Evaluation of High Level Synthesis Frameworks: Analysis Across Three Programming Paradigms

By

Farhad Andalibi

A thesis submitted to the Faculty of Graduate and Post Doctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

©2021
Farhad Andalibi
Abstract

High Level Synthesis (HLS) has played an important role in the design of high performance Field Programmable Gate Array (FPGA) based solutions and it is increasingly popular among developers. In this thesis, two HLS tools, Vivado HLS and Clash, supporting three programming language paradigms (imperative, transaction-level modeling, and functional) are investigated.

To assess the characteristics and performance of these HLS tools, we implement two trivial and non-trivial applications, Finite Impulse Response (FIR) filter and EigenValue Decomposition (EVD) with C++, SystemC and Clash.

Pre-synthesis and post-synthesis results are elaborated to ensure each tool generates consistent outputs and simulation results are verified with the same program in MATLAB. It was found that Clash usually has better performance and latency as a result of the intrinsic parallelism feature of functional programming language Haskell. However, it was also found Vivado HLS has better results regarding power consumption and resource utilization, and provides more options for optimization of a design.
Acknowledgments

Throughout the process of writing this master thesis, I’ve got broad support and
guidance.

First and foremost, I would like to express my deepest appreciation to my su-
pervisor, Dr. Paulo Garcia. The completion of my research work would not have
been possible without your assistance and nurturing. Especially during the Covid-
19 circumstances, your availability and flexibility to regulate meetings have been
invaluable.

I’d like to extend my gratitude to engineers who work in the QBayLogic company.
I’ve received a lot of support in debugging and fixing issues related to Clash compiler
from these people. I’d also like to thank other students in our research group. We
had great times during our research meetings.

Finally, I am grateful to my parents who have done a lot for me in different
situations and supported me in this long journey to achieve my goals.
Contents

List of Figures v
List of Tables vi
Acronyms vii

1 Introduction 1
   1.1 Problem statement ................................................. 1
   1.2 Overview of the thesis ............................................ 2

2 Background 3
   2.1 FPGAs .......................................................... 3
   2.2 Clash ........................................................... 4
      2.2.1 Introduction ................................................... 4
      2.2.2 Function definition in Clash ............................... 5
      2.2.3 Simulation in Clash ......................................... 6
   2.3 Vivado HLS ......................................................... 7
      2.3.1 C/C++ synthesis .............................................. 8
      2.3.2 SystemC synthesis ........................................... 9

3 Related work 12
   3.1 Overview of Other General Purpose HLS ......................... 12
   3.2 Evaluation of Different High-Level Synthesis Tools ............. 13
   3.3 DSLs in high level synthesis .................................... 16
      3.3.1 Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines ........................................ 16
      3.3.2 Optimus: Efficient Realization of Streaming Applications on FPGAs ..................................................... 16
      3.3.3 \textit{HIPA}^{cc}: A Domain-Specific Language and Compiler for Image Processing ........................................ 17
      3.3.4 RIPL: A Parallel Image Processing Language for FPGAs ... 17
      3.3.5 High-Level Synthesis from C vs. a DSL-based Approach ... 18
      3.3.6 Conclusion .................................................... 18

4 Implementation 19
   4.1 FIR Filter ......................................................... 19
      4.1.1 Vivado HLS Implementation of FIR Filter .................. 19
      4.1.2 Clash Implementation of FIR Filter .......................... 20
   4.2 Eigenvalue and Eigenvector Decomposition Algorithm ........... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1 Covariance Matrix Implementation</td>
<td>23</td>
</tr>
<tr>
<td>4.2.2 CORDIC Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.2.3 Traditional Jacobi Method</td>
<td>24</td>
</tr>
<tr>
<td>4.2.4 Improved Jacobi Algorithm for EVD</td>
<td>25</td>
</tr>
<tr>
<td>4.2.5 Systolic Array</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Clash Implementation of EVD</td>
<td>28</td>
</tr>
<tr>
<td>4.4 Vivado HLS Implementation of EVD</td>
<td>30</td>
</tr>
<tr>
<td>5 Results</td>
<td>36</td>
</tr>
<tr>
<td>5.1 FIR Filter Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1 Synthesis Results of FIR Filter</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Eigenvalue Decomposition Evaluation</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1 Synthesis Result of EVD</td>
<td>39</td>
</tr>
<tr>
<td>6 Discussion</td>
<td>41</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>43</td>
</tr>
<tr>
<td>7.1 Future Work</td>
<td>44</td>
</tr>
<tr>
<td>Bibliography</td>
<td>45</td>
</tr>
<tr>
<td>A C++ Code for EVD</td>
<td>49</td>
</tr>
<tr>
<td>B SystemC Code for EVD</td>
<td>56</td>
</tr>
<tr>
<td>C Clash Code for EVD</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Figures

2.1 FPGA Structure ................................................................. 4
2.2 zipWith structure from [1] .................................................. 5
2.3 Transfer function of mealy and moore machine ......................... 6
2.4 Simulation of dot product function ....................................... 7
2.5 MAC module ................................................................. 7
2.6 Vivado HLS design flow ..................................................... 8
2.7 Modules A and B are instantiated within Module C. Signal s is used to connect ports of A and B preeti ................................. 10

3.1 ReWire Compilation Process ............................................... 13
3.2 Utilization report from Vivado HLS ..................................... 15
3.3 Utilization report from Vivado HDL .................................... 15
3.4 Logic Utilization report ..................................................... 16
3.5 The stage of the Darkroom compiler ................................... 17
3.6 LALP compilation flow ..................................................... 18

4.1 Calculation of $\lambda_i$ and $\beta_i$ based on Eq. (12) ................... 26
4.2 CORDIC rotation ............................................................ 26
4.3 Proposed Systolic array from [22] ...................................... 27
4.4 Data exchange between each column ................................... 28

5.1 Simulation parameters in Clash ......................................... 39
List of Tables

2.1 Dot product implementation .................................................. 5
2.2 Mealy definition in Clash from Clash.Prelude library ............... 6
2.3 SystemC AND implementation ................................................ 11

3.1 MD5 hashing algorithm .......................................................... 14
3.2 Floating point addition ......................................................... 15
3.3 Tri-Diagonal linear system ..................................................... 16

4.1 FIR filter in C++ ................................................................. 20
4.2 FIR filter in SystemC ............................................................ 21
4.3 FIR filter in Clash ............................................................... 22
4.4 Rotation angles ................................................................. 24
4.5 CMC Module ...................................................................... 31
4.6 CordicA Module ................................................................. 32
4.7 LUT Module ..................................................................... 32
4.8 Resize Module ................................................................. 33
4.9 Update Module ............................................................... 33
4.10 CordicB Module .............................................................. 34
4.11 EVD Testbench Function .................................................... 35

5.1 Related code to test FIR filter in Clash interpreter ................. 37
5.2 Standard Optimization Result (FIR Filter) .............................. 37
5.3 Performance-Optimized Results (FIR Filter) ......................... 38
5.4 Standard Optimization Result (EVD) ..................................... 40
5.5 Performance-optimized Result (EVD) ................................... 40
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
<td>16</td>
</tr>
<tr>
<td>CORDIC</td>
<td>Coordinate Rotation Digital Computer</td>
<td>23</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>3</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
<td>2, 18, 43</td>
</tr>
<tr>
<td>EVD</td>
<td>EigenValue Decomposition</td>
<td>i, 23</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
<td>i</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>i, 1</td>
</tr>
<tr>
<td>FPS</td>
<td>Frame per Second</td>
<td>18</td>
</tr>
<tr>
<td>GF</td>
<td>Galois Field</td>
<td>15</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
<td>14, 17</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
<td>1, 3</td>
</tr>
<tr>
<td>HLS</td>
<td>High Level Synthesis</td>
<td>i, 1, 3, 18, 44</td>
</tr>
<tr>
<td>IOB</td>
<td>Input-Output Block</td>
<td>42</td>
</tr>
<tr>
<td>LALP</td>
<td>Language for Aggressive Loop Pipelining</td>
<td>18</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
<td>27</td>
</tr>
<tr>
<td>PSP</td>
<td>Platform Support Package</td>
<td>13</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
<td>1, 3</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years, the complicated applications in System on Chip (SoC) have made designers more interested in using High Level Synthesis [29]. One of the primary goals of HLS is to reduce the verification and debugging time corresponding to Register Transfer Level (RTL) design which addresses the time-to-market issue.

There has been extensive research work focused on designing SoCs on Field Programmable Gate Array with Hardware Description Language (HDL) such as Verilog and VHDL, as the primary programming languages for FPGAs [10]. FPGAs can decrease the processing time of tasks in a system due to its parallel computing potential, making them the best target for most applications that need high performance. As FPGAs contains programmable logical components, Verilog [12] and VHDL [31] are perfect choices of programming languages, enabling designers to develop their platform for any given application. However, designers have to be experts in the concept of hardware design, which is not a likely scenario for most users, especially with the growing use of FPGAs among software developers.

To implement SoCs on FPGAs with High Level Synthesis, an HLS tool or compiler compatible with the corresponding high level languages (e.g. C/C++, Haskell) [27] is required. There are a wide variety of HLS compilers that support different language paradigms. By using HLS, hardware implementation can be done with high level languages rather than conventional HDLs such as Verilog and VHDL. Therefore, abstraction, which is one the most desirable features of computing, can be satisfied.

Based on an experiment which was shown in [38], the design of 1Mgate requires almost 300K lines of code when the programming language is one of the traditional HDLs. However, this amount of code can be decreased to 40K when transferred to High Level Synthesis which depicts the improvement of productivity.

1.1 Problem statement

A wide range of research work have been conducted to compare different HLS tools and compilers and evaluate their performance against each other with trivial and non-trivial benchmarks. It was proved that different languages give different results for the same algorithm [6], [16], [27], [21]. Most of these studies do not offer enough transparency about utilization, performance, latency and power consumption of generated RTL code after synthesis to make decisions about which tool can perform better for different applications.
In this thesis, we compare two compilers, Clash and Vivado HLS, supporting the use of functional, transaction-level modeling and imperative programming languages, respectively. Clash uses the Haskell programming language and Vivado HLS, a famous tool among FPGA users, supports C/C++ and SystemC as input languages.

There are two applications implemented to make this comparison between mentioned languages and tools. FIR filter which is a trivial algorithm has been written in Clash, C++ and SystemC to provide a general view of the performance. Also, eigenvalue and eigenvector decomposition algorithm that is a non-trivial program for hardware implementation has been investigated.

1.2 Overview of the thesis

The contribution of this dissertation are as follow:

In chapter 2, the background information such as an introduction to Clash and its syntax, Vivado HLS and simulation process in each tool which are related to this study are discussed.

In chapter 3, we summarized relevant papers and studies corresponding to the topic of our research such as other general purpose HLS languages and Domain Specific Language (DSL)s.

Chapter 4 describes the implementation process of FIR filter and EVD algorithm in both tools following with mathematical concepts required for each design.

The standard optimization and performance-optimized result of each experiment is carried out in chapter 5.

In chapter 6, we examine the resulting design flow and performance of each tool and programming language separately in detail.

Finally, the future work and possible improvements related to this study are discussed in chapter 7.
Chapter 2

Background

2.1 FPGAs

Field Programmable Gate Array (FPGA) is a semiconductor device consisting of programmable logic blocks [24]. Figure 2.1 shows the basic structure of an FPGA: logic is surrounded by programmable input/output blocks (labeled I/O in the figure) that connect the chip to the outside world. These blocks can be configured as many times as required to implement the desired hardware. One of the features that distinguishes FPGAs from Central Processing Unit (CPU) is the ability of performing algorithms in parallel. This parallelism significantly improves the efficiency and speed of a task. For instance, in CPU, a mathematical algorithm requiring computing a matrix multiplication requires many clock cycles. However, the same application can be finished in one clock cycle in an FPGA device, thanks to streaming and parallelism.

For many years, FPGA programming was limited to hardware designers and people who had acquired HDL knowledge. The primary programming languages for FPGAs are VHDL, Verilog and SystemVerilog, classified into low level abstraction languages. By using an FPGA vendor toolchain, HDL code can be synthesized to a bitstream (a configuration file for an FPGA that implements a specific circuit). These HDLs do not offer the flexibility of high level languages which make the development process of complex designs time consuming.

In recent years, the advent of HLS has filled the gap between software development and FPGA design. HLS is the translation of software languages to hardware descriptions[23]. Different tools and compilers have been developed which accept high level languages as their input and produce RTL specification as an output compatible with FPGA structure. Both software community and hardware community can take advantage of HLS. Based on a study in [13], developed HLS tools and the automation task of converting high level languages to one of HDLs in these HLS tools enhance FPGA utilization and performance of designs which is the main focus of hardware designers. On the other hand, software designers will now be able to use FPGAs for complex applications which requires high performance such as image processing and program them using software languages.
2.2 Clash

2.2.1 Introduction

Clash is a combination of compiler and language extension which is based upon functional programming language Haskell [8]. So Clash lies in the category of functional hardware description languages that can be used to design combinational and sequential circuits. It compiles this high level of abstraction to low level synthesizable VHDL, Verilog or SystemVerilog. Clash has been developed at the University of Twente.

Since Haskell is a pure functional language which means that it just supports functional syntax and based on mathematical function [3], it would be a great choice for hardware description as each component in hardware can be defined as a set of function and hardware inputs and outputs can be expressed as function arguments in Haskell. Furthermore, the immutable state of Haskell results in parallel programming which can be compared to HDLs characteristic as they are used to declare instructions in parallel on an FPGA and this is the main difference and advantage of FPGAs against processor cores.

Another interesting feature that has made Clash an appropriate language for hardware description is the concept of higher order function exists in Haskell [8],[9]. Higher order functions can be used in function composition because they either use other functions as their input argument or as their output. This prominent susceptibility of functional languages is important in implementing mathematical algorithms on FPGA.

Clash also utilizes a strong type system that inherits from Haskell compiler. Unlike imperative languages that we have to define data types for each variable or function (e.g. Int, Unsigned), in Clash there is no need for data type annotation as the compiler will deduce it automatically due to its intelligent type system which results in faster prototyping of the hardware.

Clash exploits an interactive interpreter which helps you to evaluate your design step by step and without creating a related testbench [2]. Combinational circuits
can be simulated by writing desired function follow with its inputs on the Clash interpreter (Clashi) and synchronous sequential circuits can be simulated by assigning the function to a Clash pre-defined function namely *simulate* which will give you different outputs based on mentioned inputs.

### 2.2.2 Function definition in Clash

As we can implement each hardware component as a set of functions, it is necessary to have a good understanding of function’s concept in Clash. The structure of all functions is shown as below:

\[
\text{function name input1 input2 = output}
\]

There is no limitation in the number of inputs and outputs but each function must have one input and one corresponding output at least.

As an example, the definition of a dot product function of two vectors is shown as follow:

```
dot vec1 vec2 = sum (zipWith (*) vec1 vec2)
```

Table 2.1: Dot product implementation

As it can be seen in table 2.1, the `dot` function has two inputs `vec1` and `vec2` and one output which consists of a predefined function `sum` and a higher order function `zipWith`. Higher order functions are responsible for parallel operations. `zipWith` executes an argument function which is multiplication (*) in this case for all vectors elements. Figure 2.2 illustrates the structure of `zipWith` function.

![zipWith structure from [1]](image)

The figure represents that `zipWith` takes two vectors (i.e. lists in Haskell) and applies function \( f \) to each element of these two lists and produces a vector of the corresponding function. Function \( f \) can be either a complex function that is defined by the user or a simple pre-defined function such as `sum`, `subtract`, `multiplication` and so on which are all stored in *Clash.Prelude* library. Also by comparing dot product declaration in Clash where there is only one line of code with imperative languages such as C/C++, the power of Clash can be easily deduced.
The combinational specification of a circuit has been shown. As mentioned before, Clash supports synchronous sequential circuits as well. These kind of circuits can be defined by using the structure of two finite state machines, namely mealy machine and Moore machine [8]. The output of the mealy machine depends on both the current state and current input while in Moore machine, the output is determined by its current state. Figure 2.3a and 2.3b show the structure of Mealy and Moore machines respectively.

The type architecture of mealy is defined in table 2.2: It shows that mealy function takes one input in the form of a transfer function which by its own included two inputs, state (s) and input (i) and produces an output in the form of a tuple (new state, output), and the second input is the initial state (s) and finally generates an output of a synchronous sequential function with input and output matching mealy machine. Signal data type in Clash is a stream of values and sequential circuits design is based on these signals. The type structure of Moore machine can also be found in [1]. Therefore, mealy and Moore functions are used to create sequential circuits from a combinational function that has similar syntax to Mealy or Moore machine.

2.2.3 Simulation in Clash

As explained before, Clash can generate RTL testbench which can be used in all FPGA vendor tools (e.g. Xilinx, Quartus) for a variety of purposes such as power consumption evaluation. Also, Clash exploits an interactive interpreter where we can simulate our design or functions without creating a predefined testbench to validate the functionality of the design.

For simulation of combinational functions, we just enter the function name on the interpreter follow by its corresponding inputs and the result will pop up instantly. Figure 2.4 shows the simulation result of dot product function displayed in Table 2.1. It can be seen from the type structure that dot function takes two vectors (Vec
n a) as its inputs. The function applied to two vectors with \{1,2,3,4\} and \{2,3,4,5\} elements. The corresponding output is 40.

![Figure 2.4: Simulation of dot product function](image)

Synchronous sequential circuits and functions can be evaluated using `simulate` function which is a predefined function that exists in Clash.Prelude library. Multiply and accumulate (MAC) function which is a good fit for mealy machine structure, can be defined as a sequential circuit. Figure 2.5 displays the declaration and simulation result of MAC module in Clash.

![Figure 2.5: MAC module](image)

As illustrated in the interpreter, `mac` function has an input in the form of Signal \((o,o)\) which means that mac takes a stream of values in a tuple. `simulateN` function has been used to evaluate the first N (N = 5) input samples of mac and the output is a list of 5 computed values.

### 2.3 Vivado HLS

Vivado HLS is the Xilinx high-level synthesis tool that acts as both a compiler and a set of C libraries contain functions and constructs that are optimized for implementation of FPGAs. Vivado HLS supports C/C++ and SystemC as its input languages, however, SystemC has been deprecated in the recently published version of Vivado HLS known as Vitis and SystemC is only available in Vivado HLS 2019.2 and earlier versions. In addition to generating VHDL and Verilog files, it synthesizes C top-level function to IP block that can be integrated to hardware system using Vivado Design Suite tool [28]. Figure 2.6 shows the design flow process in Vivado HLS.
2.3.1 C/C++ synthesis

Basically, writing C code in Vivado HLS for circuit description is exactly the same as writing software in C language. However, some constructs in C are not synthesizable. These limitations are as follow:

- System calls such as `printf()` in C and `cout` in C++.
- Dynamic memory usage such as `malloc()` that consuming memory resources of the operating system.
- Recursive functions.

Same as Clash and other HLS tools, C design in Vivado HLS must have one top-level function that will be synthesized to top level RTL module and other functions should be in the hierarchy of the top function as sub-functions and they synthesize into RTL blocks. Two different tasks should be satisfied in synthesizing a C design:

1. Interface synthesise
2. Algorithm synthesise

Interface synthesis is expressed as the procedure of synthesizing C function arguments into RTL I/O ports. This can be achieved by investigating the top-level function elements and determining their role. For instance, as pointers can be instantiated as read, write, or both, it should figure out what kind of port (input, output, or both) after synthesizing.

Algorithm synthesis determines the behavior of the system over several clock cycles. It is achieved by unrolling the loops and examining the conditional statements of the design. Then in scheduling (finds which operation occurs during each clock
cycle) and binding (finds which hardware resource implements each schedules operation) stages, the optimum implementation is found using the knowledge of clock, latency, and defined constraints.

According to the fact that imperatives languages are not well suited for parallel programming, Vivado HLS provides some compiler directives (pragma) which are used for expressing parallelism and other optimization through the entire design. For example, FIFO INTERFACE can be used to tell the compiler that function arguments must act as FIFO input and output ports after synthesis or PIPELINE INTERFACE can be used to apply different levels of parallelism in design. Also, Vivado HLS acquires some helpful C libraries to ease hardware designing. These libraries are included:

- **Arbitrary Precision Data Types**: Allow specification of arbitrary precision bit width as opposed to software programming where we only use data types with 8-bits boundaries.

- **HLS Math**: Supports standard C/C++ math library for synthesising math operation.

- **HLS Video**: It inherits some useful functions from OpenCV library for image and video processing purposes.

- **HLS Stream**: Data streaming acts as a FIFO with infinite depth where data samples are sent in sequential order starting from first value. This library provides an infrastructure for streaming data.

- **HLS DSP**: It includes building block functions for DSP system modeling in C++.

### 2.3.2 SystemC synthesis

SystemC is a library of C++ classes which understands hardware and its behaviors such as concurrency, bit accuracy and simulation time advancement [17]. The most prominent feature of SystemC is that it uses syntax and specification algorithms which are common in both hardware and software, thereby setting up a basis for hardware-software co-design. Generally, C++ libraries, data types and structures can be used in a SystemC design alongside some provided mechanism for hardware implementation analogous to HDLs [30]. The clock signal is an intrinsic part of hardware modeling in FPGA and usually, HDLs are good choices to tackle this issue. Most HLS toolchains lack this ability which is one of their disadvantages. However, same as Clash, the provided mechanisms in SystemC are capable of handling the timing and multiple clock domain concepts. Also, SystemC has a rich source of data types and it supports arbitrary precision representation in addition to standard C++ data types.

A SystemC description includes several different connected modules and each of them demonstrates some functionality [14]. The top-level function of the design should be defined within SC_MODULE for synthesizing the target. Also, these modules are connected to each other by using ports and signals. Furthermore, the interface synthesis is not supported for SystemC design as I/O ports are fully
described in `SC_MODULE` and their behavior is specified in source code. For instance, a regular SystemC design in Vivado HLS has top-level module and testbench module and to simulate our design, we have to connect these two modules by signal ports. Figure 2.7 is a visual representation of this construction.

![Figure 2.7: Modules A and B are instantiated within Module C. Signal s is used to connect ports of A and B.](image)

The functions of the design and their behavior are specified in SystemC constructor processes (Threads) which run concurrently and are sensitive to signals and clock edges. SystemC supports three kinds of threads namely `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`. However, Vivado HLS only aids `SC_METHOD` and `SC_CTHREAD`.

`SC_METHOD` behaves like a function call analogous to `@always` block in Verilog. It executes once every time triggered by an event, runs continuously, and would be appropriate for combinational and simple sequential circuits. `SC_CTHREAD` is the most useful thread for high-level design which is just sensitive to a clock edge and can take one or more clock cycles to execute a single iteration.

A simple design model for implementation of AND function is illustrated as below where `a` and `b` are inputs and `result` is the output. Also notice that the data type is unsigned int with a width of 2 bits.
```cpp
#include <systemc.h>
SC_MODULE (AND)
{
    sc_in< sc_uint<2> > a ;
    sc_in< sc_uint<2> > b ;
    sc_out< sc_uint<2> > result ;
    sc_int< bool > clk ;
    void func()
    {
        result.write( a.read() & b.read() );
    }
    SC_CTOR (AND)
    {
        SC_METHOD (func);
        sensitive << clk.pos();
    }
}
```

Table 2.3: SystemC AND implementation
Chapter 3

Related work

In recent years, the proliferation of high-performance design and energy-efficient systems has made HLS a famous approach for dealing with this issue. So, several Domain Specific Languages (DSL) have been developed to cope with the difficulties exist in synthesizing general-purpose languages. In this chapter, we will explore how these DSLs operate and their structure. Also, we will go through the research work which proposed comparisons for HLS tools and languages in different categories such as throughput and performance.

3.1 Overview of Other General Purpose HLS

In this section, we examine an overview of general-purpose HLS tools and languages except for Clash, C++ and SystemC which were discussed in detail in the previous chapter. All these HLS, accept general purpose programming languages as their input. A wide range of commercial and academic HLS tools have been developed in recent years, so in this section, we present a brief review of some of them.

**Kiwi**: Kiwi [34] is a parallel programming library that inherits its syntax from the C language and along with its synthesis system, it can transform C to a corresponding Verilog file that would be suitable for FPGA systems. Also, it provides the ability to analyze or verify the high-level representation in C or the generated RTL netlist.

**Lime**: Lime [7] is a Java-based language that is executable on FPGAs and CPUs. It exploits both ”micro-functional” and ”macro-functional” features that integrate with an imperative style which makes Lime be similar to functional HLS and imperative HLS. The micro-functional focuses on recursively immutable value types while macro-functional is based on isolated tasks that internally allow arbitrarily complex sequential imperative code. Also, it supports different levels of parallelism and other optimization options such as arbitrary data types and bounded arrays.

**LegUp**: LegUp [11] is a synthesis tool same as Vivado HLS which has been developed in the University of Toronto. The input is a C/C++ program and the output is a circuit specification in Verilog. It used to support most FPGA vendors but recently, it has been acquired by Microchip and the support for other FPGAs has been deprecated from this tool. LegUp includes some C/C++ libraries that allow the creation of efficient hardware such as streaming library, C arbitrary bit-width data type, bit-level operation library and C++ arbitrary-precision data types.

**Hume**: Hume [33] is a general-purpose, functionally inspired, programming
language, initially oriented to resource-aware embedded applications, to implement fine-grain parallel applications on FPGAs by compiling a subset of Hume to Synthesizable VHDL. By using the polymorphic concepts in functional languages, Hume is distinguished by an explicit separation of coordination and expression layer which is based on synchronous finite state boxes connected by single buffered wires. This layer expression defines control within boxes and is based on pattern matching on input values to enable general recursive actions to generate output value.

**ReWire:** ReWire [32, 4] is an open source functional programming language that borrows its syntax and semantic from functional language Haskell and also a compiler that translates high level design directly into FPGA hardware. So, same as Clash, ReWire inherits most of Haskell’s qualities. Due to its Haskell-based root, ReWire supports a certain class of monads called reactive resumption monads which embody the semantic essence of clocked, sequential and reactive computation. Figure 3.1 shows ReWire compilation steps. Targeting PreHDL instead of VHDL directly allows to implement of the necessary code transformation passes on a much smaller language than VHDL itself such as ”loop flattening”.

![Figure 3.1: ReWire Compilation Process](image)

**Impulse C:** The authors of [42] evaluate the performance of an implemented computed tomography (CT) algorithm in VHDL and Impulse C which is a C-to-FPGA tool flow. The Impulse C tool includes the CoDeveloper C-to-FPGA tools and the CoDeveloper Platform Support Package (PSP). PSPs add platform-specific capabilities to the Impulse CoDeveloper programming tools. With the PSP, users can partition the application between the host and co-processors, make use of other on board resources, and automatically generate the required interfaces. The obtained results demonstrate that Impulse C design can achieve more than 61x improvement over multi-threaded applications and have similar performance as VHDL while reducing the design difficulties.

### 3.2 Evaluation of Different High-Level Synthesis Tools

The authors of [27] proposed a comprehensive analysis of different academic HLS tools that have been developed in recent years. It allocates a section to explore current optimizations in HLS tools and categorized them to:

- **Operation Chaining** which performs operation scheduling within the target clock period.

- **Bitwidth Analysis and Optimization** reduce the number of bits required by datapath operators.
• **Memory Space Allocation** that allows designer to partition and map software data structures onto dedicated BRAMs.

• **Loop Optimizations** makes loop iteration to start before the completion of its predecessor, provided that data dependencies are satisfied.

• **Hardware Resource Library** determines how to implement each operation.

• **Speculation and Code Motion** is a technique that allows operations to be moved along their execution traces.

• **Exploiting Spatial Parallelism** instantiates multiple hardware units that execute concurrently.

• **If-Conversion** is a well-known software transformation that enables predicated execution, i.e., an instruction is executed only when its predicate or guard evaluates to true.

Also, it provides a comparison between three academic HLS tools, LegUp, Bambu and, Dwarv and one commercial HLS tool. Two experiments are performed for this purpose, one with a default setting where there is no specific optimization used and another one where compiler performance optimization is used. In addition to area utilization, three other metrics used to evaluate the performance of the circuit namely **maximum frequency**, **cycle latency** and **wall-clock time** (minimum clock period × cycle latency). Based on obtained results, it can be derived that there is no salient difference between these HLS tools in terms of quality and that no single tool produced the best results for all benchmarks, but it should be noted that commercial HLS tool supports more features, while at the same time, being more stable than the academic tools.

Intel HLS Compiler, Clash, and Intel OpenCL SDK are three HLS tools that all use three different paradigms are investigated in [35]. Intel HLS compiler uses regular C++ an imperative language, Clash is a subset of Haskell which is also the focus of this thesis and Intel OpenCL SDK provides support for the OpenCL, a concurrent language written for heterogeneous computing systems like Graphics Processing Unit (GPU). The throughput of the same code on FPGA after synthesis is evaluated based on the implementation of MD5 hashing algorithm and positive floating-point addition as two separate benchmarks. The performance results which only include area utilization and maximum frequency generated by Quartus are highlighted in Table 3.1 and 3.2. Based on the fact that Intel OpenCL SDK cannot be used to generate result’s report for comparison and because of the long compilation time, the exact performance of the Intel OpenCL SDK was dropped and not listed in these two tables. It can be seen that the Intel HLS compiler performs better in both utilization area and maximum frequency. On the other hand, despite the performance result, based on examination of the tool on its ease of use, it was

<table>
<thead>
<tr>
<th>F&lt;sub&gt;max&lt;/sub&gt;(MHz)</th>
<th>Area(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clash</td>
<td>3.13</td>
</tr>
<tr>
<td>i++</td>
<td>113.77</td>
</tr>
</tbody>
</table>

Table 3.1: MD5 hashing algorithm
<table>
<thead>
<tr>
<th></th>
<th>$F_{\text{max}}$ (MHz)</th>
<th>Area(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clash</td>
<td>78.03</td>
<td>0.76</td>
</tr>
<tr>
<td>i++</td>
<td>268.67</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table 3.2: Floating point addition

concluded that the functional paradigm is the best fit since Clash has the most straightforward transformation from source to hardware.

A variety of research work have been done to compare the implementation efficiency of HLS and HDL tools. As an example, [36] presented a comparative analysis of implementing Galois Field (GF)$\left(2^n\right)$: division and multiplication in both Vivado HLS and HDL tools. It proves that HDL has better results regarding hardware resource area compared to HLS thanks to the ability of HDL in providing perfect control over the entire design at the cost of deep knowledge for digital electronics. However, code specification is much more simple and intuitive in HLS which enhances the productivity of hardware design. Figure 3.2 shows the summary of hardware resources obtained through synthesizing HLS implementation whereas figure 3.3 illustrates these results for HDL implementation.

![Utilization report from Vivado HLS](image)

Figure 3.2: Utilization report from Vivado HLS

![Utilization report from Vivado HDL](image)

Figure 3.3: Utilization report from Vivado HDL

The authors of [40] evaluate the Xilinx and Altera HLS tools which are two prevalent FPGA vendors among developers. They implement a tri-diagonal linear system solver as a test case in both HLS tools and provide related results in case of utilization and accuracy. As mentioned before, Vivado HLS uses standard C code and Altera supports Open Computing Language which is an open standard parallel programming language. The tri-diagonal linear system is represented as four arrays with $n$ elements in the form of table 3.3.
\[
U = [0, a_{1,2}, a_{1,3}, ..., a_{(n-1), n}]
\]
\[
D = [a_{1,1}, a_{2,2}, ..., a_{n,n}]
\]
\[
L = [a_{2,1}, a_{3,2}, ..., a_{n,(n-1)}, 0]
\]
\[
b = [b_1, b_2, ..., b_n]
\]

Table 3.3: Tri-Diagonal linear system

\[L, U \text{ and } D\] are the lower, upper and diagonal bands of the coefficient matrix and \(b\) is the right hand side vector. Figure 3.4 shows the logic utilization of RTL code generated by these tools against hand-coded VHDL implementation. The resources usage depends on the maximum-sized tri-diagonal linear system that is required to be solved. As it can be seen, VHDL has more LUTs and Flip Flops due to the reason that BRAM modules were deliberately were not used for simplification purposes, and also the VHDL is double-buffered, so two tri-diagonal systems are stored. Furthermore, the number of used resources in Xilinx HLS tool are significantly less than Altera OCL.

3.3 DSLs in high level synthesis

3.3.1 Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines

The authors of [18] proposed Darkroom which is a language and compiler for image processing. Figure 3.5 shows the Darkroom compilation process.

Operations can be done on images which is defined as a pure function from 2D coordinates to the values at those coordinates based on \textit{lambda}-like semantic. The high-level program in Darkroom transforms to line-buffered pipeline where it consumes one input and generates one output every clock cycle. The Darkroom compiler is based on Terra language. After generating an optimized line-buffered pipeline, the compiler maps it to efficient FPGA or Application-Specific Integrated Circuit (ASIC) design, or CPU code. According to the obtained result, after implementing the optimized output from Darkroom on a mid-range FPGA, throughput of over 125 megapixels/sec can be achieved.

3.3.2 Optimus: Efficient Realization of Streaming Applications on FPGAs

Optimus which is an optimizing synthesis compiler for streaming applications was proposed in [19]. This compiler accepts \textit{StreamIt} as its input language and pro-
duces an efficient HDL that can be fed to FPGA. This framework is suitable for streaming design such as signal processing applications. Optimus introduces a set of new macro-functional optimization. The latency can be improved by sending a larger chunk of data between filters. Also, Optimus reduces memory storage by intelligently sizing queues between filters and dedicating output registers to increase spatial reuse.

It can be derived from this research work that Optimus automatically applies area and timing optimizations and eliminates the manual procedure of applying these optimizations which has been done in some HLS tools.

### 3.3.3 HIPAcc: A Domain-Specific Language and Compiler for Image Processing

The authors of [25] developed another domain-specific compiler which is called HIPAacc. It uses C++ semantics and classes for defining components and all operations on images are confined to C++ classes. The output of the compiler targets low-level code for discrete GPUs based on the LLVM backend. Also, in recent versions of HIPAacc, it can generate C code annotated with pragmas (optimization directives) for Vivado HLS.

### 3.3.4 RIPL: A Parallel Image Processing Language for FPGAs

RIPL which is another high-level image processing DSL for FPGAs is presented in [37]. RIPL provides a collection of image processing skeletons for element-wise computation, sliding 1D/2D stencils, image reduction operations, recursive algorithms and random access into 2D images and N dimensional arrays. It is inspired by functional programming languages such as Haskell by exploiting some stream based functions e.g. `map` and `zipWith`. For assessing and comparing RIPL performance vs. Vivado HLS OpenCV library, different benchmarks have been designed.
in this paper. Obtained results show that RIPL outperforms Vivado HLS in image brightening and histogram normalization, achieving 191 Frame per Second (FPS) versus 126 FPS. However, Vivado HLS has higher performance in some other image processing algorithms such as Sobel filtering. Also, RIPL is 5x shorter than equivalent implementation in C++.

3.3.5 High-Level Synthesis from C vs. a DSL-based Approach

A special purpose language and compiler called Language for Aggressive Loop Pipelining (LALP) is focused on mapping loop structures into reconfigurable hardware and we can categorize it as a DSL, proposed in [15]. The output of this compiler is a synthesizable VHDL code that can be imported to different FPGAs. Figure 3.6 displays the compilation flow of LALP. LALP has C-like syntax and allows developers to use loop pipelining techniques. It has been developed based on the approach mentioned in [26] where minimizing data flow latency in loop iterations is the main purpose and data path operations are synchronized and balanced using cascades of registers. Also, this paper created thirteen benchmarks to compare the generated hardware by LALP with two C-based HLS tools, Vivado HLS and LegUp [11]. Results show that the use of LALP in most cases has better performance.

3.3.6 Conclusion

It can be concluded that DSLs have been emerged for some particular applications such as image processing, signal processing, streaming application and to name a few. DSLs are a perfect fit for the software domain as they let programmers define solutions at a higher abstraction level than typical general purpose languages within the semantics of the particular application domain. Hence, using DSLs can fill the gap for software developers who have little knowledge of hardware design. Another advantage of DSLs is that the domain-knowledge conveyed by DSLs can be used to extract application parallelism, perform optimization and identify a suitable system architecture for implementation automatically as opposed to general purpose HLS where the user still needs to declare optimization using annotated pragmas. However, they cannot handle general-purpose applications such as mathematical algorithms that have been used in our work, so they are outside the scope of this thesis and other imperative and functional languages are better options for this goal.
Chapter 4

Implementation

4.1 FIR Filter

In this section, we will explore the implementation of FIR filter as a trivial application in FPGA design and provide a general overview of high-level synthesis in Vivado HLS and Clash with three programming languages, C++, SystemC and Clash.

Equation (1) expresses the discrete-time domain of the FIR filter where $y[n]$ is the output, $x[n]$ is the input sample, $h[n]$ is the coefficient and $L$ is the number of filter tap.

\[
y[n] = x[n] \ast h[n] = \sum_{i=0}^{L-1} x[i]h[n - i] \quad (1)
\]

4.1.1 Vivado HLS Implementation of FIR Filter

This section describes the implementation of FIR in Vivado HLS using C++ and SystemC languages. The top-level module which is called FIR is implemented separately in both C++ and SystemC. For consistency and simplification of the design, Int has been used as a data type in all three designs. Furthermore, Clash does not support floating-point data types for implementation and fixed-point data type is not well-suited on SystemC when C/RTL Cosimulation is running which causes some issues in Vivado HLS.

C++ Implementation

Table 4.1 shows the top-level module(function) of the FIR filter. As it can be seen, the only code constraint used is ”HLS PIPELINE” which exploits the parallelism optimization feature of Vivado HLS which allows all operations to happen concurrently that results in throughput improvement. Also, the input and output are implemented as an FIFO interface. ”–SYNTHESIS–” macro has been used to exclude non-synthesizable code from the design because ”cout” is a system call and cannot be synthesized into RTL and it only applied for simulation purposes [5].
```cpp
#include "FIR.h"

void FIR(fnum *x, fnum *y)
{
    #pragma HLS INTERFACE ap_fifo depth=31 port=y
    #pragma HLS INTERFACE ap_fifo depth=31 port=x

    for(int i = 0; i < size; i++)
    {
        #pragma HLS PIPELINE
        fnum acc = 0;
        for(int j = N-1; j > 0; j--)
        {
            shift_reg[j] = shift_reg[j-1];
        }
        shift_reg[0] = *(x + i);
        for(int j = 0; j < N; j++)
        {
            acc += shift_reg[j] * coeff[j];
        }
        #ifndef __SYNTHESIS__
            cout << "reg: " << shift_reg[j] << endl;
            cout << "coeff: " << coeff[j] << endl;
        #endif
    }
    *(y++) = acc;
}
```

Table 4.1: FIR filter in C++

**SystemC Implementation**

Same as C++ design, FIR filter design in SystemC is also described in table 4.2. Again for consistency, the FIFO used as an interface. The top-level module "FIR" has two processing functions in SystemC namely "Prc1" and "Prc2" where read and write tasks are performed respectively. Similar to C++ design, the "HLS PIPELINE" pragma used for hardware optimization.

**4.1.2 Clash Implementation of FIR Filter**

Clash implementation of FIR filter is more straightforward thanks to higher-order functions in the functional language Haskell and we can create our FIR filter in just five lines of code in comparison to imperative languages that take more than this. Table 4.3 displays Clash code which consists of FIR function, top-level module "topEntity" and related testbench. FIR filter takes successive dot products of the input data with a vector of coefficients, hence, dot product function is created with higher-order function "zipWith". As illustrated, fir function takes two inputs "x"
```c
#include "FIR.h"
int shift_reg[N];
void FIR::Prc1()
{
  write_done = false;
  wait();
  while(true)
  {
    #pragma HLS PIPELINE
    while (!start.read()) wait();
    write_done = false;
    for(int i = 0; i < N; i++)
    {
      for(int j=N-1; j > 0; j--)
      {
        shift_reg[j] = shift_reg[j-1];
      }
      shift_reg[0] = din.read();
    }
    write_done = true;
    wait();
  }
}
void FIR::Prc2()
{
  int coeff[N] = {1, 4, -6, -11, 15, 17, 8, 9, 12, 19, 21};
  int out_val = 0;
  done = false;
  wait();
  while(true)
  {
    #pragma HLS PIPELINE
    while (!start.read()) wait();
    wait();
    while (!write_done) wait();
    for(int j = 0; j < N; j++)
    {
      out_val += coeff[j] * shift_reg[j];
      dout.write(out_val);
    }
  }
  ifndef __SYNTHESIS__
  cout << "out: " << out_val << endl;
  cout << "reg: " << shift_reg[j] << endl;
  endif
  done = true;
  wait();
}
```

Table 4.2: FIR filter in SystemC

21
and "coeff" and generates output "y". "window" is a function that takes a signal
to create a window over of at least size 1. Also, it can be seen that there is no
type annotation used for this design as Clash can determine the data type with its
intelligent type system.

"topEntity" is the top-level module that will be synthesized into RTL code and
we have to give it an explicit type inference to become monomorphic and it includes
fir function follow with its constant coefficients.

The process of testbench creation is almost same in all Clash designs where we
define "testInput" which are the input signals of our FIR filter along with "expected-
dOuput" which highlights our desired output from the FIR filter.

After compiling this code snippet in Clash compiler, the corresponding Verilog
or VHDL file and RTL testbench file will be created in separate folders.

module FIR where
import Clash.Prelude
import Clash.Explicit.Testbench

dotp a b = sum (zipWith (*) a b)
fir coeff x = y
  where
    y = dotp coeff z
    z = window x

topEntity
  :: Clock System
  -> Reset System
  -> Enable System
  -> Signal System (Int)
  -> Signal System (Int)
topEntity = exposeClockResetEnable
{-# NOINLINE topEntity #-}
testBench :: Signal System Bool
testBench = done
  where
    testInput = stimuliGenerator clk rst
      (0:1:2:3:4:5:6:7:8:9:10:Nil)
    expectedOutput = outputVerifier ' clk rst
done = expectedOutput (topEntity clk rst (enableGen) testInput)
    clk = tbSystemClockGen (not <$> done)
    rst = systemResetGen

Table 4.3: FIR filter in Clash
4.2 Eigenvalue and Eigenvector Decomposition Algorithm

Eigenvalue calculation arises in many fields of computer vision, image processing, signal processing and wireless sensor networks where real time performance is usually required. EVD would be a great fit for such applications where matrices are real and symmetric which makes EVD calculation becomes easier. One of the main applications where symmetric eigenvalue calculation occurs is multiple signal classification (MUSIC) [41] which uses a covariance matrix.

There are different kinds of approaches that can be used to solve EVD where most of them are iterative. In this study, Jacobi algorithm is employed for this purpose as its characteristic is suitable for parallel programming. In addition, most of the operations in the Jacobi method can be accomplished by the Coordinate Rotation Digital Computer (CORDIC) algorithm by using shift and add operations that occupy a small amount of hardware resources compared to multiplication and division operators. In particular, for the Jacobi method, the CORDIC method has been used to make the calculation of the angle and double rotation.

So in this thesis, first we implement a covariance matrix of a vector of inputs, then the eigenvalue decomposition of the corresponding covariance matrix will be implemented in both HLS tools with three mentioned languages.

4.2.1 Covariance Matrix Implementation

A covariance matrix is a square and symmetric matrix and basically, it is described as the multiplication of a vector and its transpose and it can be calculated as:

\[ R_x = \sum_{i=1}^{M} x(i)x^T(i) \quad (2) \]

where M is the number of vector elements. Covariance matrix calculation can be done using a multiply-accumulate function in parallel. Since the covariance matrix is a symmetric matrix, we can only use the upper or lower triangle elements of the matrix for further EVD operation.

4.2.2 CORDIC Algorithm

Coordinate Rotation Digital Computer (CORDIC) is a simple and efficient algorithm for computing trigonometric functions where there is no multiplier available and it only requires bit-shift operator, lookup table, addition and subtraction [22]. The idea behind the CORDIC is to execute a set of rotations on an input vector to obtain another vector as output.

The vector \((x, y)\) can be rotated by a rotation angel \(\alpha\) as:

\[ \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (3) \]

We can simplify and extend equation (3) to be decomposed to iterative rotations by angle \(\alpha_i\) \((i = 0, 1, ..., n)\), so:
\[
\begin{bmatrix}
  x^{(i+1)} \\
  y^{(i+1)}
\end{bmatrix} = \cos \alpha_i \begin{bmatrix}
  1 & -\tan \alpha_i \\
  \tan \alpha_i & 1
\end{bmatrix} \begin{bmatrix}
  x^{(i)} \\
  y^{(i)}
\end{bmatrix} \tag{4}
\]

We can choose \( \alpha_i \) in a way that \( \tan \alpha_i = 2^{-i} \), then:

\[
\alpha_i = \arctan 2^{-i} \tag{5}
\]

\[
\alpha = \sum_{i=0}^{n} d_i \alpha_i \quad (d_i = \pm 1) \tag{6}
\]

Where \( d_i \) is the sign of CORDIC rotation. Now, equation (4) can be rewritten as:

\[
\begin{bmatrix}
  x^{(i+1)} \\
  y^{(i+1)}
\end{bmatrix} = \cos(\arctan 2^{-i}) \begin{bmatrix}
  1 & -d_i 2^{-i} \\
  d_i 2^{-i} & 1
\end{bmatrix} \begin{bmatrix}
  x^{(i)} \\
  y^{(i)}
\end{bmatrix} \tag{7}
\]

For Jacobi algorithm, the rotation angle can be restricted to \( \pi/4 \) and table 4.4 shows all possible values for \( \alpha_i \). Also \( \cos(\arctan 2^{-i}) \) can be depicted as:

\[
\cos(\arctan 2^{-i}) = \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{8}
\]

Due to the equation (8), the product of \( \cos(\arctan 2^{-i}) \) is equal to \( 1/k \) where \( k = \prod_{i=1}^{n} \frac{1}{\sqrt{1 + 2^{-2i}}} \) and \( k \) converges to 1.647. Therefore, we can ignore \( \cos(\arctan 2^{-i}) \) in every iterations and the final vector will be scaled by 0.60725.

Therefore, the summary of all equations in CORDIC algorithm are shown below:

\[
x^{(i+1)} = x^i - d_i 2^{-i} y^i \\
y^{(i+1)} = d_i 2^{-i} x^i + y^i \tag{9}
\]

\[
z^{(i+1)} = z^i - d_i \alpha_i 
\]

The last expression is known as angle tracker.

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{-i} )</td>
<td>1</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/32</td>
<td>1/64</td>
<td>1/128</td>
<td>1/256</td>
<td>1/512</td>
</tr>
<tr>
<td>( \arctan 2^{-i} )</td>
<td>45</td>
<td>26.6</td>
<td>14</td>
<td>7.1</td>
<td>3.6</td>
<td>1.8</td>
<td>0.9</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 4.4: Rotation angles

### 4.2.3 Traditional Jacobi Method

Jacobi method can be used to obtain the first diagonal \( D \in \mathbb{R}^{m \times n} \) of an original matrix \( A \in \mathbb{R}^{m \times n} \) by performing the set of rotations as shown below [39]:

\[
A = UDV^T \tag{10}
\]

Where \( U \in \mathbb{R}^{m \times m} \) and \( V \in \mathbb{R}^{m \times m} \) are orthogonal and \( D \in \mathbb{R}^{m \times n} \) is diagonal with non-negative diagonal elements. By considering the Hermitian Matrix \( A \in \mathbb{R}^{m \times m} \), the singular values are equal to eigenvalues, we can reformulate equation (10) as:

\[
A = UDU^T \tag{11}
\]
Where the diagonal matrix \( D \in \mathbb{R}^{m \times m} \) will be the eigenvalues and the orthogonal matrix \( U \in \mathbb{R}^{m \times m} \) will be the eigenvectors.

The Jacobi method performs the sequence of orthogonal similarity rotation. Each rotation will eliminate one the off-diagonal elements. After each iteration, the new generated matrix \( A_{(L-1)} \) is more diagonal than its predecessor. The iteration of the Jacobi method can be illustrated as equation (12):

\[
A^{(1)} = U_1^T D U_1 \\
A^{(2)} = U_2^T D^{(1)} U_2 \\
\vdots \\
A^{(L)} = U_L^T D^{(L-1)} U_L
\]

Where \( L \) is the number of iterations. So,

\[
A^{(L)} = U'^T D U'
\]

Where \( U'^T = U_1^T U_2^T \ldots U_L^T \) and \( U' = U_1 U_2 \ldots U_L \). After \( L \) iterations, \( D^L \) is almost diagonal. The diagonal elements of \( D^L \) represents the eigenvalues and the corresponding eigenvectors are the columns of \( U' \).

This classical Jacobi method looking for the upper triangle in each iteration and sets the largest off-diagonal element to zero. This strategy is sensible for hand calculation, but it is resource-intensive for hardware implementation as the search procedure of largest off-diagonal makes each rotation a process of order \( N^2 \) instead of \( N \). Therefore, this is a prohibitive algorithm for FPGA that consumes a huge amount of resources, that why we considered the improved Jacobi method which will be discussed in the next sub-section.

### 4.2.4 Improved Jacobi Algorithm for EVD

An improved Jacobi method was proposed in [39], [22] and [20] which reduces the delay of the traditional Jacobi algorithm. This new parallel Jacobi divides the upper triangle of the original matrix into submatrices of \( 2 \times 2 \) elements which causes two computation modes, diagonal and non-diagonal.

The angle \( \theta_i \) in each iteration of the CORDIC rotation can be determined by:

\[
\theta_i = \theta - \sum_{j=0}^{i-1} d_i \alpha_i
\]

Where \( \alpha_i = \arctan(2^{-i+1}) \) (\( i = 1, 2, \ldots, k \)) and \( d_i \in \{-1, 1\} \). \( d_i \) which is also known as rotation direction can be established by the sign of \( \theta_i \). As mentioned before, the rotation angle of CORDIC for Jacobi algorithm is restricted to \( \pi/4 \) which means that sign of \( \theta_i \) is same as \( \tan(2\theta_i) \) because it is small. Hence, \( d_i \) can also be determined by the sign of \( \tan(2\theta_i) \) and based on trigonometric rules, it can be computed as:

\[
\tan(2\theta_i) = \frac{%25 \beta_i}{\lambda_i} = \tan(2\theta_{i-1} - 2d_i \alpha_i) = \frac{\tan(2\theta_{i-1}) - d_i \tan(2\alpha_i)}{1 + d_i \tan(2\alpha_{i-1}) \tan(2\alpha_i)}
\]
By substituting $\tan(2\alpha_i) = \frac{2^{1-i}}{1-2^{-2i}}$, $\tan(2\theta_i) = \frac{\lambda_i}{\beta_i}$ and $\tan(2\theta_{i-1}) = \frac{\lambda_{i-1}}{\beta_{i-1}}$ we can rewrite equation (14) as:

$$\frac{\lambda_i}{\beta_i} = \frac{(1 - 2^{-2i})\lambda_{i-1} - d_i2^{1-i}\beta_{i-1}}{(1 - 2^{-2i})\beta_{i-1} + d_i2^{1-i}\lambda_{i-1}}$$

(15)

According to equation (15), the sign of $d_i$ can be determined by the sign of $\frac{\lambda_i}{\beta_i}$. When $\lambda_i$ and $\beta_i$ have different signs, the sign of $d_i$ is -1, otherwise the sign of $d_i$ is +1. For hardware implementation, $d_i = -1$ is considered as 0.

It can be seen this sign set recognition can be computed by CORDIC algorithm inside diagonal elements with shift and add operations. Figure 4.1 shows this procedure in hardware implementation.

![Figure 4.1: Calculation of $\lambda_i$ and $\beta_i$ based on Eq. (12)](image)

After the calculation of this sign set, the off-diagonal elements receive the sign set and each element can be updated by rotation mode of CORDIC algorithm which described in equation (9) earlier and new off-diagonal elements will be scaled by the factor $k = 0.60725$. The hardware implementation of the CORDIC rotation mode is illustrated in figure 4.2.

![Figure 4.2: CORDIC rotation](image)

Based on the related equation to classical Jacobi method studied in [22], the diagonal elements of the symmetric matrix can be updated by:

$$a_{ii}^n = a_{ii}^{(n-1)} - a_{ij}^{(n-1)}\tan\theta_n$$

(16)

26
\[ a_{jj}^n = a_{jj}^{(n-1)} + a_{ij}^{(n-1)} \tan \theta_n \]  

(17)

Where the value of \( \tan \theta_n \) can be stored in a look-up table with constant values.

### 4.2.5 Systolic Array

A systolic array design is proposed in [39] and [22] to enhance parallel Jacobi implementation. As mentioned before, in this thesis EVD will be performed on a 4 × 4 symmetric matrix, so the represented systolic array is shown in figure 4.3. It contains three Processing Element (PE) and each of them includes a 2 × 2 sub-matrices. The PE1 and PE3 located on the diagonal line are called diagonal processors and PE2 is called off-diagonal processor. Using sign set computation shown in figure 4.1, the two diagonal processors update the four diagonal elements and send the desired values of \( d_i \) to the right and the top. Then the off-diagonal processor updates the diagonal elements using the CORDIC rotation method displayed in figure 4.2. After all elements are updated, they will be relocated along the thin arrows and then PEs will start the next iteration. This process can considerably reduce the total EVD computation time.

![Figure 4.3: Proposed Systolic array from [22]](image)

For hardware implementation purposes, the diagonal processors consist of two modules, one of them computes the direction of rotation value \( d_i \) and the other one uses this \( d_i \) value to get the tangent value from the look-up table and update the diagonal elements. Similarly, an off-diagonal processor consists of 4 modules using the CORDIC rotation method to update off-diagonal elements. The accuracy of the CORDIC algorithm depends on the number of iteration. In this thesis, we
conduct 10 iterations in each design to meet the high accuracy. Therefore, after 10 iterations, the upper triangle of the diagonalized matrix will be found where the diagonal elements represent the eigenvalues.

For the calculation of eigenvectors, there are 8 modules that perform the CORDIC rotation method in parallel. 4 modules will update column 1 and 2 of the matrix and the other 4 will update column 2 and 4. After each iteration, a data exchange operation will be done between the columns as shown in the figure 4.4. So the corresponding eigenvectors can be found simultaneously with eigenvalues.

![Figure 4.4: Data exchange between each column](image)

**4.3 Clash Implementation of EVD**

[22] provided a Clash implementation of multiple signal classification (MUSIC) algorithm. As mentioned earlier, one of the usages of eigenvalue decomposition algorithm is on MUSIC algorithm where, first, we should calculate the covariance matrix of the input signals and then the corresponding eigenvalue and eigenvector of the covariance matrix can be computed.

Since Clash has been undergone a lot of changes in functions, syntax and also some features have been added to it such as testbench generation, so in this thesis, we have modified a big portion of the code and the design which were proposed in the EVD section of [22] to become compatible with new Clash version. The implementation of EVD algorithm in Clash includes different modules and each of them describes the calculations of the previous section.

The code snippet related to covariance matrix calculation is shown in table 4.5 and mentioned as (CMC) module. As it can be seen, the ”mmMult” function calculates covariance matrix and it has one input with the type of ”Matrix1” which means a matrix with 1 row and 4 columns indicates the input signals. ”cmcCore” is the main module that changes the form of a $4 \times 4$ matrix to the type indicated by ”CMCR” which contains only upper-triangle values of the matrix. ”concat” which is a pre-defined function in Clash and it converts a $m \times n$ matrix to a vector with $m \times n$ elements. Finally, ”uptriangle” function finds the upper-triangle values in the vector produced by ”concat”. By employing higher-order functions like ”zipWith”,
"map" and "concat" and also function composition feature in Clash, we can see that with a few lines of code we are able to declare matrix multiplication which significantly reduces the effort of writing a program.

The "CordicA" module describes the equation (15) which calculates \( \lambda_i \) and \( \beta_i \) to find the sign of rotation direction \( d_i \). The definition of this module is almost the same as [22], however, some functions such "getSign" have been modified. Also, it is clear that all the computations related to the CORDIC algorithm are performed by right shift and addition operators. For high accuracy, we implement 10 iterations of CORDIC algorithm. Table 4.6 shows CordicA module.

After determining \( d_i \) with "CordicA" module, the "update" module used to update the diagonal elements of the covariance matrix based on the equation (16) and (17). Table 4.9 shows the definition of update function. As shown in the "update" module, "tanj" is the tangent value obtained from a list of constant tangent values defined in look-up table (LUT) module which is provided exactly the same in [22] and displayed in table 4.7. Based on "LUT" module, by using list comprehension and concatenation in Clash, "css" function creates a list of \( 2^n \) lists where each sublist contains \( n \) values being either -1 or 1. Also, "bs" is the list of rotation angles in radians shown in table 4.4. Then each generated sublist is applied to the "tangent" function to calculate the corresponding tangent value. Basically, it checks if \( d_i \) is equal to 1, the corresponding angle value in the list "bs" will be multiplied by 1, otherwise, it will be multiplied with -1. The tangent function uses "truncate" function to remove the fractional part of the tangent value. Then if the obtained tangent value is less than 0, it multiplied with -1, otherwise it again multiplied with -1 to prevent negative values.

Also the "resize" module is shown in table 4.8 which demonstrates the custom right shift function "myShiftR" with the appropriate conditions suit for this project.

The next module used to update off-diagonal elements according to Equation (9) is "CordicB". As described before, it uses \( d_i \) produced by the CordicA module. Table 4.10 shows the definition of this module. It is the same as the proposed design in [22].

Finally, we can implement the main module "EVD" which includes all the modules described before as displayed in appendix. The "evdCore" function takes 2 inputs, the initial values of the upper triangle which is depicted with "uptri_init" and also the eigenvectors are instantiated with an identity matrix multiplied with 1000 and mentioned with "evsinit". The second value is the input signal vector. The generated output contains an eigenvalue in the form of a vector with 4 elements, a \( 4 \times 4 \) matrix indicates the obtained eigenvectors and another output "end". If "end" becomes 1, it means that all off-diagonal elements are zeros. The "uptri_tmp" performs the data exchange according to the systolic array shown in figure 4.3. After getting the eigenvectors, "evec1", "evec2", "evec3" and "evec4" perform data exchange between each column of the desired eigenvectors as shown in figure 4.4.

The testbench which was created in [22] is completely different in syntax from the new version of Clash. In the recently published version of Clash, a testbench function is available which can be used to generate RTL testbench. Table 4.10 shows the related code snippet for EVD testbench.
4.4 Vivado HLS Implementation of EVD

The implementation of EVD on Vivado HLS (C++ and SystemC) can be found in the appendix. The structure and the modules that have been used in both designs are the same as Clash implementation. Both designs contain "CMC", "CordicA", "Update", "CordicB" and "EVD" modules. In this thesis, we execute two different experiments with and without optimization and we evaluate all of them.

The optimizations that have been used are HLS Pipeline, Array Partition and Loop Unrolling. The only difference between C++ and SystemC implementation that we should take into account is Interface Synthesis. Unlike C/C++, Vivado HLS does not perform interface synthesis on SystemC and it only does support interface synthesis for some memory interfaces such as RAM and FIFO ports. So in this thesis, we have used FIFO ports on the top-level interface which can be synthesized directly from the standard SystemC ports.
module CMC where

import Clash.Prelude
import Clash.Sized.Vector
import Clash.Explicit.Testbench

type Matrix1 = Vec 1 (Vec 4 (Signed 16))
type Matrix2 = Vec 4 (Vec 4 (Signed 16))
type CMCR = ((Signed 16, Signed 16, Signed 16, Signed 16),
(Signed 16, Signed 16, Signed 16),
(Signed 16, Signed 16),
(Signed 16))

dot vec1 vec2 =
  sum (zipWith (*) vec1 vec2)

mvMult mat vec =
  map (dot vec) mat

mmMult :: Matrix1 -> Matrix2
mmMult mat1 = map (mvMult (transpose mat1)) (transpose mat1)

cmcCore :: Matrix1 -> CMCR
cmcCore ys = inp
  where
    mat = concat $ mmMult ys
    inp = uptriangle mat

uptriangle :: Vec 16 (Signed 16) -> CMCR
uptriangle rs = inp
  where
    inp = ((rs!!0, rs!!1, rs!!2, rs!!3),
    (rs!!5, rs!!6, rs!!7),
    (rs!!10, rs!!11),
    (rs!!15))
module CordicA where

import Clash.Prelude
import Resize

type CordicI = (Signed 16, Signed 16)
type CordicO = Vec 10 Bit

cordic_a :: Signed 16 -> Signed 16 -> CordicO
cordic_a r u = ds

where
  ids = $(listToVecTH ([0..9]::[Int]))
  ((r',u'),ds) = mapAccumL ca (r,u) ids

ca (ri,ui) i = ((ri',ui'),di)

where
  p1 = myshiftR ri (i*2)
  q1 = ri - p1
  q2 = myshiftR ui i
  q3 = myshiftR ri i
  p2 = myshiftR ui (i*2)
  q4 = ui - p2
  di = getSign ri ui
  ri' = addSub di q1 (shiftL q2 1)
  ui' = addSub (complement (di) + 2) q4 (shiftL q3 1)

addSub 1 a b = a + b
addSub 0 a b = a - b

getSign x y = if head (bv2v (pack x))
  == head (bv2v (pack y)) then 0
  else 1

module LUT where

import Prelude

css 0 = [[]]
css n = concat [[-1:cs, 1:cs] | cs <- css (n-1)]
tangent cs = truncate $ 1024 * (tan $ sum $ zipWith (*) bs cs)
lut :: Int -> [Int]
lut n = map tangent (css n)
as = [45.0, 26.6, 14.0, 7.1, 3.6, 1.8, 0.9, 0.4, 0.2, 0.1]
bs = [pi/180*x | x<-as]
```haskell
module Resize where
import Clash.Prelude

myshiftR :: Signed 16 -> Int -> Signed 16
myshiftR inp n | n == 0 = inp
| n > 15 = 0
| bv2v (pack inp)!!(n-1) == 0 = (shiftR inp n)+1
| otherwise = (shiftR inp n)

myext :: Signed 16 -> Signed 32
myext x = resize x

mytrunc :: Signed 32 -> Signed 16
mytrunc x = resize x

module Update where
import Clash.Prelude
import LUT
import Resize

type UpdateI1 = (Signed 16, Signed 16, Signed 16)
type UpdateI2 = Vec 10 Bit
type UpdateO = (Signed 16, Signed 16)

update :: UpdateI1 -> UpdateI2 -> UpdateO
update (a, b, c) ds = (b', c')
where
  tanv = getTan (bitCoerce (v2bv (reverse ds)))
  tmp = mytrunc $ scale1 $ (myext a)*(myext tanv)
  b' = b + tmp
  c' = c - tmp

getTan :: Unsigned 10 -> Signed 16
getTan n = reverse $ (listToVecTH (lut 10)) !! n

scale1 x = shiftR x 10
```

Table 4.8: Resize Module

Table 4.9: Update Module
module CordicB where
import Clash.Prelude

type CordicI1 = (Signed 16, Signed 16)
type CordicI2 = Vec 10 Bit
type CordicO = (Signed 16, Signed 16)

cordic_b :: CordicI1 -> CordicI2 -> CordicO

cordic_b (x, y) ds = (x', y')
  where
    ids = $(listToVecTH ([1..10]::[Int]))
    (xtmp, ytmp) = foldl cb (x, y) (zip ids ds)
    x' = scale 2 xtmp
    y' = scale 2 ytmp

cb (xi, yi) (ind, di) = (xi', yi')
  where
    q5 = shiftR yi (ind-1)
    q6 = shiftR xi (ind-1)
    xi' = addSub di xi q5
    yi' = addSub (complement (di) + 2) yi q6
—scale by 1/K = 0.6075
scale2 x = y
  where
    s1 = shiftR x 1
    s2 = shiftR x 3
    s3 = shiftR x 6
    s4 = shiftR x 9
    s5 = shiftR x 13
    m1 = s1 + s2
    m2 = s3 + s4 + s5
    y = m1 - m2
—rotate according to the direction
addSub 1 a b = a + b
addSub 0 a b = a - b

Table 4.10: CordicB Module
testBench :: Signal System Bool

where


clk = tbSystemClockGen (not <$> done)
rst = systemResetGen
Chapter 5

Results

In this chapter, the results are assessed and they are split into two sections, the evaluation of FIR filter in three HLS languages and the evaluation of EVD algorithm. Three different metrics are used to evaluate the performance of these two benchmarks in this thesis: resource utilization, power consumption and latency (i.e., the number of cycles it takes to produce the output). An important requirement for the HLS tool is that the result of the design is not different after synthesis process. Therefore we need to verify our design in pre- and post synthesis.

In this thesis, despite the ability of Vivado HLS to synthesis the generated RTL by itself, we used a common FPGA vendor environment to synthesis all test cases, Xilinx Vivado Design Suit to have a fair comparison between tools. We also targeted an FPGA part from the Xilinx Zynq-7 family, a mid-range FPGA device with the part number "xc7z030-sbv485-3". To report the total number of resources used in this device, it has 530 BRAMs, 400 DSP48E, about 157k Flip Flops (FF) and 78k LUTs.

Clash generates Verilog or VHDL output file along with a RTL testbench. We can import these files to one of the FPGA vendors IDE that accepts one of the hardware description languages. So, in this thesis after compiling the high-level design on Clash, we import the corresponding top entity and testbench Verilog file to Vivado Design Suite for synthesizing purposes. On the other hand, Vivado HLS provides more options on how to use the generated RTL design. One approach is to package the RTL output as Intellectual Property (IP) that can be used in Vivado RTL environment for further process which provides a convenient way for hardware designers who are familiar with RTL design. Another way is that we can directly copy out the Verilog or VHDL files and use it for RTL synthesis on Vivado RTL environment. The downside of this approach is that we are responsible to correctly use these files and manage them, however, IP packaging performs this procedure automatically. In our experiments, we use the generated files manually as well as testbench files which are created by C/RTL Cosimulation process in Vivado HLS. In the following sub-sections, we will examine the obtained results.

5.1 FIR Filter Evaluation

For consistency and to have the same assessment for all our three designs, the same input signal values and coefficients are considered for the implemented FIR filter in these three designs. Also, the corresponding data type that has been used for
interfaces is "Int" type.

For verification purpose, both HLS tools provide a simulation environment to test whether they produced correct results. As explained earlier, Clash exploits an interactive interpreter where we can simulate our result by entering input values and get output values instantly. The implemented FIR filter in this thesis has coefficients and input signal values as follow:

\[ Coeff = \{1, 4, -6, -11, 15, 17, 8, 9, 12, 21\} \]
\[ Input = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

In order to test our FIR filter in Clash, we can use these values and type the code shown in table 5.1 to Clash interpreter.

```plaintext
simulateN @System 11
[0,1,2,3,4,5,6,7,8,9,10]
```

Table 5.1: Related code to test FIR filter in Clash interpreter

Vivado HLS also provides an option for C/C++ simulation in its Eclipse environment. The related code for C++ and SystemC testbench are depicted in the appendix. After running the C simulation in both Vivado HLS design and comparing the result with the result which was obtained from Clash earlier, we can verify that the all numerical values are consistent and match together.

### 5.1.1 Synthesis Results of FIR Filter

In this sub-section, we explore the post-synthesis result of the FIR filter design to verify that the result remains the same after synthesizing process. Also, we provide a performance comparison between the three HLS languages to see which HLS tool or HLS language can operate faster, with fewer resources, or which one consumes more power.

In the first experiment, all Vivado HLS designs are executed in "push-button" manner with all its standard and default settings. Table 5.2 shows the results of this experiment. It should be noted that power consumption is reported in watt and absolute latency is reported in ns.

<table>
<thead>
<tr>
<th></th>
<th>Clash</th>
<th>Vivado HLS (C++)</th>
<th>Vivado HLS (SystemC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DSP4SE</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FF</td>
<td>640</td>
<td>112</td>
<td>119</td>
</tr>
<tr>
<td>LUT</td>
<td>1235</td>
<td>129</td>
<td>200</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.574</td>
<td>0.123</td>
<td>0.123</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>23.370</td>
<td>0.002</td>
<td>0.003</td>
</tr>
<tr>
<td>Latency(cycles)</td>
<td>10</td>
<td>617</td>
<td>248</td>
</tr>
<tr>
<td>Latency(absolute)</td>
<td>100</td>
<td>6170</td>
<td>2480</td>
</tr>
</tbody>
</table>

Table 5.2: Standard Optimization Result (FIR Filter)

According to intrinsic features of Clash and functional languages, we know that each function and operation in Clash can be performed in parallel by default, so
there is no need to add any optimization in the design. However, in Vivado HLS, to get better performance in speed and to have a circuit that executes in parallel, we have to use optimization directives. Table 5.3 lists the same performance metrics obtained in the second scenario where we used optimizations.

<table>
<thead>
<tr>
<th></th>
<th>Clash</th>
<th>Vivado HLS (C++)</th>
<th>Vivado HLS (SystemC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DSP48E</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FF</td>
<td>640</td>
<td>522</td>
<td>633</td>
</tr>
<tr>
<td>LUT</td>
<td>1235</td>
<td>686</td>
<td>894</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.574</td>
<td>0.123</td>
<td>0.123</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>23.370</td>
<td>0.013</td>
<td>0.004</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>10</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>Latency (absolute)</td>
<td>100</td>
<td>140</td>
<td>270</td>
</tr>
</tbody>
</table>

Table 5.3: Performance-Optimized Results (FIR Filter)

5.2 Eigenvalue Decomposition Evaluation

Similar to FIR filter evaluation, first we examine the correctness and validation of our result. For this purpose, we employed "MATLAB" to find the eigenvalues and eigenvectors of the sample input signal used in this experiment. The arbitrary input signal is as follow:

\[
\text{Input} = \{1, 2, 3, 4\}
\]

We are going to find the covariance matrix, eigenvalues and eigenvectors of this input signal.

Since the principle of the covariance matrix is the multiplication of a vector and its transpose, the result of covariance matrix calculation can be verified in MATLAB with the code snippet shown below: By running this code on MATLAB command window, we will get the 4 × 4 symmetric matrix illustrated below which is our desired covariance matrix.

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 4 & 6 & 8 \\
3 & 6 & 9 & 12 \\
4 & 8 & 12 & 16
\end{bmatrix}
\]

In order to get eigenvalues and eigenvectors of this obtained matrix, we can simply run the following code on MATLAB command window.

\[
[evecs, evals] = eig (output)
\]

The resultant eigenvalues and eigenvectors will be displayed as below:
As explained before, we can simulate our design in Clash before compiling it. So, same as previous sub-section, we can simulate our EVD design in Clash to make sure that result are perfectly match with MATLAB result. Figure 5.1 shows the simulation result of one clock cycle. It consists of three components, a list of eigenvalues, a list that includes sub-lists and each sub-list denotes eigenvectors corresponding to the eigenvalue in the same place and another output that can get a value of 0 or 1. In this experiment, we got the result after 3 clock cycles. If we have complex input signals, it could take more than 3 clock cycles to generate the final result. Also, it should be noted that the eigenvectors are initialized with an identity matrix multiplied by 1000 as mentioned earlier, that is the reason why all eigenvectors scaled by 1000 in comparison with the result of MATLAB simulation. The result of Clash simulation is almost matched with the result of MATLAB. However, there is a little proportional error which is caused by fixed-point operations such as the bitwise shift in the entire program.

By starting C simulation process on Vivado HLS, we can get the result of this experiment in C++ and SystemC implementation. It was validated that there is a good concordance between the result of MATLAB, Clash and Vivado HLS which concludes we are on the right track.

### 5.2.1 Synthesis Result of EVD

The data type for input and output ports of EVD design implemented by arbitrary precision data type with "Int" and 16 bit-width. Same as what we have done for the FIR filter, we execute two scenarios with standard optimizations and manually assigned optimizations respectively. Table 5.4 shows the synthesis result of the first experiment with default configurations. Again the consumed power is reported in watt and absolute latency is reported in ns.

For the second experiment, we used three different optimizations (pragma) in Vivado HLS implementation as described below:
Table 5.4: Standard Optimization Result (EVD)

<table>
<thead>
<tr>
<th></th>
<th>Clash</th>
<th>Vivado HLS (C++)</th>
<th>Vivado HLS (SystemC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>DSP48E</td>
<td>12</td>
<td>11</td>
<td>38</td>
</tr>
<tr>
<td>FF</td>
<td>388</td>
<td>2185</td>
<td>5795</td>
</tr>
<tr>
<td>LUT</td>
<td>7519</td>
<td>3122</td>
<td>8624</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.157</td>
<td>0.123</td>
<td>NA</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>4.999</td>
<td>0.017</td>
<td>NA</td>
</tr>
<tr>
<td>Latency(cycles)</td>
<td>10</td>
<td>1860</td>
<td>8130</td>
</tr>
<tr>
<td>Latency(absolute)</td>
<td>100</td>
<td>18600</td>
<td>81300</td>
</tr>
</tbody>
</table>

- **HLS PIPELINE**: Using this optimization allows operation to execute concurrently. It can be applied to functions or loops throughout the entire design.

- **HLS ARRAY PARTITION**: In Vivado HLS, arrays are implemented as block RAM which only has a maximum of two data ports which causes limitations on throughput. By using this optimization, we can enhance bandwidth by splitting the array into multiple smaller arrays.

- **HLS DEPENDENCE**: When all operations are performed in sequential form, there are no dependencies to take into account. When we apply parallelism in our design, here where we need to deal with some dependencies such as memory or data dependencies when a read or a write occurs after a previous read or write.

Table 5.5 shows the result obtained in the second scenario.

Table 5.5: Performance-optimized Result (EVD)

<table>
<thead>
<tr>
<th></th>
<th>Clash</th>
<th>Vivado HLS (C++)</th>
<th>Vivado HLS (SystemC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>DSP48E</td>
<td>12</td>
<td>28</td>
<td>41</td>
</tr>
<tr>
<td>FF</td>
<td>388</td>
<td>24455</td>
<td>6179</td>
</tr>
<tr>
<td>LUT</td>
<td>7519</td>
<td>28020</td>
<td>13876</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.157</td>
<td>0.124</td>
<td>NA</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>4.999</td>
<td>0.170</td>
<td>NA</td>
</tr>
<tr>
<td>Latency(cycles)</td>
<td>10</td>
<td>135</td>
<td>824</td>
</tr>
<tr>
<td>Latency(absolute)</td>
<td>100</td>
<td>1350</td>
<td>8240</td>
</tr>
</tbody>
</table>

Table 5.5: Performance-optimized Result (EVD)
Chapter 6

Discussion

In this chapter, we explain the results of the HLS tools and highlight some techniques used in each experiment.

The synthesis result of the two proposed benchmarks was provided in the previous chapter. In this thesis, we have provided the result of the entire EVD calculation in comparison with [22] where only the synthesis result of a sole CORDIC algorithm was provided briefly in case of utilization, maximum frequency and the number of registers, so, a better analysis of Clash can be found in our work.

As shown in the synthesis result of FIR filter and EVD, the latency of the generated design by Clash has better performance in all cases than that of the design generated by Vivado HLS due to the intrinsic parallelism feature of Clash. The desired latency in Clash depends on the number of iterations defined in the testbench. The implemented FIR filter has 11 taps as mentioned in the previous chapter for inputs and coefficients and each iteration is performed in one clock cycle in Clash, so, it takes 11 cycles to produce the output. Furthermore, we executed 10 iterations in EVD calculation to get the highest accuracy regarding CORDIC computation. As a result, we got 100\text{ns} for Clash latency in all four designs.

On the other hand, the result of latency has a direct relation with the level of optimization used in the design. As illustrated in table 5.3 and 5.5, the latency of C++ and SystemC design is drastically lessened after optimization applied to each design. For instance, if we apply parallelism to the outer loop of a nested loop, we will see more reduction in latency. Moreover, Vivado HLS exploits the "Latency" directive in which we can specify the latency constraint for more optimizations.

It can be seen from table 5.4 and 5.5 that SystemC power consumption result was not reported. One of the limitations of Vivado HLS for SystemC design is that there is no support for some HLS math libraries of fixed-point functions that can be used for C++ implementation. The calculation of tangent values in EVD algorithm forced us to use floating-point instead of fixed-point data types. As there is no support for fixed-point math function on SystemC, we had to use floating-point data type which causes some issues in SystemC synthesizing process such as the generated circuit would be unreasonably slow as shown in the latency result. Furthermore, it prevents Vivado HLS to perform C/RTL simulation, hence the RTL testbench that can be used to get the required "saif" file for measuring the power consumption on Xilinx FPGA EDA tool cannot be generated. However, in Clash a template Haskell construct has been used which can do term-level computation and then use the result as a part of the syntax tree of the main program to be compiled.
So in Clash, we compute, at compile-time, the constant vector with the constants generated by the "LUT" module. This also can significantly reduce the number of area resources.

Clash power consumption is higher than Vivado HLS in all scenarios where the biggest portion is related to dynamic power. The static power consumption is driven by process technology and silicon design and the dynamic power consumption is driven by design’s utilization. So, dynamic power depends on the specific use of each resource. As Clash generally is not optimized in generated Input-Output Block (IOB) and it produces more IOBs in comparison with Vivado HLS. The use of general purpose I/O impacts the total power requirements and the energy required for an I/O operation is substantially greater than for internal operation which makes I/O be very energy-intensive, thus the huge results for Clash dynamic power. Also, it should be noted the entire tool-chain on Vivado HLS that generates all data is maintained by Xilinx which is compatible with its devices, so, the RTL produced by Vivado HLS has better power consumption results. As an example, based on the simulation report generated by Vivado IDE, the number of bonded IOBs generated by Clash in the EVD algorithm exceeds the total number of available IOBs in "xc7z030-sbv485-3" device.

In case of the resource utilization results, it is clear that when optimizations were used in both experiments, the number of resources in both Vivado HLS designs increased. It is because by applying pipeline directives, more resources will be occupied to perform parallel operations. Using optimization depends on the purpose of our design. If speed is the priority, we should use more optimizations in different levels to decrease latency. On the other hand, if speed is not important and we need a design with fewer resources, it is better to synthesis the design with its default and standard settings.

It should be noted that SystemC implementation is no longer supported by Vivado HLS and in the new version of Vivado HLS namely "Vitis", it has been deprecated, so most of these optimizations are not suitable and we cannot assign them properly in SystemC design. Therefore, C++ implementation has better performance and results as opposed to SystemC because Vivado HLS provides more options for C/C++ design.
Chapter 7

Conclusions

In this thesis, we provided an evaluation of three HLS languages and two HLS tools that are compatible with these languages. We first represented an extensive survey of some general-purpose HLS tools, compilers and DSLs that can be used for HLS. Then, we evaluated Vivado HLS, a commercial HLS tool that supports C/C++ and SystemC, and Clash, an HLS tool developed in the University of Twente for the Haskell language, with two benchmarks, FIR filter and Eigenvalue Decomposition (EVD). Then, we presented a comprehensive comparison based on different metrics (latency, resource utilization and power consumption), which has not been done in previous studies.

Based on the acquired results, Clash has faster performance and better latency in all designs because it offers parallelism within the implementation which is one of its main features that proves a design flow in a functional language is more like digital hardware. However, the amount of power consumed in generated RTLs from Clash in all experiments is far higher than that of the RTLs generated by Vivado HLS due to the lack of support for interface optimization in Clash. The resource utilization results depicted that the generated RTL from Vivado HLS uses less area when there is no optimization applied to the design. However, when speed is a prominent factor in designing a circuit and we have to use optimization in our design, the number of resources increased significantly.

The obtained results showed that no single tool generates the best results for all performance metrics discussed in this thesis work. Vivado HLS provides more features for optimization depends on what we need for the desired design. Furthermore, as Vivado HLS has been developed by Xilinx FPGA vendor and specifically concentrates on Xilinx FPGAs, it is more compatible with these devices and the generated RTL can work better on Xilinx FPGAs. On the other hand, Clash only exploits some functional language features such as intrinsic parallelism which impacts the final hardware circuit to work fully pipeline. The generated Verilog or VHDL file by Clash can be used to produce bitstream on other FPGA devices as well, such as Intel FPGAs. Also, the experience of developing with these tools showed that writing a function that executes a set of instructions in which required about 10 lines in C++, can be done in Clash within just one line. However, using C++ or other HLS languages based on C-like such as SystemC would be easier to adopt by developers than an unfamiliar language like Clash. So in overall, software designers who interested in utilizing FPGAs solutions must be aware of which metrics are critical for the application at hand, and choose the HLS tool/language accordingly.
7.1 Future Work

In this section, we present some aspects of this research that still need to be taken into account for further research work and experiments.

Although Vivado HLS provides support for debugging the generated RTL utilizing C/RTL simulation, some academic HLS tools such as Clash do not have this ability. Since debugging HLS at the RTL level is complicated because the original pre-synthesis code may not resemble the RTL architecture generated by the HLS tool. So, a future work could be adding debugging options to these HLS tools.

In this thesis, only Xilinx FPGA device was used to synthesis the produced HDL code. Future studies should consider whether the FPGA model matters and impacts the result of the synthesis process. It was not investigated that by using power optimization feature which exists in Vivado IDE in which extent the consumed power can be reduced. As to future work, one may investigate the advantages of enabling power optimization mode especially for the RTL generated by Clash which consumed the highest power.

It is worthwhile to be noted that the match between structure in Clash source and the structure in generated hardware can also be a very good thing as it allows control over the entire structure. However, the reason that makes Clash not be precisely considered as a High Level Synthesis is that we can construct a circuit by unrolling it over time in HLS, but Clash only unrolls over space. It would be interesting to conduct some research and evolution for Clash to do High Level Synthesis.
Bibliography


Appendix A

C++ Code for EVD

```cpp
// CMC Module in C++ //
#include "ap_int.h"
#include <iostream>
#include <cmath>
using namespace std;

void cmc(ap_int<16> in_vec[1][4], ap_int<16> inp[10])
{
    // int input[4][4];
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            in_vec_trans[i][j] = in_vec[i][j];
        }
    }

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            out_vec[i][j] = 0;
        }
    }
    for (int j = 0; j < 4; j++)
    {
        for (int k = 0; k < 1; k++)
        {
            if (i <= j)
                out_vec[i][j] += in_vec_trans[i][k] * in_vec[k][j];
        }
    }
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            out_vec[i][j] += in_vec_trans[i][k] * in_vec[k][j];
        }
    }
    for (int i = 0; i < 4; i++)
    {
        // code continues...
    }
    // HLS PIPELINE
    #pragma HLS ARRAY_PARTITION variable = in_vec complete dim = 0
    #pragma HLS PIPELINE
    ap_int<16> out_vec[4][4];
    ap_int<16> in_vec_trans[4][1];
    int n = 0;
}```
inp[i] = out_vec[0][i];
}
for (int i = 0; i < 3; i++)
{
    inp[i+4] = out_vec[1][i+i];
}
for (int i = 0; i < 2; i++)
{
    inp[i+7] = out_vec[2][2+i];
}
for (int i = 0; i < 1; i++)
{
    inp[i+9] = out_vec[3][3+i];
}
}

// CordicA Module in C++ //
#include "ap_int.h"
#include <iostream>
#include <cmath>
using namespace std;

void cordic_a(ap_int<16> ri, ap_int<16> ui, ap_int<16> ds[10],
               ap_int<16> *r, ap_int<16> *u)
{
    #pragma HLS PIPELINE
    ap_int<16> p1,p2,q1,q2,q3,q4;
    for (int i = 0; i < 10; i++)
    {
        if (ri < 0){
            p1 = ri >> ((i)*2);
            q3 = (ri >> i);
        }
        else if(i==1)
        {
            p1 = ri >> ((i)*2);
            q3 = (ri >> i) + 1;
        }
        else if((i>15) || ((i)*2)>15)
        {
            p1 = 0;
            q3 = 0;
        }
        else{
            p1 = (ri >> ((i)*2)) + 1;
            q3 = (ri >> i) + 1;
        }
        q1 = ri - p1;
        if(ui < 0){
            q2 = (ui >> i);
            p2 = ui >> ((i)*2);
        }
        else if(i==1)
        {
            q2 = (ui >> i) + 1;
            p2 = ui >> ((i)*2);
        }
        else if((i>15) || ((i)*2)>15)
{  
   q2 = 0;
   p2 = 0;
}
else{
   q2 = (ui >> i) + 1;
   p2 = (ui >> ((i)*2)) + 1;
}
q4 = ui - p2;
ds[i] = ((ri<0 && ui>0) || (ri>0 && ui<0)) ? 1 : 0;

if(ds[i] == 0){
   ri = q1 - (q2 << 1);
   *r = q1 - (q2 << 1);
} else{
   ri = q1 + (q2 << 1);
   *r = q1 + (q2 << 1);
}

if(ds[i] == 0){
   ui = q4 + (q3 << 1);
   *u = q4 + (q3 << 1);
} else{
   ui = q4 - (q3 << 1);
   *u = q4 - (q3 << 1);
}

}

// CordicB Module in C++ //
#include "ap_int.h"
#include <iostream>
#include <cmath>
using namespace std;

void cordic_b(ap_int<16> x, ap_int<16> y, ap_int<16> d[10], ap_int<16> *x1, ap_int<16> *y1)
{
   #pragma HLS PIPELINE
   ap_int<16> z1, z2;
   for(int i = 0; i < 10; i++)
   {
      z1 = y >> i;
      z2 = x >> i;
      if(d[i] == 0)
         x = x - z1;
      else
         x = x + z1;
      if(d[i] == 0)
         y = y + z2;
      else
         y = y - z2;
      *x1 = (x >> 1)+(x >> 3) - ((x >> 6)+(x >> 9)+(x >> 13));
      *y1 = (y >> 1)+(y >> 3) - ((y >> 6)+(y >> 9)+(y >> 13));
   }
}
// Update Module in C++ //
#include "ap_int.h"
#include "ap_fixed.h"
#include <iostream>
#include "hls_math.h"

using namespace std;
#define pi 3.141592653589793

void update(ap_int<16> di[10], ap_fixed<16,14> * tmp)
{
#pragma HLS PIPELINE
ap_fixed<16,14> a1[10] = {pi/180*45, pi/180*26.6, pi/180*14, pi/180*7.1, pi/180*3.6, pi/180*1.8, pi/180*0.9, pi/180*0.4, pi/180*0.2, pi/180*0.1};
ap_fixed<16,14> theta=0;
for(int i = 0; i < 10; i++)
{
  if(di[i] == 1)
  {
    theta += 1*a1[i];
  }
  else
  {
    theta += -1*a1[i];
  }
}
tmp = (1024*hls::tan(theta));
if(*tmp < 0)
  *tmp = -1*(*tmp);
else
  *tmp = -1*(*tmp);
}

void evd(ap_int<16> in[1][4], ap_int<16> eval[4], ap_int<16> evec[4][4])
{
  ap_int<16> evs[4][4] =
  {{1000,0,0,0},{0,1000,0,0},{0,0,1000,0},{0,0,0,1000}};
ap_int<16> evsinit[4][4];
#pragma HLS DEPENDENCE variable=evs array inter false
#pragma HLS DEPENDENCE variable=evsinit array inter false
#pragma HLS ARRAY_PARTITION variable=evs complete dim=0
ap_int<16> out[10];
#pragma HLS DEPENDENCE variable=out array inter false
ap_int<16> output[10];
cmc(in, out);
for(int e = 0; e < 10; e++)
{
  #pragma HLS PIPELINE
}
for (int n = 0; n < 10; n++)
{
    output[n] = out[n];
}

for (int m = 0; m < 4; m++)
{
    for (int l = 0; l < 4; l++)
    {
        evsinit[m][l] = evs[m][l];
    }
}

if (out[1]!=0 || out[2]!=0 || out[3]!=0 || out[5]!=0 || out[6]!=0 || out[8]!=0)
{
    ap_int<16> r1, u1, d1[10], r2, u2, d2[10], R1, U1, R2, U2;
    r1 = 2*output[1];
    u1 = output[4] - output[0];
    r2 = 2*output[8];
    u2 = output[9] - output[7];
    cordic_a(r1, u1, d1, &R1, &U1);
    cordic_a(r2, u2, d2, &R2, &U2);
    ap_int<16> ds1[10], ds2[10];
    for (int k = 0; k < 10; k++)
    {
        ds1[k] = d1[k];
        ds2[k] = d2[k];
    }
    ap_fixed<16,14> tanv1, tanv2;
    update(ds1, &tanv1);
    update(ds2, &tanv2);

    for (int i = 0; i < 1; i++)
    {
        out[i] = output[0] - ((output[1]*tanv1)>>10);
        out[i+9] = output[4] + ((output[1]*tanv1)>>10);
        out[i+4] = output[7] - ((output[8]*tanv2)>>10);
        out[i+7] = output[9] + ((output[8]*tanv2)>>10);
    }
    out[3] = 0; out[5] = 0;
    cordic_b(output[2], output[5], ds1, &e13_tmp, &e23_tmp);
    cordic_b(output[3], output[6], ds1, &e14_tmp, &e24_tmp);
    cordic_b(e13_tmp, e14_tmp, ds2, &out[1], &out[2]);
    cordic_b(e23_tmp, e24_tmp, ds2, &out[6], &out[8]);
    cordic_b(evsinit[0][0], evsinit[0][1], ds1, &evs[0][0], &evs[0][1]);
    cordic_b(evsinit[1][0], evsinit[1][1], ds1, &evs[1][0], &evs[1][1]);
    cordic_b(evsinit[2][0], evsinit[2][1], ds1, &evs[2][0], &evs[2][1]);
    cordic_b(evsinit[3][0], evsinit[3][1], ds1, &evs[3][0], &evs[3][1]);
    cordic_b(evsinit[0][2], evsinit[0][3], ds2, &evs[0][2], &evs[0][3]);
    cordic_b(evsinit[1][2], evsinit[1][3], ds2, &evs[1][2], &evs[1][3]);
# evd Function Header File in C++ //
#include "ap_int.h"
#include <iostream>
#include <cmath>
using namespace std;

void evd(ap_int<16>[1][4], ap_int<16> eval[4], ap_int<16> evec[4][4]);

// Testbench for EVD in C++ //
#include "evd.h"
#include <stdio.h>

int main()
{
    ap_int<16> in [1][4] = {1,2,3,4};
ap_int<16> eval[4];
ap_int<16> evec[4][4];
evd(in, eval, evec);
    for(int i = 0; i < 4; i++)
    {
        cout << "eval[" << i << "]: " << eval[i] << endl;
    }
    for(int i = 0; i < 4; i++)
    {
        for(int j = 0; j < 4; j++)
        {
            evec[i][j] = eval[j][i];
        }
    }
for (int j = 0; j < 4; j++)
{
    cout << evec[" << i << "] " << "[" << j << "]: " << evec[i][j] << endl;
}
}
Appendix B
SystemC Code for EVD

// Top Function Header (EVD.h) in SystemC //
#ifndef EVD_H
#define EVD_H

#include <systemc.h>
#include "ap_mem_if.h"

SC_MODULE(EVD)
{
    sc_in<bool> reset;
    sc_in<bool> clock;
    sc_in<bool> start;
    sc_fifo_in<sc_int<16> > in_vec;
    sc_fifo_out<sc_int<16> > eval;
    sc_fifo_out<sc_int<16> > evec;
    bool write_done;

    void Prc1();

    SC_CTOR(EVD)
    : in_vec("in_vec"), eval("eval"), evec("evec")
    {
        SC_CTHREAD(Prc1, clock.pos());
        reset_signal_is(reset, true);
    }
};
#endif

// Top Level Module (EVD) in SystemC
#include "EVD.h"
#include "cordic_a.h"
#include "cordic_b.h"
#include "update.h"
#include <systemc.h>

sc_int<16> evs[4][4] =
    {{1000,0,0,0},{0,1000,0,0},{0,0,1000,0},{0,0,0,1000}};
sc_int<16> evsinit[4][4];
sc_int<16> out[10];
sc_int<16> output[10];
void EVD::Prc1()
{
    #pragma HLS DEPENDENCE variable=evs array inter false
    sc_int<16> out1[4];
    sc_int<16> out2[16];
    sc_int<16> input[1][4];
    sc_int<16> out_vec[4][4];
    #pragma HLS ARRAY_PARTITION variable=out_vec cyclic factor=2 dim=0
    sc_int<16> in_vec_trans[4][1];
    write_done = false;
    wait();

    while(true)
    {
        while (!start.read()) wait();
        write_done = false;
        for(int j = 0; j < 4; j++)
        {
            input[0][j] = in_vec.read();
            #ifndef __SYNTHESIS__
            cout << "input: " << input[0][j] << endl;
            #endif
        }

        for(int i = 0; i < 4; i++)
        {
            #pragma HLS PIPELINE
            for(int j = 0; j < 1; j++)
            {
                in_vec_trans[i][j] = input[j][i];
            }
        }

        for(int i = 0; i < 4; i++)
        {
            #pragma HLS PIPELINE
            for(int j = 0; j < 4; j++)
            {
                out_vec[i][j] = 0;
            }
        }

        for(int i = 0; i < 4; i++)
        {
            #pragma HLS PIPELINE
            for(int j = 0; j < 4; j++)
            {
                for(int k = 0; k < 1; k++)
                {
                    if(i==j)
                        out_vec[i][j] += in_vec_trans[i][k] * input[k][j];
                }
            }
        }

        for(int i = 0; i < 4; i++)
        {
            out[i] = out_vec[0][i];
        }
    }
for(int i = 0; i < 3; i++)
{
    out[i+4] = out_vec[1][1+i];
}
for(int i = 0; i < 2; i++)
{
    out[i+7] = out_vec[2][2+i];
}
for(int i = 0; i < 1; i++)
{
    out[i+9] = out_vec[3][3+i];
}
for(int e = 0; e < 10; e++)
{

#pragma HLS PIPELINE

for(int n = 0; n < 10; n++)
{
    output[n] = out[n];
}
for(int m = 0; m < 4; m++)
{
    for(int l = 0; l < 4; l++)
    {
        evsinit[m][l] = evs[m][l];
    }
}

if(out[1]!=0 || out[2]!=0 || out[3]!=0 || out[5]!=0 || out[6]!=0 || out[8]!=0)
{
    sc_int<16> r1, u1, d1[10], r2, u2, d2[10], R1, U1, R2, U2;
    r1 = 2*output[1];
    u1 = output[4] - output[0];
    r2 = 2*output[8];
    u2 = output[9] - output[7];
    cordic_a(r1, u1, d1, &R1, &U1);
    cordic_a(r2, u2, d2, &R2, &U2);
    sc_int<16> ds1[10], ds2[10];
    for(int k = 0; k < 10; k++)
    {
        ds1[k] = d1[k];
        ds2[k] = d2[k];
    }
    sc_int<16> tanv1, tanv2;
    update(ds1, &tanv1);
    update(ds2, &tanv2);
    for(int i = 0; i < 1; i++)
    {
        out[i] = output[0] - ((output[1]*tanv1)>>10);
        out[i+9] = output[4] + ((output[1]*tanv1)>>10);
        out[i+4] = output[7] - ((output[8]*tanv2)>>10);
        out[i+7] = output[9] + ((output[8]*tanv2)>>10);
    }
    out[3] = 0; out[5] = 0;
    cordic_b(output[2], output[5], ds1, &e13_tmp, &e23_tmp);
    cordic_b(output[3], output[6], ds1, &e14_tmp, &e24_tmp);
```c
127  cordic_b(e13_tmp, e14_tmp, ds2, &out[1], &out[2]);
128  cordic_b(e23_tmp, e24_tmp, ds2, &out[6], &out[8]);
129
130  cordic_b(evsinit[0][0], evsinit[0][1], ds1, &evs[0][0], &evs[0][1]);
131  cordic_b(evsinit[1][0], evsinit[1][1], ds1, &evs[1][0], &evs[1][1]);
132  cordic_b(evsinit[2][0], evsinit[2][1], ds1, &evs[2][0], &evs[2][1]);
133  cordic_b(evsinit[3][0], evsinit[3][1], ds1, &evs[3][0], &evs[3][1]);
134
135  cordic_b(evsinit[0][2], evsinit[0][3], ds2, &evs[0][2], &evs[0][3]);
136  cordic_b(evsinit[1][2], evsinit[1][3], ds2, &evs[1][2], &evs[1][3]);
137  cordic_b(evsinit[2][2], evsinit[2][3], ds2, &evs[2][2], &evs[2][3]);
138  cordic_b(evsinit[3][2], evsinit[3][3], ds2, &evs[3][2], &evs[3][3]);
139
140  for (int i = 0; i < 4; i++)
141  {
142      int t = evs[i][1];
143      evs[i][1] = evs[i][2];
144      evs[i][2] = evs[i][3];
145      evs[i][3] = t;
146  }
147
148  for (int j = 0; j < 1; j++)
149  {
150      out1[j] = out[j];
151      out1[j+1] = out[j+4];
152      out1[j+2] = out[j+7];
153      out1[j+3] = out[j+9];
154  }
155
156  for (int i = 0; i < 1; i++)
157  {
158      for (int j = 0; j < 4; j++)
159      {
160          out2[j] = evs[j][i];
161      }
162  }
163
164  for (int i = 1; i < 2; i++)
165  {
166      for (int j = 0; j < 4; j++)
167      {
168          out2[j+4] = evs[j][i];
169      }
170  }
171
172  for (int i = 2; i < 3; i++)
173  {
174      for (int j = 0; j < 4; j++)
175      {
176          out2[j+8] = evs[j][i];
177      }
178  }
for (int i = 3; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        out2[j+12] = evs[j][i];
    }
}

for (int i = 0; i < 4; i++)
{
    eval.write(out1[i]);
}
for (int i = 0; i < 16; i++)
{
    evec.write(out2[i]);
}
write_done = true;
wait();
}

// CoridcA Module in SystemC //
#include <iostream>
#include <systemc.h>
#include <cmath>
using namespace std;

void cordic_a(sc_int<16> ri, sc_int<16> ui, sc_int<16> ds[10],
              sc_int<16> *r, sc_int<16> *u)
{
#pragma HLS PIPELINE
    sc_int<16> p1, p2, q1, q2, q3, q4;
    for (int i = 0; i < 10; i++)
    {
        if (ri < 0){
            p1 = ri >> ((i)*2);
            q3 = (ri >> i);
        } else if (i==1){
            p1 = ri >> ((i)*2);
            q3 = (ri >> i) + 1;
        } else if ((i>15) || ((i)*2)>15){
            p1 = 0;
            q3 = 0;
        } else{
            p1 = (ri >> ((i)*2)) + 1;
            q3 = (ri >> i) + 1;
        }
        q1 = ri - p1;
        if (ui < 0){
            q2 = (ui >> i);
            p2 = ui >> ((i)*2);
```c
}  
else if (i == 1)
{
    q2 = (ui >> i) + 1;
    p2 = ui >> ((i) * 2);
}  
else if ((i > 1) || ((i) * 2) > 15)
{
    q2 = 0;
    p2 = 0;
}
else {
    q2 = (ui >> i) + 1;
    p2 = (ui >> ((i) * 2)) + 1;
}
q4 = ui - p2;
ds[i] = ((ri < 0 && ui > 0) || (ri > 0 && ui < 0)) ? 1 : 0;
```

```c
if (ds[i] == 0){
    ri = q1 - (q2 << 1);
    *r = q1 - (q2 << 1);
}
else{
    ri = q1 + (q2 << 1);
    *r = q1 + (q2 << 1);
}
if (ds[i] == 0){
    ui = q4 + (q3 << 1);
    *u = q4 + (q3 << 1);
}
else{
    ui = q4 - (q3 << 1);
    *u = q4 - (q3 << 1);
}
```
if (d[i] == 0)
    y = y + z2;
else
    y = y - z2;
*x1 = (x >> 1) + (x >> 3) - ((x >> 6) + (x >> 9) + (x >> 13));
*y1 = (y >> 1) + (y >> 3) - ((y >> 6) + (y >> 9) + (y >> 13));
}

// Update Module in SystemC //
#include "ap_fixed.h"
#include <iostream>
#include <cmath>
#include <systemc.h>
using namespace std;
#define pi 3.141592653589793
void update(sc_int<16> di[10], sc_int<16> *tmp)
{
    #pragma HLS PIPELINE
    float a1[10] = {pi/180*45, pi/180*26.6, pi/180*14, pi/180*7.1, pi/180*3.6,
            pi/180*1.8, pi/180*0.9, pi/180*0.4, pi/180*0.2, pi/180*0.1};
    float theta = 0;
    for(int i = 0; i < 10; i++)
    {
        if (di[i] == 1)
        {
            theta += 1*a1[i];
        }
        else
        {
            theta += -1*a1[i];
        }
    }
    *tmp = (1024*tan(theta));
    if(*tmp < 0)
    *tmp = -1*(tmp);
else
    *tmp = -1*(tmp);
}

// Testbench Header in SystemC //
#ifndef EVD_TB_H
#define EVD_TB_H
#include <systemc.h>

SC_MODULE(EVD_tb)
{
    sc_in_clk clock;
    sc_in <bool> reset;
    sc_in <bool> start;
    sc_fifo_out<sc_int<16>> in_vec;
    sc_fifo_in<sc_int<16>> eval;
    sc_fifo_in<sc_int<16>> evec;
    void source();
    void sink();
    SCCTOR(EVD_tb)
}
#include "EVD_tb.h"
#include <iostream>
#include <stdio.h>
using namespace std;

void EVD_tb::source()
{
    while (!start.read()) wait();
    for(int i = 1; i < 5; i++)
    {
        in_vec.write(i);
    }
    wait();
}

void EVD_tb::sink()
{
    sc_int<16> eigenval[4];
    sc_int<16> eigenvec[16];
    while (!start.read()) wait();
    wait();
    for(int i = 0; i < 4; i++)
    {
        eigenval[i] = eval.read();
        cout << "eval[" << i << "]:" << eigenval[i] << endl;
    }
    for(int i = 0; i < 16; i++)
    {
        eigenvec[i] = evec.read();
        cout << "evec[" << i << "]:" << eigenvec[i] << endl;
    }
    wait();
}

int sc_main (int argc , char *argv[])
{

}
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
    SC_D0 NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);

sc_signal<bool> s_reset;
sc_signal<bool> s_start;
sc_signal<bool> s_done;
sc_fifo<sc_int<16>> in_vec("in_fifo",4);
sc_fifo<sc_int<16>> eval("eval_fifo",4);
sc_fifo<sc_int<16>> evec("evec_fifo",16);
sc_clock s_clock("s_clock",10,SC_NS);

tb_init U_tb_init("U_tb_init");
EVD U_dut("U_dut");
EVD_tb U_tb_driver("U_tb_driver");

U_tb_init.clock(s_clock);
U_tb_init.reset(s_reset);
U_tb_init.start(s_start);
U_tb_init.done(s_done);
U_dut.clock(s_clock);
U_dut.reset(s_reset);
U_dut.start(s_start);
U_dut.in_vec(in_vec);
U_dut.eval(eval);
U_dut.evec(evec);
U_tb_driver.clock(s_clock);
U_tb_driver.reset(s_reset);
U_tb_driver.start(s_start);
U_tb_driver.in_vec(in_vec);
U_tb_driver.eval(eval);
U_tb_driver.evec(evec);

int end_time = 10000;
sc_start(end_time,SC_NS);
return 0;

// Testbench Initialization //
#ifndef TB_INIT_H
#define TB_INIT_H

#include <systemc.h>
#include <iostream>
#include "string"
using namespace std;

SC_MODULE(tb_init)
{
    sc_in<bool> clock;
    sc_in<bool> done;
    sc_out<bool> reset;
    sc_out<bool> start;
}
void prc_reset();

SC_CTOR(tb_init)
{
    SC_CTHREAD(prc_reset,clock.pos());
}

};
#endif

// Testbench initialization function //
#include "tb_init.h"

void tb_init::prc_reset()
{
    reset = true;
    start = false;
    wait(10,SC_NS);

    reset = false;
    wait(10,SC_NS);
    start = true;
    wait(40,SC_NS);
    while (!done.read()) wait();
    start = false;
};


65
Appendix C

Clash Code for EVD

```
-- CMC Module in Clash --
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module CMC where
import Clash.Prelude
import Clash.Sized.Vector
import Clash.Explicit.Testbench

type Matrix1 = Vec 1 (Vec 4 (Signed 16))
type Matrix2 = Vec 4 (Vec 4 (Signed 16))
type CMCR = ((Signed 16, Signed 16, Signed 16, Signed 16),
            (Signed 16, Signed 16, Signed 16),
            (Signed 16, Signed 16),
            Signed 16)

dot vec1 vec2 = sum (zipWith (*) vec1 vec2)
mvMult mat vec = map (dot vec) mat

mmMult :: Matrix1 -> Matrix2
mmMult mat1 = map (mvMult (transpose mat1)) (transpose mat1)

cmcCore :: Matrix1 -> CMCR
cmcCore ys = inp
  where
    mat = concat $ mmMult ys
    inp = uptriangle mat

uptriangle :: Vec 16 (Signed 16) -> CMCR
uptriangle rs = inp
  where
    inp = ((rs!!0, rs!!1, rs!!2, rs!!3), (rs!!5, rs!!6, rs!!7), (rs!!10, rs!!11), rs!!15)

-- CordicA Module in Clash --
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module CordicA where
import Clash.Prelude
import Resize

type CordicI = (Signed 16, Signed 16)
```
type CordicO = Vec 10 Bit

cordic_a :: Signed 16 -> Signed 16 -> CordicO

cordic_a r u = ds 
   where 
      ids = $(listToVecTH ([0..9]:[Int])) 
      ((r',u''),ds) = mapAccumL ca (r,u) ids 

   ca (ri,ui) i = ((ri',ui'),di) 
   where 
      p1 = myshiftR ri (i*2) 
      q1 = ri - p1 
      q2 = myshiftR ui i 
      q3 = myshiftR ri i 
      q4 = ui - q2 
      di = getSign ri ui 
      ri' = addSub di q1 (shiftL q2 1) 
      ui' = addSub (complement (di) + 2) q4 (shiftL q3 1) 

addSub 1 a b = a + b 
addSub 0 a b = a - b 

getSign x y = if head (bv2v (pack x)) == head (bv2v (pack y)) then 0 
else 1 

-- CordicB Module in Clash -- 
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}

module CordicB where 
import Clash.Prelude 

type CordicI1 = (Signed 16, Signed 16) 
type CordicI2 = Vec 10 Bit 
type CordicO = (Signed 16, Signed 16) 

cordic_b :: CordicI1 -> CordicI2 -> CordicO 
cordic_b (x, y) ds = (x', y') 
   where 
      ids = $(listToVecTH([1..10]:[Int])) 
      (xtmp, ytmp) = foldl cb (x, y) (zip ids ds) 
      x' = scale2 xtmp 
      y' = scale2 ytmp 

   cb (xi,yi) (ind,di) = (xi',yi') 
   where 
      q5 = shiftR yi (ind-1) 
      q6 = shiftR xi (ind-1) 
      xi' = addSub di xi q5 
      yi' = addSub (complement (di) + 2) yi q6 

-- scale by 1/K = 0.6075 

scale2 x = y 
   where 
      s1 = shiftR x 1 
      s2 = shiftR x 3 
      s3 = shiftR x 6 
      s4 = shiftR x 9
s5 = shiftR x 13
m1 = s1 + s2
m2 = s3 + s4 + s5
y = m1 - m2
-- rotate according to the direction
addSub 1 a b = a + b
addSub 0 a b = a - b

-- LUT Module in Clash --
module LUT where
import Prelude
css 0 = [[]]
css n = concat [[-1:cs, 1:cs] | cs <- css (n-1)]
tangent cs = truncate $ 1024 *(tan $ sum $ zipWith (*) bs cs)
lut :: Int -> [Int]
lut n = map tangent (css n)
as = [45.0, 26.6, 14.0, 7.1, 3.6, 1.8, 0.9, 0.4, 0.2, 0.1]
bs = [pi/180*x | x<-as]

-- Resize Module in Clash --
module Resize where
import Clash.Prelude
myshiftR :: Signed 16 -> Int -> Signed 16
myshiftR inp n | n == 0 = inp
| n >15 = 0
| bv2v (pack inp)!!(n-1) == 0 = (shiftR inp n)+1
| otherwise = (shiftR inp n)
myext :: Signed 16 -> Signed 32
myext x = resize x
mytrunc :: Signed 32 -> Signed 16
mytrunc x = resize x

-- Update Module in Clash --
module Update where
import Clash.Prelude
import LUT
import Resize
type UpdateI1 = (Signed 16, Signed 16, Signed 16)
type UpdateI2 = Vec 10 Bit
type UpdateO = (Signed 16, Signed 16)
update :: UpdateI1 -> UpdateI2 -> UpdateO
update (a, b, c) ds = (b', c')
where
tanv = getTan (bitCoerce (v2bv (reverse ds)))
tmp = mytrunc $ scale1 $(myext a)* (myext tanv)
b' = b + tmp
c' = c - tmp
getTan :: Unsigned 10 -> Signed 16
getTan n = reverse $ (listToVecTH (lut 10)) !! n
scale1 x = shiftR x 10
module EVD where

import Clash.Prelude
import Clash.Explicit.Testbench
import CordicA
import CordicB
import Update
import CMC

type Ev1 = Vec 1 (Vec 4 (Signed 16))
type Ev = Vec 4 (Signed 16)
type Col = (Signed 16, Signed 16, Signed 16, Signed 16)
type EvdS = (Bit, EvdI, Matrix, Bit)
type EvdI = ((Signed 16, Signed 16, Signed 16, Signed 16),
            (Signed 16, Signed 16, Signed 16),
            (Signed 16, Signed 16),
            Signed 16)
type EvdO = (Ev, Evecs, Bit)
type Matrix = (Col, Col, Col, Col)
type Evecs = Vec 4 Ev

uptri_init :: EvdI
uptri_init = ((0, 0, 0, 0), (0, 0, 0), (0, 0), 0)
evinit :: Matrix
evinit = ((1000, 0, 0, 0), (0, 1000, 0, 0), (0, 0, 1000, 0), (0, 0, 0, 1000))
ev d inp = outp
where
  outp = mealy evdCore (0, uptri_init, evsinit, 0) inp

evdCore :: EvdS -> Ev1 -> (EvdS, EvdO)
evdCore (rst, uptri, evs, end) inp1 = ((rst', uptri', evs', end'), (evals, evecs, end'))
where
  inp = cmcCore inp1
  rst' = 1
  ((e11, e12, e13, e14),
   (e22, e23, e24),
   (e33, e34),
   e44) = muxx rst inp uptri
  r1 = 2 * e12
  u1 = e22 - e11
  r2 = 2 * e34
  u2 = e44 - e33
ds1 = cordic_a r1 u1
ds2 = cordic_a r2 u2
  (e22', e11') = update (e12, e22, e11) ds1
  (e44', e33') = update (e34, e44, e33) ds2
  (e13_tmp, e23_tmp) = cordic_b (e13, e23) ds1
  (e14_tmp, e24_tmp) = cordic_b (e14, e24) ds1
  (e13', e14') = cordic_b (e13_tmp, e14_tmp) ds2
  (e23', e24') = cordic_b (e23_tmp, e24_tmp) ds2
  (e12', e34') = (0, 0)
uptri_tmp = ((e11', e13', e14', e12'),
             (e33', e34', e23'),
             (e11', e13', e14', e12'),
             (e33', e34', e23'),
             (e12', e34'),
             (e11', e13', e14', e12'),
             (e33', e34', e23'),
             (e12', e34'))
(e44',e24'),
e22')

uptri' = muxx end' uptri_tmp uptri
evals = e11 :> e22 :> e33 :> e44 :> Nil
((v11,v21,v31,v41),
 (v12,v22,v32,v42),
 (v13,v23,v33,v43),
 (v14,v24,v34,v44)) = evs
evs' = muxx end' ((v11',v21',v31',v41'),
 (v13',v23',v33',v43'),
 (v14',v24',v34',v44'),
 (v12',v22',v32',v42')) evs
(v11', v12') = cordic_b (v11, v12) ds1
(v21', v22') = cordic_b (v21, v22) ds1
(v31', v32') = cordic_b (v31, v32) ds1
(v41', v42') = cordic_b (v41, v42) ds1
(v13', v14') = cordic_b (v13, v14) ds2
(v23', v24') = cordic_b (v23, v24) ds2
(v33', v34') = cordic_b (v33, v34) ds2
(v43', v44') = cordic_b (v43, v44) ds2
evec1 = v11 :> v21 :> v31 :> v41 :> Nil
evec2 = v12 :> v22 :> v32 :> v42 :> Nil
evec3 = v13 :> v23 :> v33 :> v43 :> Nil
evec4 = v14 :> v24 :> v34 :> v44 :> Nil
evec5 = evec1 :> evec2 :> evec3 :> evec4 :> Nil
end' = if (e12, e13, e14, e23, e24, e34) == (0,0,0,0,0,0)
  then 1
  else 0
muxx s a b = if s == 0 then a
  else b
topEntity :: Clock System
  -> Reset System
  -> Enable System
  -> Signal System Ev1
  -> Signal System Evd0
topEntity = exposeClockResetEnable evd
{-# NOINLINE topEntity #-}
testBench :: Signal System Bool
testBench = done
where
testInput = stimuliGenerator clk rst $(listToVecTH[(((1::Signed 16)::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil ),((1::2::3::4::Nil)::Nil )]
expectedOutput = outputVerifier' clk rst $(listToVecTH[(((1:: Signed 16)::4::9::16::Nil ),((1000::Signed 16)::0::0::0::0::Nil ) :>(0::Signed 16)::1000::0::0::0::Nil ) :>(0::Signed 16)::0::0::0::0::Nil ),0::
Bit),((0:0:25:5:Nil),((866:>(-502):0:0:Nil)):
(0:0:796:>(-604):Nil),((0:0:605:796):Nil):
(503:866:0:0:Nil),((0:0:30:0:0:Nil)):
(766:>(-442):>373:>(-282):Nil):((0:0:30:0:0:Nil)):
(766:>(-442):>373:>(-282):Nil):((0:0:30:0:0:Nil)):
(766:>(-442):>373:>(-282):Nil):((0:0:30:0:0:Nil)):
(766:>(-442):>373:>(-282):Nil):((0:0:30:0:0:Nil)):

\texttt{done = expectedOutput (topEntity clk rst \text{ (enableGen) testInput})}
\texttt{clk = tbSystemClockGen (not <$> done)}
\texttt{rst = systemResetGen}