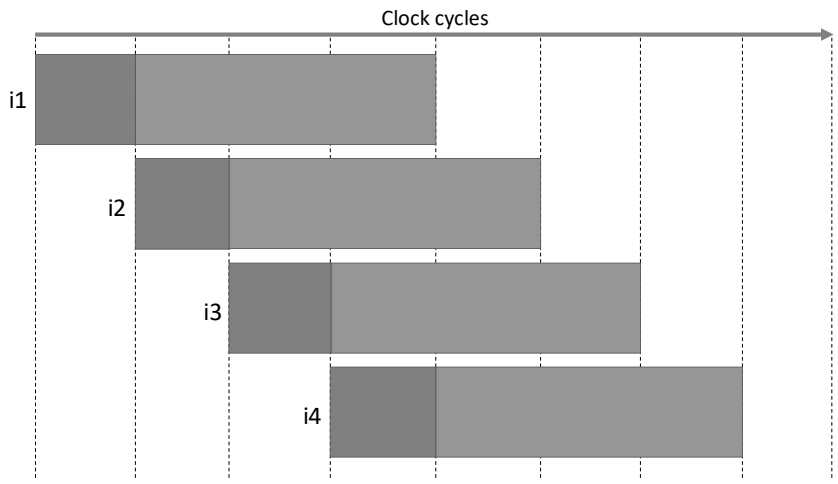

FIGURE 19. A loop with an II of one

FIGURE 18 is an extreme example of a completely unpipelined loop, where the II is equal to the number of cycles it takes to complete one iteration. It is very inefficient compared to the ideal scenario of a pipelined loop shown in FIGURE 19, where a new iteration can be started in every clock cycle even though the previous one has not finished. Although achieving an II of 1 through pipelining uses more resources, the benefits in reduced clock cycles usually outweigh the increased FPGA area usage. The Intel HLS Compiler normally aims for a low II value, but using the II pragma can force the compiler to achieve an II of 1, even with the cost of additional resources or reduced f_{MAX}.

The entire C++ source file with the implementation of the matrix multiplier component is included in appendix 5.

4.4.2 Verification

C++ Test bench

Since the Intel HLS Compiler builds the test bench from the `main` function of the C++ file calling the component function, no separate test bench is needed to test the functionality of the HLS component.

Because very little thought needed to be put into optimisation, the test bench function was rather effortless to write. To calculate valid results, the same matrix multiplication performed by the component is also performed in the `main` function and the results are compared to each other.

Again, if there is a mismatch between the results, an error message would be shown during test bench execution. The test bench function can be found together with the component function in appendix 5.

Simulation

As a result of compiling the source file containing the component and its test bench, an executable file is created. If the compile target is x86-64, the executable performs a high-level simulation and no HDL is generated. Usually this flow was used in the design process whenever the component function would not run or produce correct results. If the high-level simulation worked, the fault could be traced to the FPGA implementation. Running a high-level simulation was quick, in the order of seconds to perform a 32 by 64 matrix-matrix multiplication.

If the compile target is an FPGA, running the executable performs a co-simulation using the test bench defined in the `main` function and the component HDL files generated by the HLS Compiler. If the `-ghdl` option is used in this case, the simulation results will be stored in a waveform file which can be opened and observed with ModelSim.

A waveform capture of the matrix multiplication defined earlier in EQUATION 2 is shown in FIGURE 20. By observing the waveform, it can be seen that the component does not begin the multiplication operation before all input matrix elements have been received and stored, as was intended. There are also other differences compared to the RTL component. As indicated by the a_ready and b_ready signals, the input streaming interfaces seem to take a few clock cycles to be ready to receive data. After that, however, they take in new data every clock cycle. In addition, the component start signal only needs to be raised high for one clock cycle, due to the data validity being controlled by the port-specific valid signals. There is also a done signal present which is not included in the RTL implementation. It indicates that the component is done and ready to start a new matrix multiply operation. However, the busy signal does not seem to be used for indicating a running component invocation.

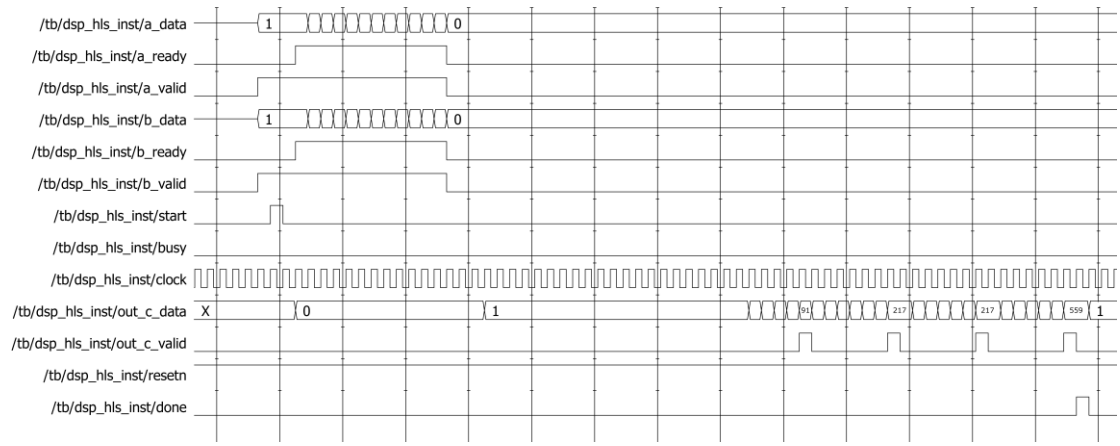

FIGURE 20. A waveform capture of a matrix multiplication performed by the HLS component

The component was also tested in a slightly modified version of the Verilog test bench intended for the RTL component, as the HLS implementation was supposed to be nearly identical in behaviour with the RTL implementation. The modifications adapted the test bench mainly to work with the different start signal behaviour and streaming interfaces, but after making these changes, the component simulated successfully in the Verilog test bench, too.

Report and Optimisation

The report file generated by the HLS Compiler helped debug and optimise the design. It was especially helpful for pinpointing the source of a problem, such as the operation responsible for the excess resource usage.

As an example, the System Viewer representation of the nested loops used to store the input matrices in cache RAM is shown in FIGURE 21. By clicking the nodes in the graph more detailed information can be acquired, like the bit width and latency of the selected operation.

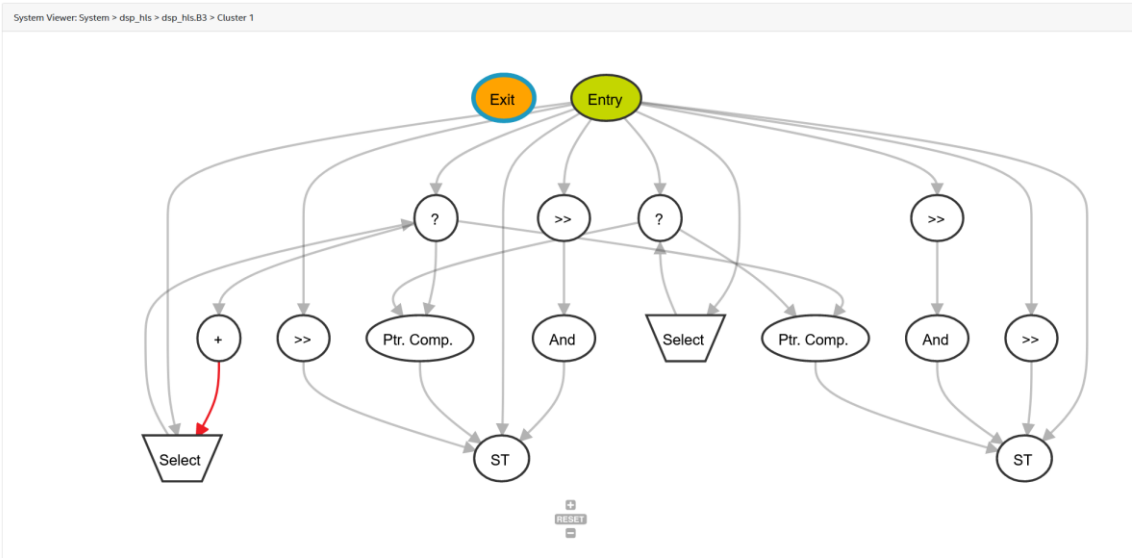

FIGURE 21. System Viewer representation of the loops used to write into cache RAM

The Loop Analysis and Area Analysis of System sections of the report were helpful in finding the resource and performance bottlenecks in the system and thus helped optimise the design.

The poor II of the pass-by-value interface component was troubleshooted with the help of the Loop Analysis view. With the Loop Analysis it was observed that the latency of the operations performed in one invocation of the component was too high to allow a lower II. It is possible that even though the matrix multiplication operation would only be performed once all the input matrix elements had been stored, the HLS Compiler prepares to perform the operation on any component invocation, thus increasing the latency and II. Several optimisation efforts were made to reduce the latency to allow the use of pass-by-value interfaces, but none were effective enough.

There was also a problem relating to the MAC operation generated by the HLS Compiler. While the MAC operation of the RTL implementation was optimised to fit into one DSP block on the Cyclone V, the HLS design seemed to use more than one DSP blocks even when there were no parallel MAC operations and the data width was between 20 and 27 bits. It was confirmed from the Area Analysis page that it was indeed the MAC operation that consumed these DSP blocks. The problem was further studied by viewing the occupied DSP blocks in the Quartus Chip Planner and it was noticed that they were not placed in their MAC configuration at all.

Surprisingly, according to the Intel HLS Compiler Reference Manual, it is not possible to implement a MAC operation using only DSP blocks with anything else but floating point numbers in the Intel

HLS Compiler. For arbitrary precision integers, a combination of DSP blocks and ALMs is used instead. (31, p. 140.)

5 RESULTS AND COMPARISON

Having completed both designs, it was possible to start collecting results of resource usage and performance of different configurations. It was decided that the input matrices would be of a fixed size, with the first matrix having 32 rows and 64 columns, and the second matrix having 64 rows and 32 columns. Only the effect of changing the input data width and the number of parallel MAC operations would be studied, since they were deemed as the most significant factors that affect the performance and resource usage of the components.

Four different input data widths were selected: 16, 20, 27 and 32 bits. Because it was known that the MAC operations can only be implemented inside a single DSP block on the Cyclone V device only if the width is between 20 and 27 bits, 16 and 32 bits were selected outside that range. They also provide more standard bit width references, both being power of two values and 32 bits being common in modern processor architectures.

The number of parallel MAC operations was chosen to be varied among three values: 1, 2 and 4. Having over four parallel MAC operations would consume significantly more resources and thus it was not studied thoroughly.

The resource usage is presented in the results as the number of consumed ALMs, DSPs and RAM blocks (BRAM). Performance on the other hand is presented with 5 different measures. One of them is the maximum component clock frequency, f_{MAX}. Then there is C_{OUT}, which is the number of clock cycles between output RAM writes, and C_{END}, which is the number of clock cycles between feeding the first input elements and the end of the matrix multiply operation.

The most accurate measures of the performance of the component are the output frequency (f_{out}), which depicts the frequency at which results are written to output RAM, and the time to finish (t_{end}), which depicts the time it takes to perform a complete matrix multiplication with the component. Both of these are calculated from the component f_{MAX} and the number of clock cycles it takes to perform the said operations (C_{OUT} and C_{END}).

5.1 Performance Comparison

The performance results of the components with different parameters are presented in different tables according to the input data widths. It should be noted that the input and output data widths represent the width of the data itself, not the width of the input and output streams. In the case of the HLS implementation, the input and output stream widths are rounded to the next largest power of two value that can contain the data. This is due to the previously mentioned limitations of the Avalon ST interface in the current version of HLS Compiler. On the RTL implementation the input and output streams have the same width as the data.

Starting from the results of the 16-bit wide inputs shown in TABLE 2, it can be seen that the RTL implementation is clearly superior to the HLS implementation in all cases. The sign extension of the 16-bit inputs to 20 bits allow the DSP blocks to be used in their MAC configuration, providing good results in performance. While C_{OUT} is virtually the same between the implementations, the C_{END} of the HLS component is slightly worse. This is mainly because of the HLS component having to wait until all input elements have been fed before starting the matrix multiplication process.

TABLE 2. The performance of the components with 16-bit inputs

Type	*Input width*	*Output width*	*MACs*	f_{MAX} *(MHz)*	C_{OUT}	C_{END}	f_{OUT} *(MHz)*	t_{END} *(µs)*
HLS	*16*	*38*	*1*	155.13	65	68625	2.387	442.4
RTL	*16*	*38*	*1*	219.68	64	65545	3.433	298.4
HLS	*16*	*38*	*2*	150.44	33	35857	4.559	238.3
RTL	*16*	*38*	*2*	207.81	32	32777	6.494	157.7
HLS	*16*	*38*	*4*	163.75	17	21522	9.632	131.4
RTL	*16*	*38*	*4*	165.95	16	16393	10.372	98.8

TABLE 3 presents the results of the components with 20-bit wide inputs. The RTL implementation is still ahead of the HLS implementation in all of the cases. The RTL implementation does fall short in f_{MAX} when there are 4 parallel MAC operations, but makes up for it by having a better C_{OUT} and C_{END}.

TABLE 3. The performance of the components with 20-bit inputs

Type	Input width	Output width	MACs	f_{MAX} (MHz)	C_{OUT}	C_{END}	f_{OUT} (MHz)	t_{END} (µs)
HLS	*20*	*46*	*1*	154.11	65	70678	2.371	458.6
RTL	*20*	*46*	*1*	218.25	64	65545	3.410	300.3
HLS	*20*	*46*	*2*	159.36	33	37909	4.829	237.9
RTL	*20*	*46*	*2*	211.91	32	32777	6.622	154.7
HLS	*20*	*46*	*4*	167.98	17	21526	9.881	128.1
RTL	*20*	*46*	*4*	158.55	16	16393	9.909	103.4

When the input width is raised to 27 bits, the HLS component encounters major performance loss when measuring f_{OUT} and t_{END}, as can be observed from TABLE 4. This is because the HLS Compiler failed to achieve an II of 1 for the inner loop of the matrix multiplication despite using the `ii` pragma. The best II value it could achieve was 2, which meant that the value of C_{OUT} doubled and therefore f_{OUT} was halved.

TABLE 4. The Performance of the components with 27-bit inputs

Type	Input width	Output width	MACs	f_{MAX} (MHz)	C_{OUT}	C_{END}	f_{OUT} (MHz)	t_{END} (µs)
HLS	*27*	*60*	*1*	147.73	131	136210	1.128	922.0
RTL	*27*	*60*	*1*	215.33	64	65545	3.365	304.4
HLS	*27*	*60*	*2*	147.23	67	70676	2.197	480.0
RTL	*27*	*60*	*2*	195.69	32	32777	6.115	167.5
HLS	*27*	*60*	*4*	146.99	35	39955	4.200	271.8
RTL	*27*	*60*	*4*	144.53	16	16393	9.033	113.4

However, it was discovered that the HLS Compiler was able to reach an II of 1 for the inner loop by forcing a lower target frequency with the `--clock` option. For this purpose, a target of 100 MHz was used and the results are shown in TABLE 5. Now f_{OUT} and t_{END} of the HLS component are improved, but the performance is still not on par with the RTL implementation due to the lower f_{MAX}.

TABLE 5. The performance of the components with 27-bit inputs when the HLS clock target is 100 MHz

Type	Input width	Output width	MACs	f_{MAX} (MHz)	C_{OUT}	C_{END}	f_{OUT} (MHz)	t_{END} (µs)
HLS	*27*	*60*	*1*	132.05	65	68622	2.032	519.7
RTL	*27*	*60*	*1*	215.33	64	65545	3.365	304.4
HLS	*27*	*60*	*2*	128.95	33	35854	3.908	278.0
RTL	*27*	*60*	*2*	195.69	32	32777	6.115	167.5
HLS	*27*	*60*	*4*	128.49	17	19470	7.558	151.5
RTL	*27*	*60*	*4*	144.53	16	16393	9.033	113.4

TABLE 6 shows the results with 32-bit inputs. Now the RTL implementation suffers from a terrible f_{MAX}, since the DSP blocks can no longer be placed in their MAC configuration due to the input width being over 27 bits. Again, the HLS Compiler was not able to reach an II of 1 with the default settings. This was solved by setting the target frequency to 100 MHz, allowing the HLS design to surpass the RTL design in terms of performance.

TABLE 6. The performance of the components with 32-bit inputs when the HLS clock target is 100MHz

Type	Input width	Output width	MACs	f_{MAX} (MHz)	C_{OUT}	C_{END}	f_{OUT} (MHz)	t_{END} (µs)
HLS	*32*	*70*	*1*	104.57	65	68622	1.609	656.2
RTL	*32*	*70*	*1*	82.89	64	65545	1.295	790.7
HLS	*32*	*70*	*2*	116.33	33	35855	3.525	308.2
RTL	*32*	*70*	*2*	85.57	32	32777	2.674	383.0
HLS	*32*	*70*	*4*	111.96	17	19471	6.586	173.9
RTL	*32*	*70*	*4*	83.56	16	16393	5.223	196.2

Across the results of all data widths, it would appear that the HLS Compiler is good at keeping the f_{MAX} of the component consistent: While the f_{MAX} of the RTL component drops all the way from the highest frequency of 219.64 MHz to the lowest of 82.89 MHz, resulting in a delta of 136.75 MHz, the delta of the best and the worst f_{MAX} of the HLS component was 63.41 MHz and it would have been even lower if the `--clock` option would not have been used with 27 and 32-bit inputs.

5.2 Resource Usage Comparison

In order to evaluate the QoR of the implementations as a whole, the resource consumption was studied as well. In case of the 16-bit inputs (TABLE 7), the HLS component always uses more ALMs than the RTL component, but less DSP blocks when there are 2 or 4 parallel MACs. When cross-comparing these results to the equivalent performance results in TABLE 2, the use of HLS in this case can hardly be justified. However, the performance of the HLS implementation is quite close to the RTL performance when there are 4 parallel MACs.

TABLE 7. The resource usage of the components with 16-bit inputs

Type	Input width	Output width	MACs	ALMs	DSPs	BRAM
HLS	*16*	*38*	*1*	261.6	1	12
RTL	*16*	*38*	*1*	69.0	1	12
HLS	*16*	*38*	*2*	270.2	1	12
RTL	*16*	*38*	*2*	83.0	2	12
HLS	*16*	*38*	*4*	311.7	2	12
RTL	*16*	*38*	*4*	112.5	4	12

With 20-bit inputs the HLS implementation still cannot compete in resources, as can be seen from TABLE 8. Compared to the equivalent results of 16-bit inputs, the resource consumption of the HLS component has nearly doubled in some cases. In the worst scenario, 4 times the DSP blocks and nearly 5 times the ALMs are consumed compared to the RTL implementation. Considering this and the slightly worse performance of the HLS component with this input width, the RTL implementation would be a better choice with 20-bit inputs.

TABLE 8. The resource usage of the components with 20-bit inputs

Type	Input width	Output width	MACs	ALMs	DSPs	BRAM
HLS	*20*	*46*	*1*	399.2	4	15
RTL	*20*	*46*	*1*	71.5	1	13
HLS	*20*	*46*	*2*	441.1	8	13
RTL	*20*	*46*	*2*	98.5	2	13
HLS	*20*	*46*	*4*	611.6	16	13
RTL	*20*	*46*	*4*	129.0	4	13

The situation is not improving with an input width of 27 bits either. As can be seen from TABLE 9, the resource usage is again several times larger in the HLS implementation, except for BRAM. These results are with the default compiler settings and an II of 2 for the inner loop.

With the `--clock 100MHz` option however, resource usage drops significantly for the HLS design as is shown in TABLE 10. This is likely because the HLS Compiler does not need to create any stall logic for the inner loop, which now only takes 1 iteration instead of 2. In addition, if the target frequency is lower, less registers need to be placed into the design to achieve it. Still, the HLS implementation uses up to 3 times more resources than the RTL implementation.

TABLE 9. The resource usage of the components with 27-bit inputs

Type	Input width	Output width	MACs	ALMs	DSPs	BRAM
HLS	*27*	*60*	*1*	509.1	3	18
RTL	*27*	*60*	*1*	85.5	1	18
HLS	*27*	*60*	*2*	585.3	5	18
RTL	*27*	*60*	*2*	115.5	2	18
HLS	*27*	*60*	*4*	705.4	10	18
RTL	*27*	*60*	*4*	154.0	4	18

TABLE 10. The resource usage of the components with 27-bit inputs when the HLS clock target is 100MHz

Type	Input width	Output width	MACs	ALMs	DSPs	BRAM
HLS	*27*	*60*	*1*	255.7	1	18
RTL	*27*	*60*	*1*	85.5	1	18
HLS	*27*	*60*	*2*	270.1	2	18
RTL	*27*	*60*	*2*	115.5	2	18
HLS	*27*	*60*	*4*	312.8	4	18
RTL	*27*	*60*	*4*	154.0	4	18

In TABLE 11, the results of the 32-bit inputs show that the HLS component uses less DSP blocks in some cases, since the MAC operation cannot be implemented with a single DSP block with this data width. The HLS component does still uses more ALMs, but not as substantially in the case of 4 parallel MACs. Unless the non-optimal implementation of the MACs in the RTL design is solved, the HLS design could be a viable option if performance is a priority and there are excess ALMs available.

TABLE 11. The resource usage of the components with 32-bit inputs when the HLS clock target is 100MHz

Type	Input width	Output width	MACs	ALMs	DSPs	BRAM
HLS	*32*	*70*	*1*	301.2	3	21
RTL	*32*	*70*	*1*	142.5	3	21
HLS	*32*	*70*	*2*	330.7	4	23
RTL	*32*	*70*	*2*	231.0	6	21
HLS	*32*	*70*	*4*	430.9	8	23
RTL	*32*	*70*	*4*	376.0	12	21

Curiously enough, the BRAM usage sometimes varies in the HLS implementation with different numbers of parallel MACs, unlike in the RTL implementation. It was noticed from the HTML reports that the HLS Compiler automatically adjusts the widths of the accesses to the cache RAMs according to the number of parallel MACs and this also seems to affect the total number of consumed RAM blocks.

5.3 Overall Evaluation

Looking at the performance and resource usage results, the HLS implementation seems to fall behind the RTL implementation in most cases. However, the f_{MAX} results of the HLS implementation are more consistent with different parameters compared to the results of the RTL implementation. As mentioned before, an II of 1 for the inner loop in the HLS design was achieved with 27 and 32-bit inputs by using the `--clock` option of the HLS Compiler to force a lower target frequency. According to the Intel HLS Compiler Reference Manual though, the HLS Compiler should have tried to achieve the specified II even when the `--clock` option was not used (31, p. 99). It is unsure whether this is a bug or the intended behaviour of the HLS Compiler.

Although the RTL design was parametrised so that no modifications to the modules would be necessary when changing the data width and parallel MACs, wildly different performance results were produced with different parameters. The design could likely be optimised further to better meet the needs of different data widths and different number of parallel MACs. The optimisation would require a significant effort though, and it is possible that different implementations would be required for different parameters. For instance, different Verilog implementations of the MAC

modules would likely have to be used whenever the data width prevents the use of the MAC configuration of the DSP blocks.

The HLS Compiler is better in this regard. While creating a design from the ground up with an HDL achieves far superior results for a narrow set of parameters, the HLS Compiler succeeds at providing consistent performance results even with radically different parameters, albeit at the cost of resources. An example of this is the implementation of the MAC operation: By not using the MAC configuration of the FPGA's DSP blocks on any data width, the HLS Compiler ensures a consistent performance across all data widths. This, however, completely cancels out the benefits from having such specialised embedded constructs on the FPGA. Although it can be speculated that this design decision has been made to ease the synthesis process of the HLS Compiler, a possibility to force the compiler to use a different implementation could be beneficial.

The development of the HLS component took less than half the time it took to develop the RTL component, with the most time being consumed in finding the correct pragmas and compiler options for the HLS Compiler to produce the best results. No separate test bench needed to be coded apart from the high-level one described in the source file itself. The verification of the RTL design on the other hand took a lot of time, more than the development of the component itself. Although some of the development time of both designs was spent in learning the tools and methods, it is clear that the HLS design flow was both easier to learn and quicker to develop with.

A theoretical advantage of HLS is the portability of the high-level code. Optimally, the code written for HLS is pure high-level code with no device-specific optimisations and the HLS tools would perform these optimisations automatically. While RTL code can also be written in a way that allows it to be synthesised for different FPGAs even from different manufacturers, optimising the code for specific hardware can require significant effort.

In this work the C++ code of the component function is rather pure with little optimisations. The only parts of the component code that are specific to the Intel HLS Compiler are the pragmas used in the loops of the component and the Avalon streaming interfaces. However, if there are no similar streaming interfaces available in the targeted HLS tool, the code structure has to be modified. Had the pass-by-value interfaces been used instead, the component function would be almost completely portable with very little modifications.

5.4 Verdict

Having been released as late as 2017, the HLS Compiler is still a relatively new tool on the market and is clearly a work-in-progress: Not all Intel FPGA families can be targeted, only one simulation tool is supported and the HTML report has some views that are stated to be in a beta or alpha state. Also, the Avalon interface symbol width restriction is yet to be fixed, although it was acknowledged as an issue already in 2020 (42). Even if these issues may not be critical for development, it might be worthwhile to research other available tools as well while the Intel HLS Compiler evolves further.

That said, the Intel HLS Compiler may be a useful tool for projects that require swift development of a DSP component, which does not need to have the absolute best performance. Due to the easy implementation of Avalon interfaces, the HLS Compiler could also prove useful in accelerator applications where data processing is offloaded from the CPU to an application-specific component. It is not, however, suitable for implementing control logic due to its interface limitations, nor should it be used in high throughput communication or data processing applications due to its performance limits. In smaller FPGAs and resource-critical applications, the available logic elements would be better utilised by designing components traditionally with an HDL due to the higher resource consumption of the HLS designs.

6 CONCLUSION

The aim of this thesis was to study the feasibility of using the Intel HLS Compiler in FPGA design by comparing the QoR of a design implemented in Verilog to the QoR of the same design implemented with C++ and the Intel HLS Compiler. By defining a common specification, two components with negligible differences in behaviour (caused by the interface limitations of the HLS Compiler) were created. The best-case results of both implementations were then collected and compared to each other in order to evaluate the HLS Compiler.

The study process in its entirety was reasonably challenging. Before beginning the development of either of the components, the limitations and possibilities of the HLS Compiler had to be studied. After all, if the implementations would not provide the same functionality, the comparison between them would not be valid. In addition, the author had not designed for FPGAs this extensively before, only having written a handful of circuits in VHDL. Verilog, timing analysis and some of the design tools were not familiar at all and took some time to learn. Toward the end of the development of the RTL component however, writing circuit descriptions in Verilog and using the design tools had become fluent.

Due to the Intel HLS Compiler being a relatively new tool, not many studies of it exist. Therefore, this thesis could prove useful when assessing whether to use the Intel HLS Compiler for FPGA design or not. For the same reason the results may not be valid for a long time, however, since the QoR is likely to be impacted by future updates. On the other hand, this thesis provides a reference point for possibly comparing a future version of the tool to the current version, 21.2.

As a whole, the thesis work can be considered a success. A clear verdict could be drawn from the results, providing an evaluation of the Intel HLS Compiler and paving way for further studies on it. These future studies could involve more complicated components and more extensive trials of using the different pragmas and attributes of the Intel HLS Compiler. Another interesting approach would be to implement one or more components of a larger system with the HLS Compiler and see how much time could be saved by using it when it is feasible.

REFERENCES

1. Nane, Razvan, Sima, Vlad-Mihai, Pilato, Christian, Choi, Jongsok, Fort, Blair, Canis, Andrew, Chen, Yu Ting, Hsiao, Hsuan, Brown, Stephen, Ferrandi, Fabrizio, Anderson, Jason & Bertels, Koen 2015. A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (10). Accessed 22.7.2021. https://ieeexplore.ieee.org/document/7368920. Requires a license.
2. Intel. Intel High Level Synthesis Compiler. Accessed 3.8.2021. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.
3. Xilinx 2021. Vivado Design Suite User Guide. High-Level Synthesis. Accessed 3.8.2021. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.
4. Xilinx. Vitis Unified Software Development Platform 2021.1 Documentation. Accessed 3.8.2021. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/introductionvitishls.html.
5. National Instruments 2020. FPGA Fundamentals. Accessed 22.7.2021. https://www.ni.com/fi-fi/innovations/white-papers/08/fpga-fundamentals.html#section-1251506615.
6. Xilinx 2021. Opening a World of Possibilities: Vitis HLS Front-end is Now Open Source for All on GitHub. Accessed 22.7.2021. https://forums.xilinx.com/t5/AI-and-Machine-Learning-Blog/Opening-a-World-of-Possibilities-Vitis-HLS-Front-end-is-Now-Open/ba-p/1211207.
7. Etteplan. Accessed 22.7.2021. https://www.etteplan.com/
8. Etteplan. FPGA competence group meeting 31.5.2021.
9. Maxfield, Clive 2011. FPGAs: Instant Access. Oxford, UK: Newnes.
10. Maxim integrated 2008. What is a Transmission Gate (Analog Switch)? Accessed 26.7.2021. https://www.maximintegrated.com/en/design/technical-documents/tutorials/4/4243.html.
11. Altera corporation 2020. Cyclone V Device Handbook Volume 1: Device Interfaces and Integration. Accessed 26.7.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v2.pdf.
12. Intel. Adaptive Logic Module (ALM) Definition. Accessed 26.7.2021. https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_alm.htm.
13. FPGAKey. Slice. Accessed 26.7.2021. https://www.fpgakey.com/wiki/details/52.
14. National Instruments 2020. Slices on an FPGA Chip. Accessed 26.7.2021. https://www.ni.com/fi-fi/support/documentation/supplemental/18/slices-on-an-fpga-chip.html.

15. Altera Corporation 2005. Stratix II vs. Virtex-4 Density Comparison. Accessed 26.7.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wpstxiixlnx.pdf.
16. Intel 2017. DSP block definition. Accessed 27.7.2021. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_dsp_block.htm.
17. Xilinx 2020. UltraScale Architecture. DSP Slice. Accessed 27.7.2021. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.
18. Holdsworth, Brian & Woods, Clive 2002. Digital Logic Design. Fourth edition. Oxford, UK: Newnes.
19. Navabi, Zainalabedin & Kaeli, David R. 2009. Computer Science and Engineering. Oxford, UK: EOLSS Publishers/UNESCO.
20. Mehler, Ronald W. 2014. Digital Integrated Circuit Design Using Verilog and SystemVerilog. Oxford, UK: Newnes.
21. Smith, Gina 2010. FPGAs 101. Everything you need to know to get started. Oxford, UK: Newnes.
22. Intel. About the Netlist Viewers. Accessed 3.8.2021. https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/verify/rtl/rtl_view.htm.
23. Intel. About the Chip Planner. Accessed 3.8.2021. https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/optimize/ace/acv_view_acv_overview.htm.
24. Joe Sawicki 2020. Redefining Electronic Design Automation, an IC Perspective. Accessed 3.8.2021. https://blogs.sw.siemens.com/news/redefining-electronic-design-automation/.
25. Intel. ModelSim - Intel FPGA Edition Software. Accessed 3.8.2021. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.
26. Siemens 2019. HDL Simulation. ModelSim. Accessed 3.8.2021. https://static.sw.cdn.siemens.com/siemens-disw-assets/public/6gMVEbq0wlPDGdqA1yvNuE/en-US/Siemens%20SW%20HDL%20Simulation%20ModelSim%20FS%2078330%20C2.pdf .
27. Intel. ModelSim – Intel FPGA Edition Software. Accessed 4.9.2021. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.
28. Intel 2018. Intel Quartus Prime Standard Edition: User Guide. Timing Analyzer. Accessed 4.8.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qps-timing-analyzer.pdf.
29. Intel 2019. Timing Constraints. Accessed 4.8.2021. https://community.intel.com/t5/FPGA-Wiki/Timing-Constraints/ta-p/735562.
30. Intel. Specifying Timing Constraints and Exceptions (TimeQuest Timing Analyzer). Accessed 4.8.2021.

https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/analyze/sta/sta_pro_constraints.htm.

31. Intel 2021. Intel High-Level Synthesis Compiler Pro Edition Reference Manual. Accessed 10.6.2021. https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html.
32. Intel. Simulation with the NativeLink Feature in Quartus II Software. Accessed 4.8.2021. https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/simulation/modelsim/exm-ncsim-native-link.html.
33. Intel 2021. Intel High-Level Synthesis Compiler Pro Edition: User Guide. Accessed 4.8.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf.
34. Intel 2017. Introduction to High-Level Synthesis (Part 1 of 7). Accessed 4.8.2021. https://www.youtube.com/watch?v=nYbw9k7KNJ4.
35. Intel 2020. Timing Analyzer: Introduction to Timing Analysis. Accessed 16.8.2021. https://www.youtube.com/watch?v=HMAqjjCuDEI.
36. Nykamp, Duane. Multiplying matrices and vectors. Accessed 24.8.2021. https://mathinsight.org/matrix_vector_multiplication.
37. Weisstein, Eric W. Matrix Multiplication. Accessed 24.8.2021. https://mathworld.wolfram.com/MatrixMultiplication.html.
38. Intel 2020. Introduction to Intel FPGA IP Cores. Accessed 27.8.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_intro_to_megafunctions.pdf.
39. Octave. Arithmetic Operators. Accessed 26.8.2021. https://octave.org/doc/v4.0.3/Arithmetic-Ops.html.
40. Intel 2021. Avalon Interface Specifications. Accessed 26.8.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.
41. Intel 2021. Intel High Level Synthesis Compiler Pro Edition: Best Practices Guide. Accessed 26.8.2021. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls-best-practices.pdf.
42. Intel Community Forums user and Intel Employee whitepau 2020. Intel Community Forum message. Accessed 30.8.2021. https://community.intel.com/t5/Intel-High-Level-Design/About-10bitsPerSymbol-of-ihc-stream-out/m-p/692172#M625.

APPENDICES

Appendix 1. Verilog code of the stream_into_ram module.

Appendix 2. Verilog code of the matrix_multiply module.

Appendix 3. Verilog code of the multiply_acc module.

Appendix 4. Verilog code of the RTL test bench module.

Appendix 5. C++ code of the HLS implementation and test bench.

```
// Module to stream incoming data into block RAM

module stream_into_ram(
    clk,
    RST,
    CE,
    CE_out,
    busy,
    stall,
    data_a,
    data_b,
    ram_a_addr,
    ram_a_data,
    ram_a_write,
    ram_b_addr,
    ram_b_data,
    ram_b_write);

    parameter RAM_ADDRESS_WIDTH = 12;
    parameter DATAWIDTH = 16;
    parameter FIRSTMATRIXROWS = 32; // Row number of first matrix, column
 number of second matrix
    parameter FIRSTMATRIXCOLS = 64; // Colum number of first matrix, row
of second matrix
    parameter COMP_MULT = 1; // Affects how many elements from each matri
x are read from RAM

    input wire RST, CE, clk, stall;
    input wire [DATAWIDTH*COMP_MULT-1:0] data_a;
    input wire [DATAWIDTH*COMP_MULT-1:0] data_b;
    output reg CE_out;
    output wire ram_a_write, ram_b_write;
    output wire [RAM_ADDRESS_WIDTH-1:0] ram_a_addr;
    output reg [DATAWIDTH*COMP_MULT-1:0] ram_a_data;
    output wire [RAM_ADDRESS_WIDTH-1:0] ram_b_addr;
    output reg [DATAWIDTH*COMP_MULT-1:0] ram_b_data;

    output reg busy;

    reg [RAM_ADDRESS_WIDTH-1:0] ram_addr_reg;

    reg CE_in_reg;

    assign ram_a_write = CE_in_reg & !busy;
    assign ram_b_write = CE_in_reg & !busy;
    assign ram_a_addr = ram_addr_reg;
    assign ram_b_addr = ram_addr_reg;
```

```
    always @(posedge clk)
    begin
        if(!RST)
        begin
            ram_addr_reg <= 0;
            ram_a_data <= 0;
            ram_b_data <= 0;
            CE_out <= 0;
            CE_in_reg <= 0;
            busy <= 0;
        end
        else
        begin
            CE_in_reg <= CE;
            ram_a_data <= data_a;
            ram_b_data <= data_b;

            CE_out <= CE_in_reg;

            if(CE_in_reg & !busy)
            begin
                ram_addr_reg <= ram_addr_reg + 1;
            end

            // If we reach the end of the defined matrix size
            if(ram_addr_reg == FIRSTMATRIXROWS*FIRSTMATRIXCOLS/COMP_MULT-
1)
            begin
                ram_addr_reg <= 0;
                busy <= 1; // Raise busy out to stall upstream component
from writing
            end

            // If busy_reg is high, lower it only when stall is asserted
            if(busy)
            begin
                if(!stall)
                begin
                    busy <= 0;
                end
            end

        end
    end

endmodule
```

```
// Matrix multiplier
 module matrix_multiply(
    RST,
    CE,
    busy,
    clk,
    CE_out,
    inputram_a_addr,
    inputram_a_read,
    inputram_a_q,
    inputram_b_addr,
    inputram_b_read,
    inputram_b_q,
    outputram_addr,
    outputram_data,
    outputram_write);

    parameter DATAWIDTH = 16;
    parameter FIRSTMATRIXROWS = 32; // Row number of first matrix, column
 number of second matrix
    parameter FIRSTMATRIXCOLS = 64; // Colum number of first matrix, row
number of second matrix
    parameter COMP_MULT = 1; // Factor to multiply inferred components (m
ultipliers, adders etc..) by. Increases resource usage but multiply opera
tion will be done in less clock cycles
    parameter INPUT_RAM_ADDRESS_WIDTH = 11;
    parameter OUTPUT_RAM_ADDRESS_WIDTH = 10;
    localparam MACC_INPUT_WIDTH = (DATAWIDTH < 20) ? 20 : DATAWIDTH;
    localparam MACC_OUTPUT_WIDTH = DATAWIDTH * 2 + $clog2(FIRSTMATRIXCOLS
 / COMP_MULT); // Multiply-adders output width
    localparam INPUT_RAM_DATAWIDTH = DATAWIDTH * COMP_MULT;
    localparam OUTPUT_RAM_DATAWIDTH = DATAWIDTH * 2 + $clog2(FIRSTMATRIXC
OLS);
    localparam PRIMARY_COUNTER_WIDTH = $clog2(FIRSTMATRIXCOLS / COMP_MULT
);
    localparam STATE_WIDTH = 2;

    localparam [STATE_WIDTH - 1:0] // States
    WAIT_FOR_CE = 0,
    COMPUTE = 1,
    INTERRUPTED = 2;

    input wire RST, CE, clk;
    output wire CE_out, busy;

//  reg CE_in_reg;
    reg CE_out_reg;
    reg busy_reg;
```

```
    // Registers for tracking internal state
    reg [STATE_WIDTH - 1:0] state_current;
    reg [PRIMARY_COUNTER_WIDTH:0] count_primary;
    reg output_pending;
    reg stalling;
//  reg start;

    // Input RAM wires
    output wire [INPUT_RAM_ADDRESS_WIDTH - 1:0] inputram_a_addr;
    input wire [INPUT_RAM_DATAWIDTH - 1:0] inputram_a_q;
    output wire inputram_a_read;
    output wire [INPUT_RAM_ADDRESS_WIDTH - 1:0] inputram_b_addr;
    input wire [INPUT_RAM_DATAWIDTH - 1:0] inputram_b_q;
    output wire inputram_b_read;

    // Output RAM wires
    output wire [OUTPUT_RAM_ADDRESS_WIDTH - 1:0] outputram_addr;
    output wire [OUTPUT_RAM_DATAWIDTH - 1:0] outputram_data;
    output wire outputram_write;


    // Output RAM control registers
    reg [OUTPUT_RAM_ADDRESS_WIDTH - 1:0] outputram_addr_reg;
    reg [OUTPUT_RAM_DATAWIDTH - 1:0] outputram_data_reg;
    reg outputram_write_reg;

    // Input RAM control registers
    reg [INPUT_RAM_ADDRESS_WIDTH - 1:0] inputram_a_addr_reg;
    reg [INPUT_RAM_ADDRESS_WIDTH - 1:0] inputram_b_addr_reg;
    reg inputram_read_reg;

    // Multiply-accumulator registers
    reg [INPUT_RAM_DATAWIDTH - 1:0] in_a_reg ;
    reg [INPUT_RAM_DATAWIDTH - 1:0] in_b_reg ;

    reg signed [OUTPUT_RAM_DATAWIDTH - 1:0] final_output_reg;

    wire [MACC_INPUT_WIDTH - 1:0] in_a_wires [0:COMP_MULT-1];
    wire [MACC_INPUT_WIDTH - 1:0] in_b_wires [0:COMP_MULT-1];
    wire signed [MACC_OUTPUT_WIDTH - 1:0] out_c_wires [0:COMP_MULT - 1];
    reg signed [MACC_OUTPUT_WIDTH - 1:0] out_c_regs [0:COMP_MULT - 1];

    reg macc_CE_reg;
    reg [4:0] macc_sload_reg;

    // Counter a and b end wires rise when their respective input RAM add
resses reach their end values
    wire counter_a_end;
    wire counter_b_end;
```

```
    wire counter_primary_end; // Rises every FIRSTMATRIXCOLS / COMP_MULT
cycles
    wire can_compute;

    // Connect RAM control wires to their registers
    assign inputram_a_read = inputram_read_reg;
    assign inputram_a_addr = inputram_a_addr_reg + count_primary;
    assign inputram_b_read = inputram_read_reg;
    assign inputram_b_addr = inputram_b_addr_reg;

    assign outputram_data = outputram_data_reg;
    assign outputram_write = outputram_write_reg;
    assign outputram_addr = outputram_addr_reg;

    assign counter_a_end = (inputram_a_addr == (FIRSTMATRIXROWS * FIRSTMA
TRIXCOLS) / COMP_MULT - 1);
    assign counter_b_end = (inputram_b_addr_reg == (FIRSTMATRIXROWS * FIR
STMATRIXCOLS) / COMP_MULT - 1);
    assign counter_primary_end = (count_primary == FIRSTMATRIXCOLS / COMP
_MULT - 1);


    // can_compute indicates if we have data to run the component forward
.
    // It will go LOW if CE goes LOW before all data is fed into cache ra
m,
    //    triggering the INTERRUPT state.
    assign can_compute =  CE  | (! CE  & stalling);

    assign CE_out = CE_out_reg;
    assign busy = busy_reg;


    // Final adder input registers
    integer j;
    always @(posedge clk)
    begin
        for(j = 0; j < COMP_MULT; j = j + 1)
        begin
            // Not wiring reset to these improves fMAX 10-
15 MHz in some cases, although out_c_regs are inferred in ALM's
//          if(!RST)
//          begin
//              out_c_regs[j] <= 0;
//          end else
//          begin
                out_c_regs[j] <= out_c_wires[j];
//          end
        end
```

```
    end

    // Combinational adder for outputs of multiplier-accumulators
    // Calculates the final addition for storing in output RAM
    // Note: When COMP_MULT > 2, this will infer combinational chains of
adders, which limits fMAX.
    always @(*)
    begin
            final_output_reg = 0;
            for(j = 0; j < COMP_MULT; j = j + 1)
            begin
                final_output_reg = final_output_reg + out_c_regs[j];
            end
    end


    genvar n;
    genvar k;
    generate
        for(n = 0; n < COMP_MULT; n = n + 1)
        begin: MACWIREGEN

            // Assign wires for MACs by separating input elements from ca
che RAM output registers:
            assign in_a_wires[n][DATAWIDTH - 1:0] = in_a_reg[DATAWIDTH *
COMP_MULT - DATAWIDTH * n - 1:DATAWIDTH * COMP_MULT - DATAWIDTH * n - DAT
AWIDTH];
            assign in_b_wires[n][DATAWIDTH - 1:0] = in_b_reg[DATAWIDTH *
COMP_MULT - DATAWIDTH * n - 1:DATAWIDTH * COMP_MULT - DATAWIDTH * n - DAT
AWIDTH];

            // Sign-extension for MACs if datawidth is less than 20 bits:
            for(k = 0; k < MACC_INPUT_WIDTH - DATAWIDTH; k = k + 1)
            begin: SIGNEXTGEN
                assign in_a_wires[n][DATAWIDTH + k:DATAWIDTH + k] = in_a_
wires[n][DATAWIDTH - 1:DATAWIDTH - 1];
                assign in_b_wires[n][DATAWIDTH + k:DATAWIDTH + k] = in_b_
wires[n][DATAWIDTH - 1:DATAWIDTH - 1];
            end
        end
    endgenerate

    // Generate multiplier-accumulators
    genvar i;
    generate
        for(i = 0; i < COMP_MULT; i = i + 1)
        begin: MACCGEN
            multiply_acc macc(
                .clk(clk),
```

```
                .CE(macc_CE_reg),
                .sload(macc_sload_reg[2]),
                .a(in_a_wires[i]),
                .b(in_b_wires[i]),
                .c(out_c_wires[i])
                );
                defparam
                macc.DATAWIDTH = DATAWIDTH,
                macc.ADDITIONS = FIRSTMATRIXCOLS / COMP_MULT;


        end
    endgenerate

    // Update multiplier-accumulator input and output registers
    always @(posedge clk)
    begin
        if(!RST)
        begin
            // Wiring reset to these registers hinders fMAX, when COMP_MU
LT = 2. Because there is no sclr in DSP blocks, and these registers are i
nferred within DSP's
//          in_a_reg <= 0;
//          in_b_reg <= 0;
            outputram_data_reg <= 0;
        end else
        begin
            in_a_reg <= inputram_a_q;
            in_b_reg <= inputram_b_q;
            outputram_data_reg <= final_output_reg;
        end
    end


    // State machine to keep track of current state
    //
    // WAIT_FOR_CE: No data is being processed, ready to start new matrix
 multiplication
    //
    // COMPUTE: Currently multiplying a matrix and new data is being fed
to input RAM (or all data has been stored in the
    //          input RAM and the upstream component is being stalled unt
il the end of the multiply operation).
    //
    // INTERRUPTED: Matrix multiplying has been interrupted because CE in
put was asserted before all elements
    //               of input matrices could be stored in RAM. Multiplyin
g will continue once CE is raised.
    //
```

```
    always @(posedge clk)
    begin
        if(!RST)
        begin
            state_current <= WAIT_FOR_CE;
        end else
        begin
            state_current <= state_current;
            case (state_current)

                WAIT_FOR_CE : begin
                    if( CE )
                        state_current <= COMPUTE;
                end

                COMPUTE : begin
                    if(counter_a_end & counter_b_end)
                    begin
                        state_current <= WAIT_FOR_CE;
                    end else
                    begin
                        if(!can_compute & !counter_b_end)
                            state_current <= INTERRUPTED;
                    end
                end

                INTERRUPTED : begin
                    if( CE )
                        state_current <= COMPUTE;
                end

            endcase
        end

    end

    // Addressing, multiplier-
accumulator control and input/output control
    always @(posedge clk)
    begin
        if(!RST) // RESET condition
        begin
            count_primary <= 0;
            inputram_a_addr_reg <= 0;
            inputram_b_addr_reg <= 0;
            busy_reg <= 0;
            macc_CE_reg <= 0;
            macc_sload_reg <= 5'b00111;
        end
```

```
        else
        begin

            // Registers to delay sload signal for multiply-accumulators
            // Also used for triggering output RAM writes after
            // macc result is valid:
            macc_sload_reg[4] <= macc_sload_reg[3];
            macc_sload_reg[3] <= macc_sload_reg[2];
            macc_sload_reg[2] <= macc_sload_reg[1];
            macc_sload_reg[1] <= macc_sload_reg[0];

            // State-dependent operations
            case (state_current)
                WAIT_FOR_CE : begin

                end

                COMPUTE : begin
                    busy_reg <= 1'b1;
                    macc_CE_reg <= busy_reg;
                    macc_sload_reg[0] <= counter_primary_end;

                    count_primary <= count_primary + 1;
                    inputram_b_addr_reg <= inputram_b_addr_reg + 1;

                    if(counter_primary_end)
                    begin
                        count_primary <= 0;
                    end


                    if(counter_b_end)
                    begin
                        inputram_b_addr_reg <= 0;
                        inputram_a_addr_reg <= inputram_a_addr_reg + (FIR
STMATRIXCOLS / COMP_MULT);
                        if(counter_a_end)
                        begin
                            // RESET
                            count_primary <= 0;
                            busy_reg <= 0;
                            inputram_a_addr_reg <= 0;
                        end
                    end

                end

                INTERRUPTED : begin
                    // Stop multiply-accumulators
```

```
                    macc_CE_reg <= macc_sload_reg[0] | macc_sload_reg[1];

                    // If we have processed all data for the next output,
 reset multiply-accumulators:
                    if(counter_primary_end)
                    begin
                        macc_sload_reg[0] <= 1;
                    end
                end
            endcase

        end
    end

    // State-independent operations
    always @(posedge clk)
    begin
        if(!RST)
        begin
            output_pending <= 0;
            stalling <= 0;
            outputram_write_reg <= 0;
            outputram_addr_reg <= 0;
            CE_out_reg <= 0;
            inputram_read_reg <= 0;
        end else
        begin

            // Read from input RAM only if the data there is valid
            inputram_read_reg <= can_compute | counter_b_end;

            // Set output pending LOW after data has been written.
            if(macc_sload_reg[4])
            begin
                output_pending <= 0;
            end

            // If counter b has reached its end at least once, it means a
ll data should now be in input RAM.
            // Stalling register is used in internal state tracking.
            if(counter_b_end)
            begin
                stalling <= 1;
            end

            // Raise output pending when primary counter reaches end
            if(counter_primary_end)
            begin
                output_pending <= 1;
```

```
            end

            // Raise output write signal if output is pending and multipl
y-accumulators are done
            outputram_write_reg <= output_pending & macc_sload_reg[4];

            // Raise valid signal 1 cycle after writing to output RAM
            CE_out_reg <= outputram_write_reg;

            // If not busy and data has been writted to output RAM, it me
ans all data should have been processed.
            // Thus we can reset the appropriate registers.
            if(outputram_write_reg & !busy_reg)
            begin
                outputram_addr_reg <= 0;
                outputram_write_reg <= 0;
                output_pending <= 0;
                stalling <= 0;
            end

            // Increment output RAM address after a successful write
            if(outputram_write_reg & busy_reg)
            begin
                outputram_addr_reg <= outputram_addr_reg + 1;
            end
        end
    end
endmodule
```

```
// Multiply-accumulator module
//
//
// NOTE: DSP Blocks of Cyclone V devices are placed in their
//  MAC configuration only if data width is over 19 bits.
//    Therefore the input width is limited to minimum of 20 bits.
//
//    If the input width is less than 20 bits, the added bits should be p
added
//    with the MSB of the original vector before feeding them to this mod
ule.
 module multiply_acc(
    clk,
    CE,
    sload,
    a,
    b,
    c);
    parameter DATAWIDTH = 16;
    parameter ADDITIONS = 64;
    localparam DATAWIDTH_LIMITED = (DATAWIDTH < 20) ? 20 : DATAWIDTH;
    localparam OUTPUT_WIDTH = DATAWIDTH_LIMITED * 2 + $clog2(ADDITIONS);
    localparam COUNTER_WIDTH = $clog2(ADDITIONS);

    input wire clk, CE;
    input wire signed [DATAWIDTH_LIMITED - 1:0] a;
    input wire signed [DATAWIDTH_LIMITED - 1:0] b;
    output reg signed [OUTPUT_WIDTH - 1:0] c;
    input wire sload;

    wire signed [DATAWIDTH_LIMITED * 2 - 1:0] mult;

    reg signed [OUTPUT_WIDTH - 1:0] c_reg;
    reg CE_in_reg;
    reg sload_reg;

    assign mult = a * b;

    always @(sload_reg, c)
    begin

        if(sload_reg)
        begin
            c_reg <= 0;
        end else
        begin
            c_reg <= c;
        end
```

```
    end

    always @(posedge clk)
    begin
        CE_in_reg <= CE;
        sload_reg <= sload;

        if(CE_in_reg)
        begin
            c <= c_reg + mult;
        end
    end

endmodule
```

```
// Test bench for dsp_rtl
`timescale 1ps / 1ps

module dsp_rtl_tb();

    `define NULL 0

    localparam DATAWIDTH = 20;
    localparam FIRSTMATRIXROWS = 32; // Row number of first matrix, colum
n number of second matrix
    localparam FIRSTMATRIXCOLS = 64; // Colum number of first matrix, row
 number of second matrix
    localparam COMP_MULT = 1; // Factor to multiply number of components
(multipliers, adders etc..) by. Increases resource usage but multiply ope
ration will be done in less clock cycles
    localparam OUTPUT_RAM_WIDTH = DATAWIDTH*2 + $clog2(FIRSTMATRIXCOLS);
    localparam INPUT_RAM_ADDRESS_WIDTH = 11;
    localparam OUTPUT_RAM_ADDRESS_WIDTH = 10;
    localparam RUN_TWICE = 1; // 0 for not, 1 for run twice

    reg [DATAWIDTH-1:0] input_data_a [0:FIRSTMATRIXROWS*FIRSTMATRIXCOLS-
1];
    reg [DATAWIDTH-1:0] input_data_b [0:FIRSTMATRIXROWS*FIRSTMATRIXCOLS-
1];
    reg [OUTPUT_RAM_WIDTH-
1:0] res_expected [0:FIRSTMATRIXROWS*FIRSTMATRIXCOLS-
1]; // Expected results
    reg [OUTPUT_RAM_WIDTH-
1:0] res_actual [0:FIRSTMATRIXROWS*FIRSTMATRIXCOLS-1]; // Gotten results

    reg clk;
    reg start;
    reg start_internal, interrupted;
    wire busy_w;
    reg done;
    wire valid_w;
    reg reset;
    reg [DATAWIDTH*COMP_MULT-1:0] data_a;
    reg [DATAWIDTH*COMP_MULT-1:0] data_b;

    wire [OUTPUT_RAM_ADDRESS_WIDTH-1:0] outputram_addr;
    wire [OUTPUT_RAM_WIDTH-1:0] outputram_data;
    wire outputram_write;

    integer count;
    integer output_count;
    integer output_file;
    integer run_count;
```

```
    dsp_rtl dsp_block(
        .data_a(data_a),
        .data_b(data_b),
        .clk(clk),
        .start(start),
        .busy(busy_w),
        .valid(valid_w),
        .RST(reset),
        .outputram_addr(outputram_addr),
        .outputram_data(outputram_data),
        .outputram_write(outputram_write));
        defparam
        dsp_block.INPUT_RAM_ADDRESS_WIDTH = INPUT_RAM_ADDRESS_WIDTH,
        dsp_block.DATAWIDTH = DATAWIDTH,
        dsp_block.FIRSTMATRIXROWS = FIRSTMATRIXROWS,
        dsp_block.FIRSTMATRIXCOLS = FIRSTMATRIXCOLS,
        dsp_block.OUTPUT_RAM_ADDRESS_WIDTH = OUTPUT_RAM_ADDRESS_WIDTH,
        dsp_block.COMP_MULT = COMP_MULT;

    initial
    begin
        clk = 0;
        start = 0;
        reset = 0;
        count = 0;
        output_count = 0;
        interrupted = 0;
        run_count = 0;

        output_file = $fopen("io_files/counter_output_run_1.csv", "w"); /
/ open file
        $fdisplay(output_file, "count, res_expected, res_actual");

        $readmemh("io_files/input_data_a_run_1.csv", input_data_a);
        $readmemh("io_files/input_data_b_run_1.csv", input_data_b);
        $readmemh("io_files/expected_results_run_1.csv", res_expected);

        #120;
        reset = 1;
        #120;
        start_internal = 1;
    end

    always
    begin
        clk = !clk;
        #50;
    end
```

```
    integer i;

    // Test for interrupt mid-calculations
    always @(output_count)
    begin
        if(output_count == 4)
        begin
            start_internal = 0;
            interrupted = 1;
            #1000;
            start_internal = 1;
            interrupted = 0;
        end

    end

    // Test for interrupt at the end of a multiply operation
    always @(count)
    begin
        if(count == 63 + COMP_MULT)
        begin
            start_internal = 0;
            interrupted = 1;
            #1000;
            start_internal = 1;
            interrupted = 0;
        end

    end

    always @(posedge clk)
    begin
        if(reset)
        begin
            if(start_internal)
            begin
                data_a = 0;
                data_b = 0;

                for(i = 0; i < COMP_MULT; i = i + 1)
                begin
                    if(i > 0)
                    begin
                        data_a = data_a << DATAWIDTH; // Shift left by DA
TAWIDTH*2
                        data_b = data_b << DATAWIDTH; // Shift left by DA
TAWIDTH*2
                    end
```

```
                    data_a = data_a | input_data_a[count + i]; // Insert
new data
                    data_b = data_b | input_data_b[count + i]; // Insert
new data
                end

                if(count == FIRSTMATRIXROWS * FIRSTMATRIXCOLS + COMP_MULT
)
                begin
                    start = 0;
                end else
                begin
                    if(busy_w == 0)
                    begin
                        start = 1;
                        count = count + COMP_MULT;
                    end
                end

                // If data is valid @ output RAM
                if(outputram_write)
                begin
                    res_actual[output_count] = outputram_data;
                    case(res_actual[output_count] == res_expected[output_
count])
                        1'b1 : begin

                        end
                        default : begin
                            $display("Unexpected multiply result at index
 %d, run %d: Expected: %d Got: %d", output_count, run_count, res_expected
[output_count], res_actual[output_count]);
                        end
                    endcase

                    $fdisplay(output_file, "%d,%d,%d", output_count, res_
expected[output_count], res_actual[output_count]);

                    output_count = output_count + 1;
                end

                if(output_count >= FIRSTMATRIXROWS * FIRSTMATRIXROWS)
                begin
                    start_internal = 0;
                    count = 0;
                end
            end else
            begin
                start = 0;
```

```
                if(output_count >= FIRSTMATRIXROWS * FIRSTMATRIXROWS)
                begin
                    count = 0;
                    output_count = 0;
                    if(!interrupted)
                    begin
                        #1000;
                        $fclose(output_file);
                         // Start run 2 to test if components reset corre
ctly:
                        if(run_count < RUN_TWICE)
                        begin
                            run_count = run_count + 1;
                            $readmemh("io_files/input_data_a_run_2.csv",
input_data_a);
                            $readmemh("io_files/input_data_b_run_2.csv",
input_data_b);
                            $readmemh("io_files/expected_results_run_2.cs
v", res_expected);
                            output_file = $fopen("io_files/counter_output
_run_2.csv", "w"); // open file
                            $fdisplay(output_file, "count, res_expected,
res_actual");
//                            reset = 0;
//                            #120;
//                            reset = 1;
                            start_internal = 1;
                        end else
                        begin
                            $stop;
                        end
                    end
                end
            end
        end
    end
endmodule
```

```
#include "HLS/stdio.h"
#include "HLS/math.h"
#include "HLS/hls.h"
#include "HLS/ac_fixed.h"
#include "HLS/ac_fixed_math.h"
#include "HLS/math_dsp_control.h"
#include "HLS/matrix_mult.h"

#define first_matrix_rows 32
#define first_matrix_cols 64
#define data_width 20
#define comp_mult 1
#define output_width 46

typedef ac_int<data_width, true> fixed_input_t;
typedef ac_int<output_width, true> fixed_output_t;

struct fixed_input_arr
{
    fixed_input_t data[comp_mult];
};

typedef ihc::stream_in<
    ac_int<32, true>,
    ihc::bitsPerSymbol<32>, // Has to be a power of two value
    ihc::buffer<0> > in_stream_interface_t;

typedef ihc::stream_out<
    ac_int<64, true>,
    ihc::bitsPerSymbol<64>,
    ihc::buffer<0>,
    ihc::usesReady<false> > out_stream_interface_t;


// Avalon ST interfaces
component hls_max_concurrency(1) hls_use_stall_enable_clusters hls_stall_
free_return hls_disable_component_pipelining
    void  dsp_hls(in_stream_interface_t &a, in_stream_interface_t &b, out
_stream_interface_t &out_c)
{

    // Declare memories
    fixed_input_t a_mem[first_matrix_rows][first_matrix_cols] hls_memory_
impl("BLOCK_RAM");
    fixed_input_t b_mem[first_matrix_rows][first_matrix_cols] hls_memory_
impl("BLOCK_RAM");

    #pragma loop_coalesce
```

```
    for(uint6 input_row_counter = 0; input_row_counter < (uint6) first_ma
trix_rows; input_row_counter++){
        for(uint7 input_col_counter = 0; input_col_counter < (uint7) firs
t_matrix_cols; input_col_counter++){
            a_mem[input_row_counter][input_col_counter] = a.read();
            b_mem[input_row_counter][input_col_counter] = b.read();
        }
    }

    #pragma loop_coalesce 2
    for(uint6 a_row = 0; a_row < (uint6) first_matrix_rows; a_row++){
        for(uint6 b_col = 0; b_col < (uint6) first_matrix_rows; b_col++){

            fixed_output_t c = (fixed_output_t) 0;

            #pragma ii 1
            #pragma unroll comp_mult
            for(uint7 count_primary = 0; count_primary < (uint7) first_ma
trix_cols; count_primary++){
                c += a_mem[a_row][count_primary] * b_mem[b_col][count_pri
mary];
            }

            out_c.write(c);
        }
    }
}




// Test bench
int main()
{

    fixed_input_t in_a[first_matrix_cols * first_matrix_rows];
    fixed_input_t in_b[first_matrix_cols * first_matrix_rows];
    fixed_output_t out_c[first_matrix_rows * first_matrix_rows];
    fixed_output_t expected_c[first_matrix_rows * first_matrix_rows];

    in_stream_interface_t in_streams[2];
    out_stream_interface_t out_c_stream;

    for(int i = 0; i < first_matrix_cols * first_matrix_rows; i++){
        in_a[i] = (fixed_input_t) i + 1;
        in_b[i] = (fixed_input_t) i + 1;
        in_streams[0].write((fixed_input_t) i + 1);
        in_streams[1].write((fixed_input_t) i + 1);
    }
```

```
    int count = 0;
    int out_count = 0;
    fixed_output_t c = 0;

    // Expected results:
    for(int a_count = 0; a_count < first_matrix_cols * first_matrix_rows;
 a_count = a_count + first_matrix_cols){

        for(int b_count = 0; b_count < first_matrix_cols * first_matrix_r
ows; b_count++){
            c = c + in_a[a_count + count] * in_b[b_count];
            if(count == first_matrix_cols - 1){
                expected_c[out_count] = c;
                c = 0;
                count = 0;
                out_count++;
            }
            else{
                count++;
            }
        }
    }

    dsp_hls(in_streams[0], in_streams[1], out_c_stream);

    printf("Component calls done\n");

    for(int i = 0; i < first_matrix_rows * first_matrix_rows; i++){
        out_c[i] = out_c_stream.read();
        if(out_c[i] != expected_c[i]){
            printf("Unexpected result at %d: %X, expected %X\n", i, out_c
[i].to_uint(), expected_c[i].to_uint());
        }
    }

    return 0;

}
```