AUTOMATING FPGA-BASED HARDWARE ACCELERATION

by

Songseok Choi

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Benjamin Carrion Schaefer, Chair

Dr. Diana Cogan

Dr. Mehrdad Nourani

Thank you to my beloved wife, Yousun Baek, and my one and only baby, Jiwon Choi.

AUTOMATING FPGA-BASED HARDWARE ACCELERATION

by

SONGSEOK CHOI, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

December 2018

ACKNOWLEDGMENTS

First and foremost, I would like to praise and thank God, for His showers of blessings throughout my research work and for its successful completion.

I would like to thank my thesis advisor, Dr. Benjamin Carrion Schaefer, for giving me the opportunity to do this research and for his constant guidance, motivation and never-ending belief that helped me successfully complete this research and grow as a researcher. I am highly inspired by his research aptitude and wonderful personality.

I would like to thank Dr. Diana Cogan and Dr. Mehrdad Nourani for being in my committee and spending their quality time reviewing my work.

I am grateful to the Department of Electrical and Computer Engineering at The University of Texas at Dallas for their support that provided me with ease throughout the journey.

November 2018

AUTOMATING FPGA-BASED HARDWARE ACCELERATION

Songseok Choi, MSEE
The University of Texas at Dallas, 2018

Supervising Professor: Dr. Benjamin Carrion Schaefer

In the field of field programmable gate array (FPGA), High-level synthesis (HLS) has shown to be a valid contender to traditional RT-Level based VLSI design based on low-level hardware Description Languages (HDLs) such as Verilog or VHDL. HLS facilitates hardware (HW) engineers do use FPGAs using high-level language, such as C / C ++ / System, by converting these automatically into efficient HDLs. Furthermore, HLS helps to reduce the development process time. In addition, HLS opens a door to software (SW) engineers and beginner HW engineers to the use of FPGA. However, HLS is still not a magic bullet and requires substantial HW knowledge to generate optimized circuits and more important to have a final working FPGA prototype.

This thesis aims at facilitating the use of FPGA to non-experts through HLS. The developed flow is built around simple templates so that SW engineers and HW engineers alike can easily make use of HLS and providing a full flow from HLS to a state-of-the-art configurable FPGAs composed of embedded processors and FPGAs, e.g., Xilinx Zynq FPGAs. The proposed flow makes the use of democratizes using the FPGAs, and shortens the design time substantially. In order to verify that the proposed flow is effective, extensive experimental results were conducted.

According to the measured results, these benchmarks could be accelerated by mapping the computationally intensive kernel on the FPGA. All of this analysis and results were made to ZedBoard Zynq - 7000 ARM / FPGA SoC Development Board using Xilinx 's tool.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machines
AXI	Advanced eXtensible Interface
CLB	Configurable Logic Block
CNN	Convolutional Neural Networks
DLL	Delay-Locked Loop
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
IP	Intellectual Property
KNN	k-nearest neighbors
PL	Programmable Logic
PLL	Phase-Locked Loop
PS	Processor System
UART	Universal asynchronous receiver/transmitter
RTL	Register-Transfer Level
SDK	Software Development Kit
SoC	System on Chip

TCL	Tool Command Language
VHDL	VHSIC Hardware Description Language
XSCT	Xilinx Software Command-Line Tool

CHAPTER 1

INTRODUCTION

1.1 Thesis Motivation

Due to the recent rise of artificial intelligence, machine learning and cloud-based systems, dedicated integrated circuit (IC) that can exploit the massive inherent parallelism of these, such as FPGAs, are widely being deployed. Machine learning and deep learning have been shown to work extremely well for videos and images processing [1]. These technologies are needed in order to process vast amounts of data in a very short time. Developers, therefore, naturally tend to look for devices with low cost, high computational performance, and high energy efficiency.

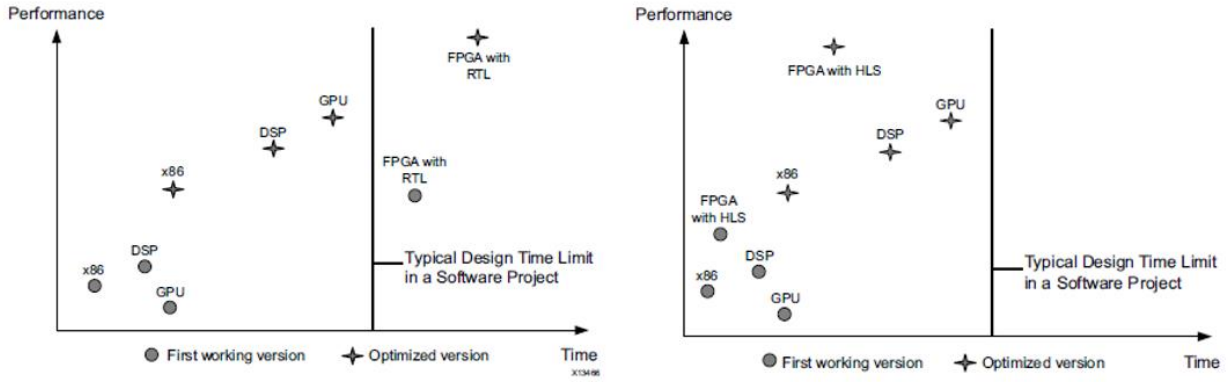


Figure 1.1. Design vs. application performance with RTL and HLS Comparison [2]

As you can see in Figure 1.1, FPGAs lead to better performance than general purpose solutions. Furthermore, FPGA combined with HLS have shown higher performance than FPGAs with RTL (Verilog or VHDL). This has led to the interest in other, non-hardware expert, communities in the use FPGA to e.g., accelerate computationally intensive scientific applications, cloud computing,

etc. These non-hardware experts are often pure SW engineers with limited or non-hardware knowledge, but who are involved in creating various algorithms and development processes of artificial intelligence, machine learning, and cloud-based systems. Thus, they need a way to easily target FPGA.

Despite this, SW engineers encounter several challenges when targeting FPGAs. First, FPGA vendors originally only allowed to be configured using hardware description language (HDL), such as VHDL/Verilog/System Verilog [3]. This implied that SW engineers had to learn these low-level HDLs in order to use the FPGAs. Second, although FPGA has many advantages, it is not easy to use for a SW engineer because it involves unfamiliar terms, tools, and other difficult development procedures. Lastly, SW engineers have accumulated an enormous number of open-source algorithms, which had to manually be converted into an HDL to be mapped on a FPGA. Being able to configure FPGAs using software language such as C/C++ allows SW engineers to easily target these FPGAs

In addition, before starting the FPGA development process, most HW engineers prefer to first verify their algorithm using model-based design simulation, via MATLAB, Simulink, or C-based design (e.g., C/C++/SystemC) [2, 4]. This enables HW engineers to verify if the functional correctness of the model. This process is essential, and increasingly accurate simulation models are obtained through trial and error, which requires a large amount of time in order to ensure accurate results [2]. After the verification process, if the results of the algorithm meet the desired results, the engineer can develop the algorithm using the HDL. Also, it is expected that the development period can be shortened if the sources used in this verification are converted directly to HDL language. This is basically what HLS does.

In past decades, HLS has been extensively studied and recently it has started being used in commercial product designs [5, 6]. Around the same period, hardware and software co-design has become increasingly important due to the heterogeneity of ICs. We will discuss HLS in more detail in Chapter 2. Basically, HLS can be defined as a process that converts untimed behavioral descriptions into efficient RTL [7, 8]. One of the problems with HLS is that it is a single process synthesis method. Thus, the interface between the other modules in a complex SoC, e.g., embedded processor, has to be manually done. This thesis aims at bridging this problem by providing a complete flow from untimed C description to a configurable SoC FPGA composed of embedded processors and reconfigurable fabric.

1.2 Thesis Contribution

The following are the contributions to the current thesis:

- HLS templates were developed to make HLS easier to use for SW engineers and HLS beginners.
- Created an automated design flow that *stitches* all the tools required to map a single behavioral description for HLS onto a configurable SoC (e.g., Xilinx Zynq FPGA).
- Perform comprehensive experimental results to validate our approach and measure the runtime speedup achieved when a task is mapped on the FPGA vs. run on software only.

1.3 Thesis Organization

The organization of the remainder of this thesis is presented in this section. Chapter 2 explains HLS. Chapter 3 explores typical FPGA architectures and puts more emphasis of modern Configurable System on Chip (SoC) FPGA, like Xilinx Zynq FPGas. In particular, its architectural features and PS-PL overview of Zynq-7000 SoC device, AXI interface [9]. Chapter 4 motivates the use of HLS templates and their structure. Also, C-based source/testbench, code rule in HLS and factors to consider when creating a template using code generator are discussed. Chapter 5 explains system design flow in Xilinx HLx. How to use Vivado HLS, Vivado Design Suite, Vivado SDK and GUI mode as well as batch mode of each individual tool. In addition, it is explained that how it makes automation in batch mode. Chapter 6 discusses the results of 6 benchmarks and analysis of resource and execution time in each benchmark. Chapter 7 concludes with a summary and presents possible future work originating from this thesis. All projects in this thesis were developed using Xilinx FPGA Zynq-7000 associated and tools.

CHAPTER 2

HIGH-LEVEL SYNTHESIS (HLS)

2.1 Introduction

Recently, due to the increased complexity of applications and growing capacity of silicon technology, design methods and tools are moving to levels involving higher abstraction. The need for synthesis, verification, and automatic acceleration have been continuously relied upon as important factors in the system design process. This has led to various attempts for efficient synthesis, and in the 1990s, commercial tools using high-level synthesis techniques were first introduced. At the same time, hardware-software co-design - including estimation, exploration, partitioning, interfacing, communication, synthesis and co-simulation - began to attract more attention [10]. Co-design was due to problems encountered from the beginning of system design and increase efficiency by making them easier to debug. More cooperation between hardware engineers and software engineers was required. To make co-design easier, several commercial C or VHDL code generators have been developed which were suitable for final design. The initial HLS version did not yield any better performance than RTL. However, as HLS becomes more developed and more optimized, it has begun to be viewed as having similar levels of performance as RTL [11].

2.2 High-Level Synthesis Benefit

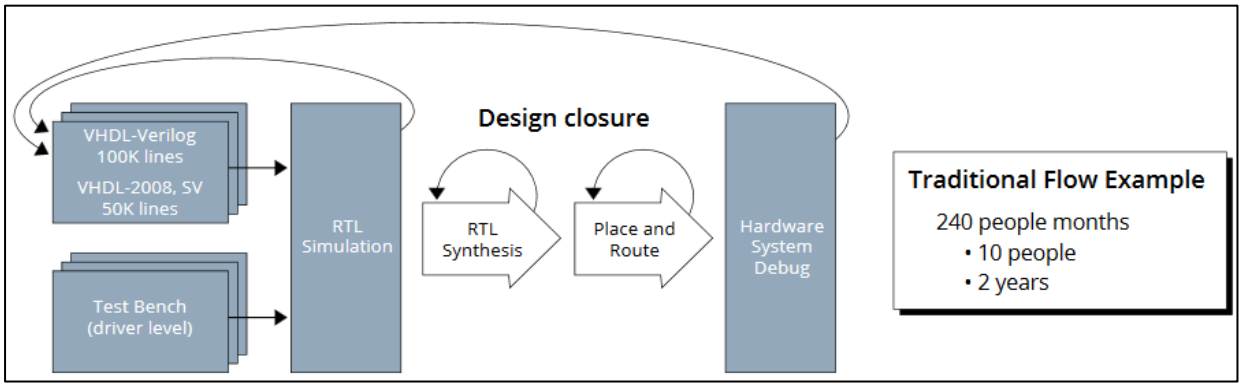


Figure 2.1. Traditional RTL Design Flow

First of all, it is necessary to compare traditional FPGA RTL design flow with modern design flow using HLS. When using traditional FPGA RTL design, RTL design cycle is composed of verification and design closure iterations for each block as well as the entire design. The connected

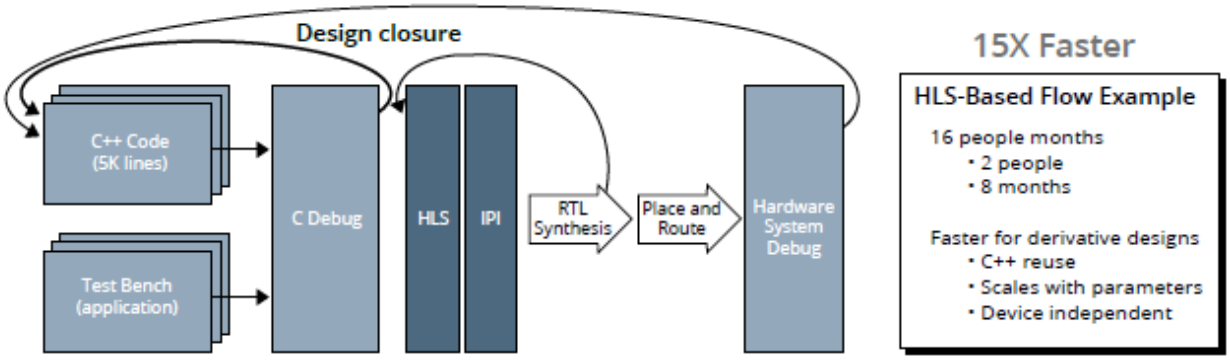


Figure 2.2. Modern HLS Design flow

targeted platform design can be unstable because if differentiated logic can be changed, IO interface may be affected thus failing timing requirement. Therefore, overall development time is expended as shown in Figure 2.3. Modern HLS design flow can reduce development time and requires fewer people to process the same project, as shown in Figure 2.4 [8].

Not all projects follow the model shown in Figure 2.4, but these results have important implication for FPGA design. This is because performance can be improved by using hardware that requires parallelization in algorithms and can be processed by software in other parts, which can lead to performance improvement [5]. In addition, HDL uses timing simulation for debugging. This has the characteristics of timing simulation that involves a synchronized clock, making debugging difficult and time-consuming. In contrast, it is possible to reduce development time since C-based language can be debugged sequentially, rather than via clock synchronization. Lastly, the C-based language has a variety of open sources that are available for reuse [5].

2.3 Introduction to C-Based FPGA Design

Generally, C-based FPGA design uses HLS tools. The HLS compiler provides a similar development environment to enable application development, using C-based sources such as C/C++ and SystemC. HLS shares methods of the SW process compiler to provide similar interfaces, analysis, and optimization of C/C++ programs. Therefore, when a C-based source is compiled within HLS, it will perform interpretation, analysis, and optimization to suit the desired target [2]. In addition, HLS executes to synthesize C-based source that is required acceleration to HDL. This generated HDL is converted to RTL by the HLS compiler and then it is executed with parallel programming in FPGA known as Programming Logic (PL) in Zynq Architecture, as the latest generation of Xilinx's All programming System-on-Chip (SoC) families, combines a dual-core ARM (Advanced RISC Machines) cotex-A9 as Shown in Figure 2.3. Other typical parts and kinds of testbench do not require acceleration is transferred and operated on the ARM side.

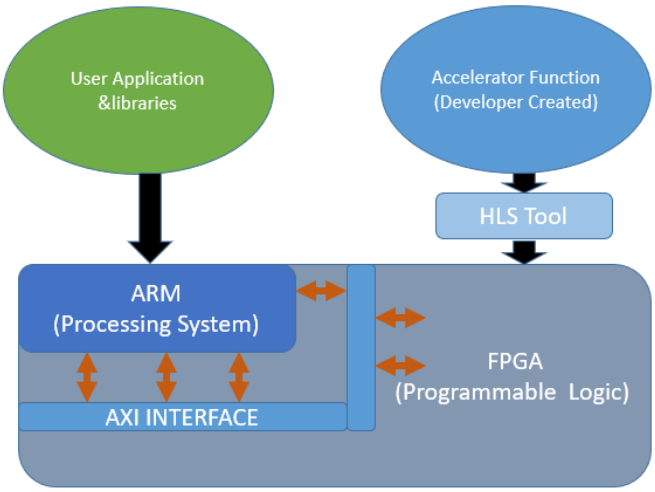


Figure 2.3. C-Based Hardware Acceleration Block Design on Zynq Architecture

Therefore, by using HLS, as an individual, one can program applications or design projects that are required for hardware acceleration with the characteristics of C language. In addition, HLS engineers utilize FPGA's general advantages such as parallelism, low cost, and relative power efficiency [4]. Before I explain High-Level Synthesis design flow, it is better to have understanding about the current configurable SoC design flow. We will look at the tool flow of the most popular SoC FPGA, Xilinx company, which we have chosen as the target. Figure 2.4 shows the overview of the Xilinx tool flow.

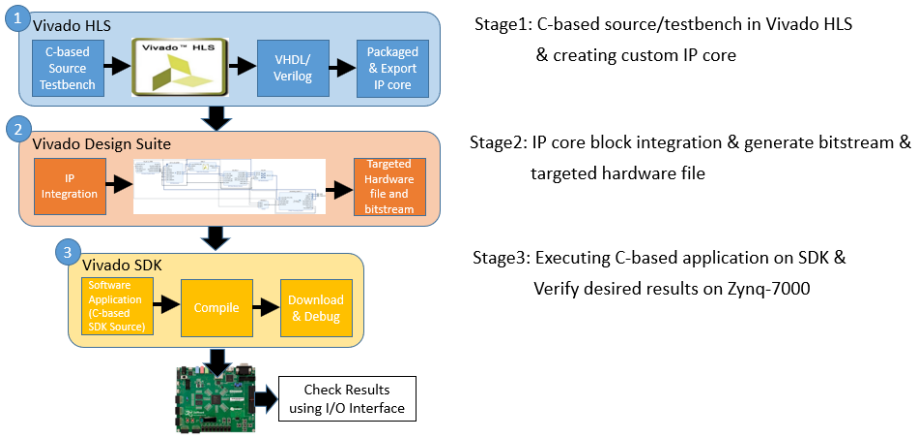


Figure 2.4. Overview of the Xilinx Tool Flow

Xilinx tool flow has three stages. The first stage is about High-Level Synthesis. This stage is that C-based source/testbench is compiled by HLS compiler and synthesized and also converts the HDL and RTL. This RTL is transformed and packaged by the IP core, and thus, it exports this IP core to the integration stage. Second is the IP integration stage. This stage is to integrate block design IP core from HLS with ZYNQ7 Processor system (ARM), Logic synthesis & implementation, Generating Bitstream and Export to SDK and the Hardware handoff file. The last stage is about Software generation. This stage can create software application for the integrated block design with the IP on the Vivado SDK tool. The result is verified using SDK on terminal and ZYNQ I/O interface.

2.4 High-Level Synthesis Design Steps

High-Level Synthesis has three main steps for the following: Allocation, Scheduling, and binding. Understanding these fundamental concepts goes a great way towards providing three main steps. Consider an example of y=a+b+c+d as Shown in Algorithm 1.

Algorithm 1. Example C-based Code to High-Level Synthesis Design Steps.

```
1: Void accumulate( int a, int b, int c, int d, int & dout ) {
2: int t1,t2;
3: t1 = a + b;
4: t2 = t1 + c;
5: dout = t2 + d;
6: }
```

The HLS process begins by analyzing the data dependencies. The analysis leads to Data Flow Graph (DFG) description shown in Figure 2.5. Each node of the DFG transforms an operation in the C-based code. All operations use the “add” operator.

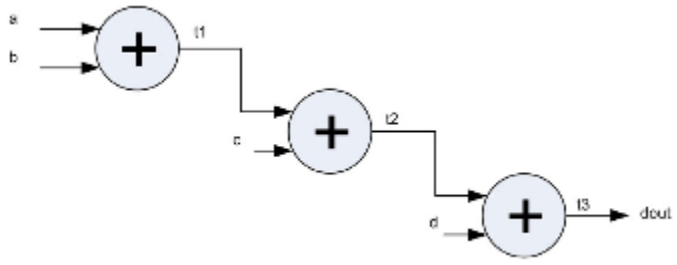


Figure 2.5. Data Flow Graph Description

Allocation

After the DFG is assembled, each operation is transformed to a hardware resource that is used during scheduling. This process is defined as the resource allocation. Allocation defines specific types and numbers of hardware resources (e.g., functional units, storage or connectivity components) that will be necessary to satisfy the design constraints. functional units, storage or connectivity components from a hardware library as shown in Figure 2.6 [12]. Depending on the HLS tool, some components may be added during scheduling and binding operations. For example, the connection components such as buses or point-to-point connections components among can be added before or after binding and scheduling operations. The components are selected from the RTL component library which also contains component characteristics such as area, delay, and power. By default, the HLS tool will try to maximize the parallelism as much as possible in the

scheduling stage and hence allocating large number of functional units, which will always serve as an advantage.

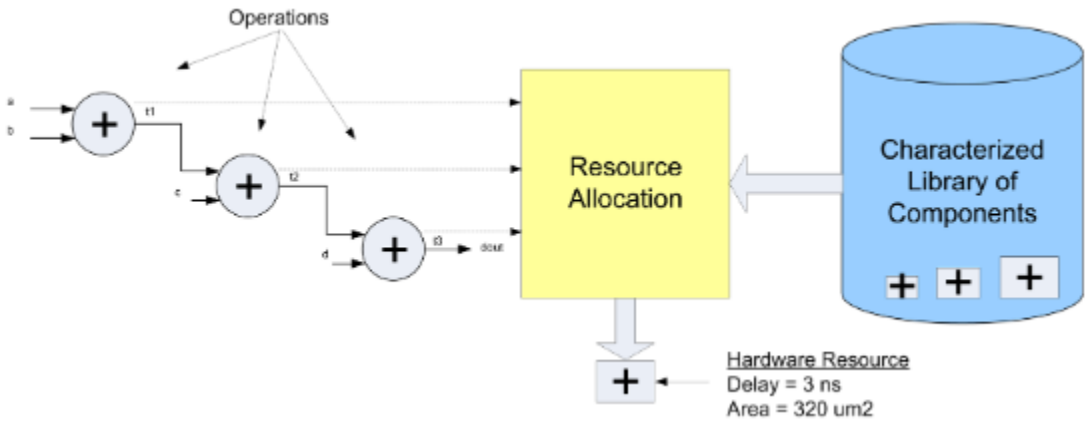


Figure 2.6. Resource Allocation

Scheduling

Scheduling defines which operations should be determined during each clock cycle as shown in Figure 2.7 [12]. This should be decided by the following factor: length and clock frequency of the clock cycle, time-taken for each operation to complete on the target device and user-specified optimization directives [4]. For a brief example, many operations can be completed in one clock cycle if the clock cycle is long enough. On the contrary, if the clock cycle is very short, HLS will calculate and schedule how many more clock cycles should be deployed and how many more clock cycles are optimized.

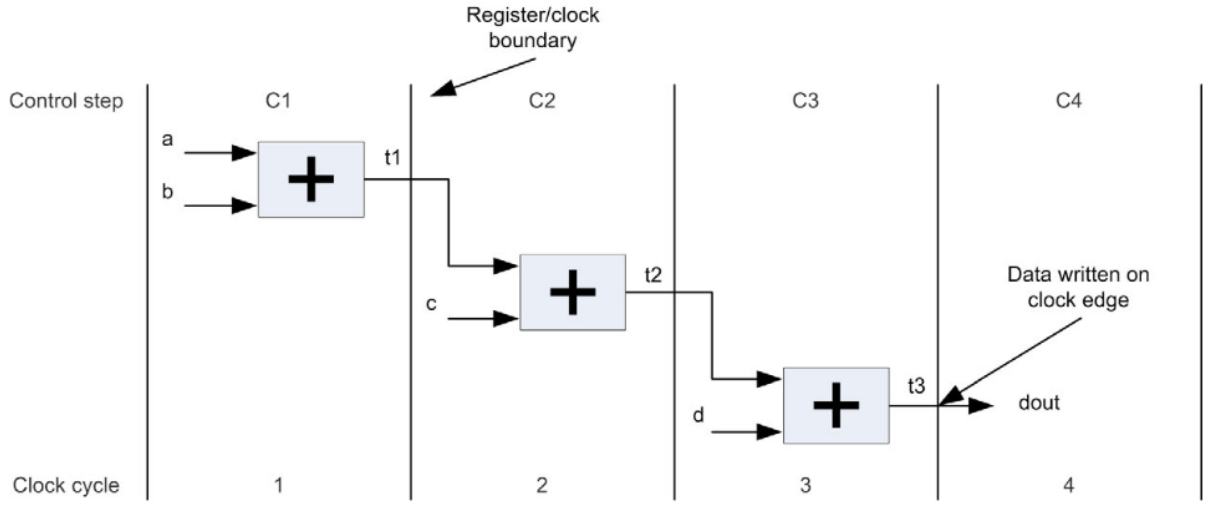


Figure 2.7. Scheduled Design

Binding

All operations must be carried out according to the clock cycle and bound to the storage unit. In addition, when different operations are performed together on the same cycle, they can be bound to the same storage unit over a range of non-overlapping or mutually exclusive [10]. Binding method determines the hardware resources that implement the HLS scheduled operation. HLS uses information from the target device to find the optimal solution [4].

Synthesize rule for C-based source

Generally, HLS follows several rules to synthesize C-based sources [4].

− Top-level function arguments should synthesize into RTL input/output ports.

− C based function synthesizes into blocks according to the RTL hierarchy.

− Basically, loop in the C based function remains in the rolled.

− In C code, arrays synthesize to be stored using block ram in the final FPGA design. Therefore, if C code declares an excessive array beyond the target device's memory, synthesis may not be possible. These rules may vary for each tool used by the manufacturer of the target device, but the HLS tool designer generally put into account these rules and develop the HLS tool.

2.5 Vivado High-Level Synthesis Design Flow

HLS tool converts c -based sources into RTLs to create custom IPs. This sequence is shown in Figure 2.8. We want to describe this process in more detail from the perspective of the engineer. First of all, we go through HLS design flow: c-simulation, synthesis, co-simulation using c-based

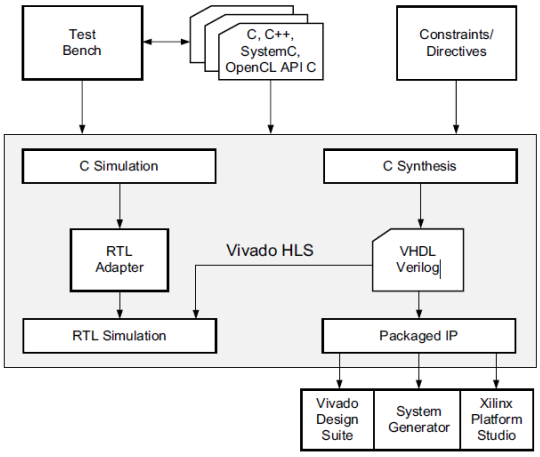


Figure 2.8. Vivado HLS Design Flow

source and testbench and then verify the result. If the desired result is not achieved in c-simulation, it is necessary to correct the code again and repeat the above process. If the desired result is obtained, it converts c-based source to RTL by using the HLS tool. HLS user checks the result from co-simulation with converted RTL and c-based source. If co-simulation does not perform the appropriate value, modify the code again and repeat it until co-simulation has accomplished the desired result. If the expected result is obtained, package this RTL to IP core through logic synthesis, implementation, and integrate IP core with targeted hardware system as shown in Figure 2.8.

2.6 Summary

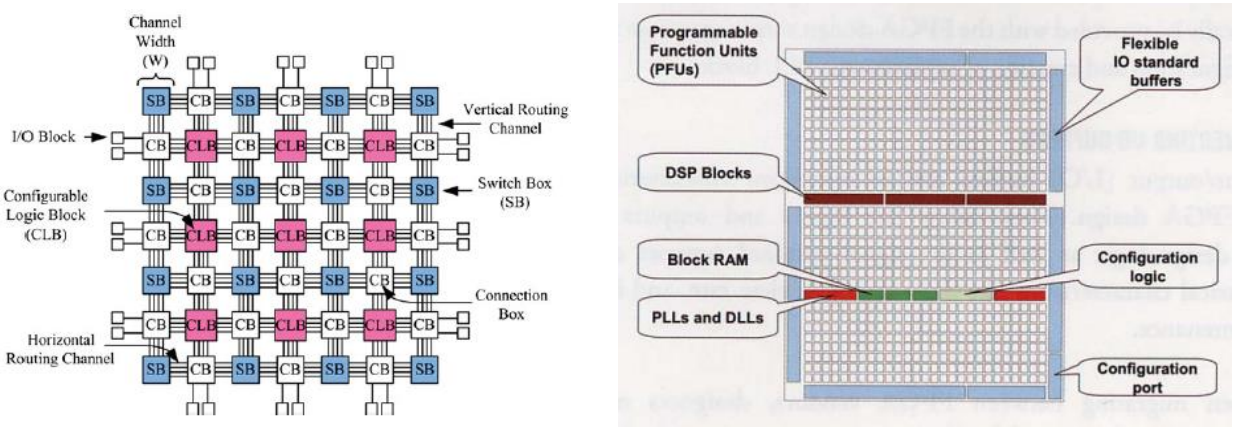
HLS tools transform C-based source/testbench to RTL design. C-based design can reduce development time and allows SW engineer to access FPGA-based design. And the advantage is that HLS can reuse C-based algorithm. Therefore, various algorithms can be applied to FPGA-based design more easily.

CHAPTER 3

MODERN SOC FPGA

3.1 Introduction

This chapter explain what FPGAs are and discusses their internal architecture. It also reviews the architectural change that have happened, since they first appeared. The chapter includes discussion about how Processing System (PS) and Programming Logic (PL) work in modern FPGA architecture, complement each other, and improve performance.



(a) (b)

Figure 3.1. Traditional FPGA architecture (a) and Modern FPGA(b) [13]

The traditional FPGA had an architecture called “island-style” as shown in (a) of Figure 3.1. I will briefly explain the components of traditional FPGA. The I/O blocks are connected to the periphery of the chip for external connection. In general, an I/O block is connected to the logic block to send and receive data. A logic-block is usually called configurable logic block (CLB). The logic blocks are surrounded by switch blocks and connection blocks. The wires in the channel

are connected with other CLB in each zone and the zones of each wire vary. The switch block is connected by each wire and used as a programmable routing switch or a programmable connection switch. It is made of pass-transistor or bi-direction buffer [13, 14].

The architecture of the modern FPGA consists of a combination of various blocks added to the traditional FPGA architecture for performance improvement and efficiency. Additional main blocks are PLL/DLL, DSP, embedded block ram. The role of PLL/DLL is to generate various clocks by dividing and multiplying the clock, and can also adjust the I/O timing. DSP can make high-speed digital signal processing more effective. Embedded Block RAM will help store and output data faster [15].

3.2 SoC FPGA architecture

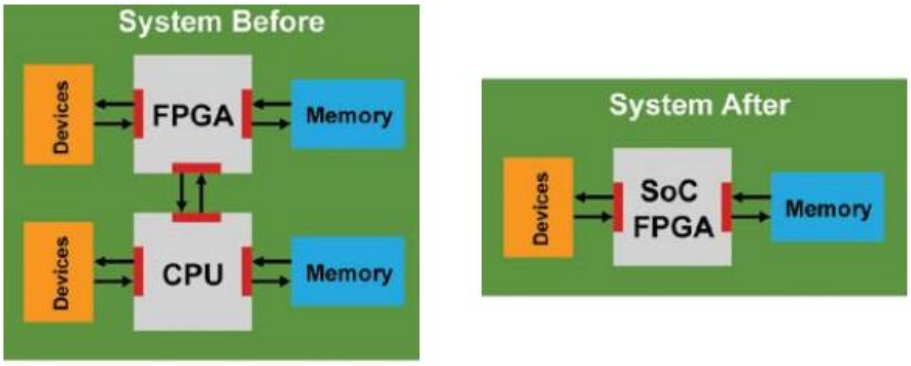


Figure 3.2. SoC FPGA Trend [16]

Prior to SoC FPGA, the FPGA architecture was configured in the system with the processor. The CPU and FPGA were connected to each other through an external communication interface. It has been used for conducting various applications. Since the FPGA has the structure of the SoC FPGA, which consists of only one IC, it has the advantages of the high bandwidth and low latency of

internal communication. Along with this, the efficiency and performance have been increased, and the size of the whole system has been reduced. It has been applied to a wide range of fields as shown in Figure 3.2 [16,17].

Recently, Alta Stratix and Xilinx Virtex-5/Zynq 7000 have become major brands of SoC FPGA in the market [17]. Since research used to develop several projects used Xilinx ZedBoard Zynq-7000 ARM/FPGA SoC Development Board, we will explore the structure and functions of Zynq-7000 to learn more about the characteristics of modern SoC FPGAs.

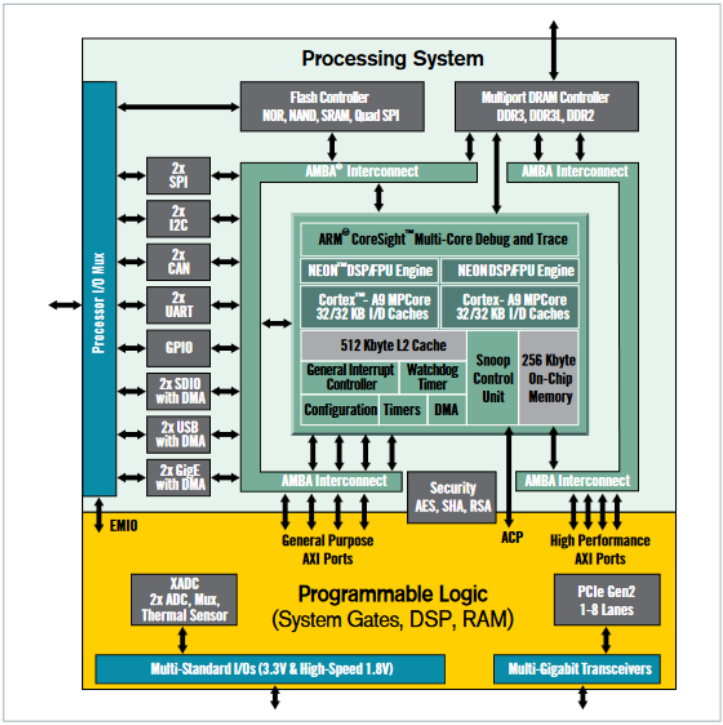


Figure 3.3. Zynq-7000 SoC

Xilinx Zynq Structure

Figure 3.3. delineates the functional blocks of the Zynq-7000 SoC [18]. PL and PS are separated and communicated via AMBA and AXI interface. PS consists of ARM and is characterized by

having SDRAM and Flash memory [18]. Also, PL consists of more memory and DSPs than the previous FPGA.

Processing System (PS)

The application processor unit (APU) has a dual ARM Cortex-A9 MPCore with version 7 ARM ISA. The maximum frequency is 1GHz. It has 512MB DDR3 memory and has a 256 Mb Quad-SPI Flash. There is a 32KB L1 4-way set associative instruction and data cache, and a 512KB L2 8-way set associative data cache with supporting byte-parity.

Programming Logic (PL)

PL is based on the artrix-7 FPGA logic of Xilinx. The PL contains CLB that has a 6-input look-up table, memory capability within the LUT, Register/shift register and cascadable adders. It includes 36KB block RAM, and DSP with 48 bit high-resolution. The PL resources consist of 13,300 Logic slice, 53,200 LUTs, 140 BRAM and 220 DSPs.

AXI Interface

AXI is the Advanced eXtensible Interface protocol. Xilinx has been employed from Spartan-6 and Vertex-6 device. In addition, Zynq-7000 uses AXI to communicate with PL and PS. There are three types of AXI. First is AXI4, which is for high-performance memory-mapped interfaces. Then,

AXI4-Lite is the low-throughput memory mapped interface. Lastly, AXI4-Stream is used for high-speed streaming data for video/audio processing [19]. The Figure 3.4 is a AXI interface diagram that shows a streaming data transfer between the processor and FPGA fabric on Zynq platform. Generally, the AXI4-Stream interface is used together with a DMA (Direct Memory Access) controller to transfer a large of data from the processor (ARM) to the programmable logic (FPGA). In general, this data is transformed as vector data on the software side. The DMA controller reads the vector data from the memory and streams it to FPGA through the AXI4-Stream interface.

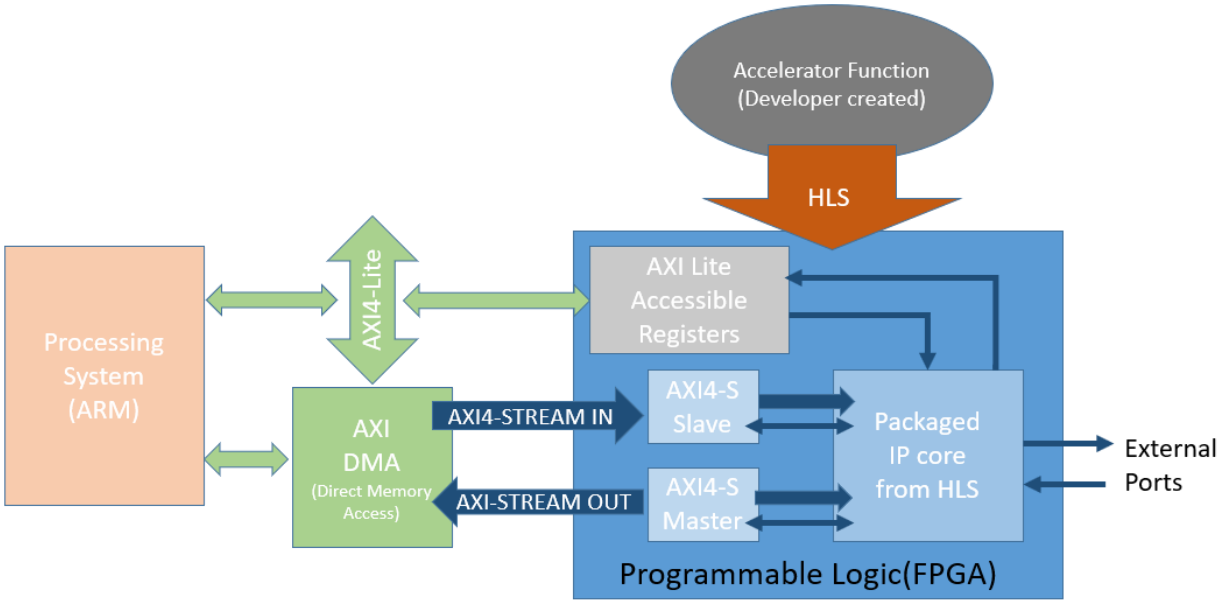


Figure 3.4. AXI Interface Diagram on Zynq Architecture

3.3 Summary

Modern SoC FPGA architecture is a FPGA trend. Through this, a wide range of applications are being developed. In addition, structures that use the AXI interface by configuring PL and PS as SoC that are well evaluated in terms of performance and efficiency. In particular, Xilinx Zynq

series will be used in my research as a representative device for the Soc FPGA. It appears that more research/development will take place on this Soc architecture.

CHAPTER 4

C-BASED TEMPLATE BASED FPGA DESIGN

4.1 Introduction

This chapter presents the advantages of using templates to facilitate the programmability of FPGAs. Two essential input files are required to convert a C-based model to custom-IP using the Vivado HLS tool. The two files are the C-based source file and the C-based source's testbench file. The testbench is not synthesizable, and thus, will be mapped to the embedded processor, while the computationally intensive kernel will be mapped onto the FPGA.

These two files must follow a set of rules of HLS. In other words, one must set various HLS settings, such as interface for directive HLS, AXI interface setting, and changing function/library names and so on. For example, to create an input using the HLS AXI stream interface, one should enter the code as follows using Pragma Directives "#pragma HLS resource core=AXI4Stream variable=in". However, these make it difficult for SW engineers to access HLS because they have to study HLS anew and learn as well as reflect on the development. Therefore, we created a template for HLS for SW engineers and engineers who are new to HLS. This template is for Vivado HLS and we will also cover how to use it in this chapter.

4.2 C-source /testbench file

As described in Chapter 3, HLS requires two types of files. The source/testbench files should be programmed as shown in Figure 4.1 [2]. You have created a code generator that automatically

generates c-based sources/testbench. This code generator generates two files. One source is C-based source template and the other is C-testbench template, which is testbench.

Test Bench Code

```
int main()
{
   int i;
   int B[10];
   int C[10];
   int result;

   for(i=0; i < 10; i++){
        B[i] = i;
        C[i] = i;
   }
   result = example(B,C);
   return result;
}
```

Algorithm Code

```
int example(int B[10], int C[10])
{
   int i;
   int A=0;

   for(i=0; i < 10; i++){
         A += B[i] * C[i];
         if(i == 11)
              A = 0;
   }
   return A;
}
```

Figure 4.1. Example codes for HLS

4.3 Instruction of code generator for HLS

This code generator requires the configuration file input and type by the user first. Configuration files consist of array size, function name, input file name, golden output txt file name, result txt file name, input variable and output variable. The code generator reads the information from the configuration entered by the user and reflects it in the file to be generated. In this way, one can create the configuration.txt by typing the information mentioned above. When executed, two files are created in Appendix A.1 and A.2. This code generator design flow is as shown in Figure 4.2.

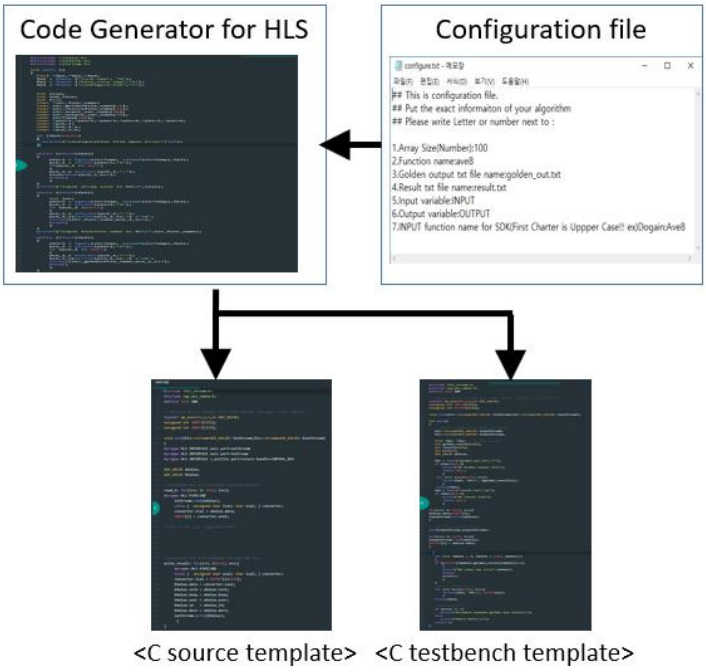


Figure 4.2. Code Generator Design Flow for HLS

So, I set the configuration files as follow: array size: 100, function name: example, input file name: input.txt, golden_out file name: golden_out.txt, result file name: result.txt, input variable: INPUT and output file name: OUPUT. The source and testbench files for the HLS are generated according to the input values of this configuration. A brief description of each file is given below.

The first thing to notice in the source file is the SIZE in the third line. SIZE refers to array size. This array size is the size of the array that is entered from the input.txt file. It was assumed that the input size from Input.txt and the result.txt's output of the array size are the same. The second point of interest is the need to code the algorithm, which is the main point of the template as shown in Figure 4.3. To code the desired algorithm, enter the desired algorithm at the bottom of the 19th line. An important point is to create your own algorithm using the INPUT[i]/OUTPUT[i] array

variables. For example, if you want to write an algorithm y[i]=x[i]*5, it should be in the following form.

Example1. algorithm code

```
1: for (int i=0; i<SIZE; i++) {
2:      OUTPUT[i]= INPUT[i]*5;
3:      }
```

So far in this chapter, we have learned that engineers who want to use HLS can easily implement algorithms using this template. An individual only needs to write the algorithm using the array size and input / output variable name they set.

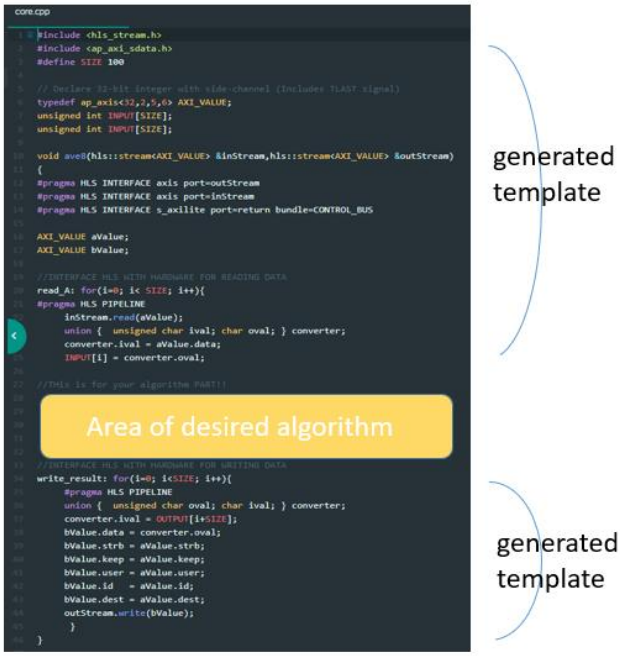


Figure 4.3. Input Desired Algorithm in C-based Template for HLS

Here, I will briefly explain the testbench file. First, you will receive two files. One is an input file and the other is a golden out file. Input.txt contains data to enter into the algorithm. The golden out file has expected data for verification against the results. In the 45th line, input data

will be sent to core function as much as the array size. Then call up the Algorithm function in the 50th line and the core function performs what we want. The function will perform and return data of the array size. In the 56th line, it compares the entered data with the gold out and add 1 to the error variable if the wrong result is generated. The 69th line will then be used to print the "Mismatch between gold and result." If the result is good, "Match data!" is printed out.

4.4 Summary

In this chapter, I explained how to build a HLS template. HLS is not easy for SW engineers or FPGA beginner to access because they have to learn new information about HLS in order to use HLS. This template not only reduces development time but also eliminates the time to study information about HLS.

CHAPTER 5

AUTOMATION OF C-BASED HARDWARE ACCELERATION METHODOLOGY

5.1 Introduction

This chapter presents an fully automated flow that "stitches" all the Xilinx tools required to

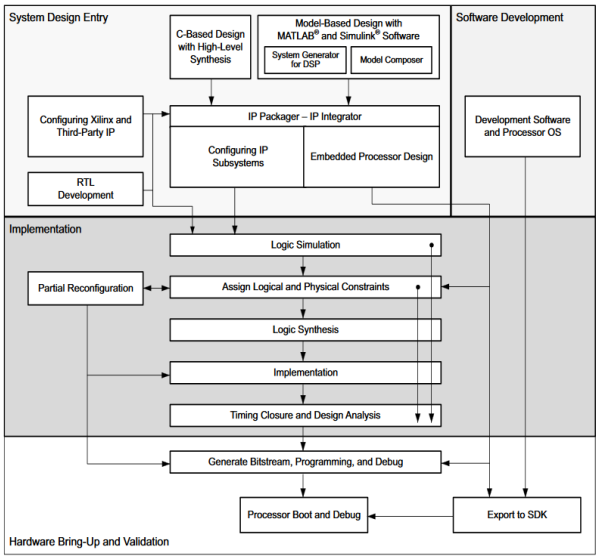


Figure 5.1. Vivado System Design Flow

program a complex configurable SoC FPGA. Before proceeding further into this chapter, it is necessary to understand the Vivado System Design Flow. Traditional FPGA Design Flow uses HDL to synthesize the RTL level into IP core and logic simulation as shown in Figure 5.1 [20]. After that, the traditional FPGA Design tool performs logic synthesis, implementation, Place and Route and then generates a bitstream. Generated bitstream is imported to FPGA and then user can check expected result using I/O ports [21,22]. In the process of using this design flow, a new way to improve productivity is to introduce automation design methods. Also, it has been that the

development time could be reduced if this method was properly implemented [23].

5.2 C-Based Design Flow on Xilinx FPGA

Prior to describing the automation design method, I will examine the system design methods and tools currently used in Xilinx. To develop algorithms using HLS, HLS tool had to have new functions. That's why Xilinx companies have been able to divide functions by tool and develop three tools. Three Vivado tools will be used for HLS design flow. Below is a description of each of the three tools.

-Vivado HLS

-Converts the C-based source, which is a high-level language, into RTL and IP core.

-Vivado Design suite

-Adds the generated IP core by HLS and integrates with ARM core, AXI interface, and DMA block.

-Performs logic synthesis/implementation/generates bit file and then exports bit file into FPGA. In addition, sets the targeted hardware platform to export and launch SDK.

-Vivado SDK (Software Development Kit)

-The hardware platform and bit file created in the Vivado design suite is input and forwarded to ARM and FPGA respectively.

-Creates an application project and codes an appropriate C-based source to perform the project and check the desired results.

5.3 Proposed Automating C-based Design Flow

I want to explain proposed C-based acceleration design flow. It has three stages. The same stages with previous design as we can see in Figure 2.2. In order to facilitate C-based hardware acceleration for SW engineer and FPGA beginner, I design automating methodology for convenience and reducing development time. First stage is automating HLS has 3 steps.

Second stage is automating IP integration. Last stage is automating software generation has two steps in shown Figure 5.2.

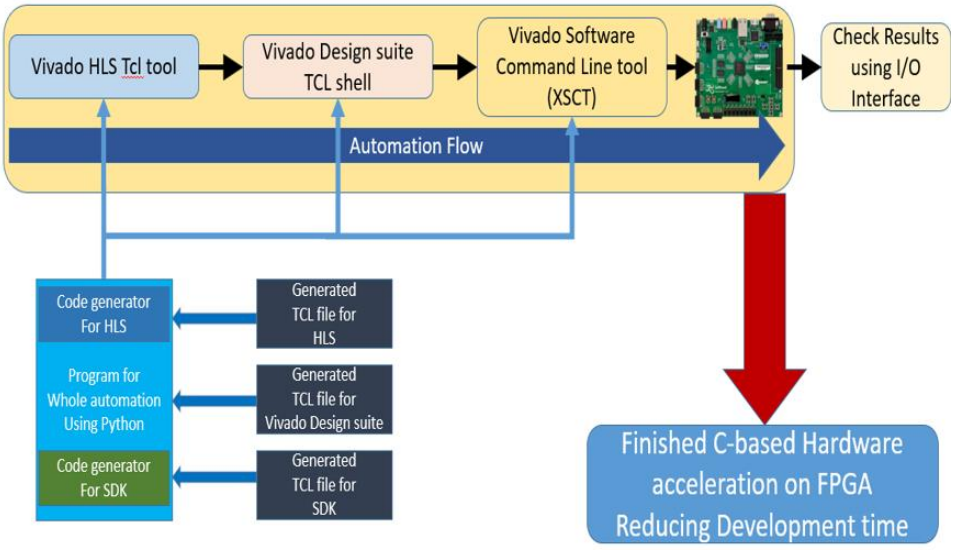


Figure 5.2. Proposed Automating C-based Design Flow

5.3 Automation of Xilinx FPGA Design using batch mode

Each tool supports two modes. One is in GUI mode and the other is in batch mode. The general method is to use these three Vivado tools in GUI mode as it is more intuitive, making the tools

more convenient to use. Currently, most GUI programs perform each work with clicking mouse button. To set the configuration or change a property of the project, application must also be set using the GUI internal menu bar by clicking. However, it takes time to find the menu bar and click mouse's button every time. To solve this trouble, FPGA vendors provide automatic generation methods using TCL-based scripts. This has further improved the productivity of implementing algorithms [24]. Also, tool command language (TCL) is a kind of programming language designed to use command-based tools [25].

5.3.1 Vivado HLS Batch Mode

Vivado HLS supports batch mode using TCL. For automation, only the HLS TCL file is needed. This will create a TCL file automatically when the project is created using GUI mode. The file is created with the name script.tcl. We will look at the script.tcl file.

Example2. Script.tcl for vivado HLS

```
1  ############################################################
2  ## This file is generated automatically by Vivado HLS.
3  ## Please DO NOT edit it.
4  ## Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
5  ############################################################
6  open_project example
7  set_top example
8  add_files core.cpp
9  add_files -tb test_core.cpp
10
11 #Solution
12 open_solution "solution1"
13 set_part {xc7z020clg484-1} -tool vivado
14 create_clock -period 10 -name default
15 #source "./example/solution1/directives.tcl"
16 csim_design
17 csynth_design
```

```
18 cosim_design -rtl vhdl
19 export_design -rtl vhdl -format ip_catalog
20 exit
```

First, one must set the project and top name in the 1st and 2nd line. As seen on the 7th and 8th lines, a source/testbench file created in Chapter 4 can be added to the project. You can also insert additional input files using the command add_file. The 12th to 14th lines are about project settings. You can set the Device part name and enter the period of the clock. The 16th to 19th lines are the commands that perform the C-simulation/synthesis/co-simulation/export design IP in order. You can use the Script.tcl file to perform the HLS design process in the Vivado HLS batch mode. Run the Vivado HLS command from the program and navigate to the folder containing the script.tcl file. When you run the Vivado_hls and script.tcl on the command line, the programs entered in script.tcl will run one line after another.

5.3.2 Vivado Design Suite Batch Mode

Vivado design suite is a tool to integrate custom IP which is created by Vivado HLS with other IP blocks such as Zynq PS block and AXI interface block. If you configure the block with the GUI, you can see the configuration as shown in Figure 5.3. Creating blocks using a GUI takes a relatively large amount of time. This is because each block must be searched using each IP name on the menu bar. In addition, added IP and input and output ports are connected using wire by mouse device. However, using automation in batch mode can shorten those times.

As with this Vivado HLS tool, it can be automated using TCL in batch mode.

This vivado design suite.tcl file for automation in Vivado Design Suite is attached in Appendix

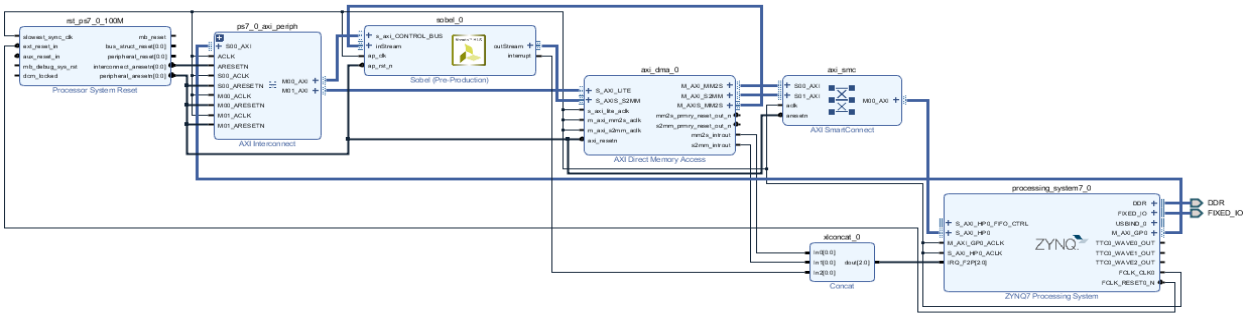


Figure 5.3. Block Design

A.3. In order to use this tcl file, it is easy to automate by changing only the project name, target board part number, IP core folder location, and the name of the project folder in the code. This script.tcl runs in the Vivado tcl shell program. Move to the folder where the file contains and enter the command "source file_name". As each line is read sequentially by the compiler, custom IP is added, blocks are created, logic synthesis, implementation, place and route, and generating bitstream are performed automatically.

5.3.3 Vivado SDK Batch Mode

The Vivado SDK can use batch mode via XSCT (Xilinx Software Command Line Tool). Before we do automation in SDK, in order to execute the Vivado SDK, a person must code the C-based source. The C-based source for SDK is not easy to create as a SDK beginner because there are several SDK-oriented libraries, drivers, and directives. As such, I developed a SDK code generator for SDK beginners and engineers. C-based source for SDK is generated via this code generator. The description of the SDK code generator will be briefly described in the appendix. For automation, a TCL file for SDK is required. This is a TCL file for automation in Vivado SDK.

Example3. Script.tcl for vivado SDK

```
1  ##open hardware
2  openhw D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
3  ## Set SDK workspace
4  setws D:/example/example.sdk
5  ## Get design properties
6  get_addr_tag_info -json
   D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
7  get_all_periphs -json
   D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
8  get_design_properties -json
   D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
9  get_bram_blocks
   D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
10 # Connect to a remote hw_server
11 connect
12 # Select a target
13 Target 2
14 # Reset the system before initializing the PS and configuring the FPGA
15 rst
16 # Configure the FPGA.
17 fpga -file
   D:/example/example.sdk/design_1_wrapper_hw_platform_0/design_1_wrapper.bit
18 # Run load Hardware
19 puts "Run load Hardware"
20 loadhw D:/example/example.sdk/design_1_wrapper_hw_platform_0/system.hdf
21 # Source the ps7_init.tcl script and run ps7_init and ps7_post_config commands
22 puts "Source the ps7_init.tcl script and run ps7_init and ps7_post_config commands"
23 source D:/example/example.sdk/design_1_wrapper_hw_platform_0/ps7_init.tcl
24 ps7_init
25 ps7_post_config
26 # Download the application program
27 puts "Download the application program"
28 dow D:/example/example.sdk/sobel_cc/Debug/example.elf
29 # Continue execution until the target is suspended
30 puts "Continue execution until the target is suspended"
31 con -block -timeout 1
```

If the user includes the generated TCL file in the project folder as vivado design suite, it can be run via automation by typing command for "source file_name.tcl" in the XSCT.

5.4 Automation flow of HLS, Vivado, SDK

Figure 5.4 shows that automation of Vivado HLS Design Flow, including SDK. So far, automation using three TCL files is a method that is not inconvenient for a SW engineer to use for HLS as shown Figure 5.2. However, the entire process was automated using Python language.

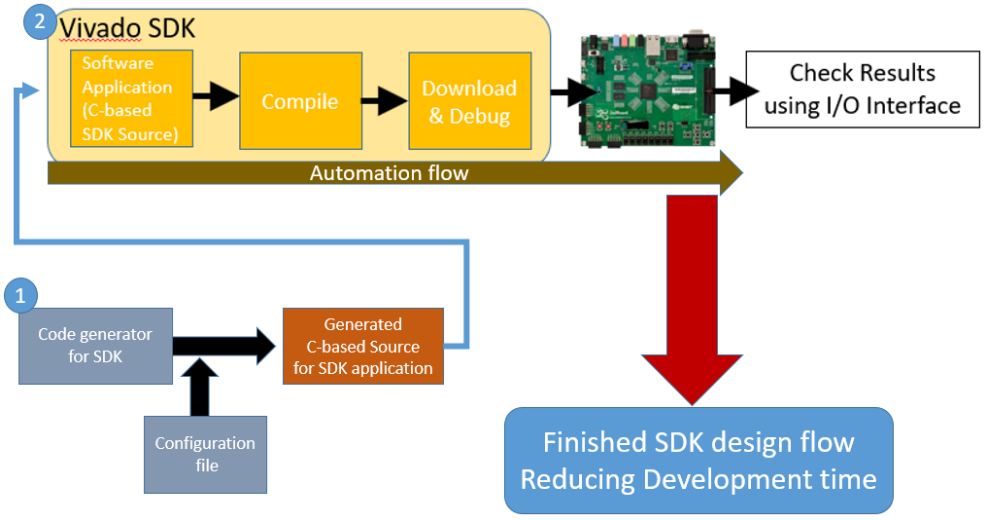


Figure 5.4. Automation of Vivado HLS Design Flow including SDK

Example4. full_automation.py

```
1  ##Move folder to Vivado HLS template
2  import os
3  env = os.environ
4  newpath = r'D:\example;'+env['PATH']
5  env['PATH'] = newpath
6  ##Making Vivado HLS template
7  import subprocess
8  subprocess.call(['HLS_format.exe'],shell=True)
9  ##Add Environment Path for Vivado Design suite
10 import os
11 env = os.environ
12 newpath = r'C:\Xilinx\Vivado\2017.3\bin;'+env['PATH']
13 env['PATH'] = newpath
14 ##Running VIVADO HLS
```

```
15 import subprocess
16 subprocess.call(['vivado_hls','-f','D:/example/run_hls.tcl'], shell=True)
17 ##Move folder to Vivado Design Suite
18 import subprocess
19 subprocess.call(['cd','D:/example'],shell=True)
20 ##Running VIVADO Design Suite
21 import subprocess
22 subprocess.call(['vivado','-mode','batch','-source','vivado_sobel.tcl'],shell=True)
23 ##Move folder to Vivado SDK template
24 import os
25 env = os.environ
26 newpath = r'D:\example;'+env['PATH']
27 env['PATH'] = newpath
28
29 ##Making Vivado HLS template
30 import subprocess
31 subprocess.call(['sdk_format.exe'],shell=True)
32 ##Add Environment Path for Vivado Design suite
33 import os
34 env = os.environ
35 newpath = r'C:\Xilinx\SDK\2017.3\bin;'+env['PATH']
36 env['PATH'] = newpath
37 ##Running VIVADO SDK
38 import subprocess
39 subprocess.call(['xsct.bat','D:/example/sdk_auto2.tcl'], shell=True)
```

For executing this code, after installing Python compiler in the development environment, type command “source full_automation.py” or file name to automatically execute the entire process and check the results. However, this automation process is not perfect because after creating the HLS template, one must put in the algorithm desired.

5.5 Summary

In this chapter, we have explained using a whole automation process via Xilinx HLx design flow. This will help a SW engineer and HLS beginner to implement their own algorithm and easily

utilize SoC FPGA with ARM. In addition, this automation method can shorten processing and development time of Xilinx FPGA.

CHAPTER 6

EXPERIMENTAL RESULTS WITH SIX BENCHMARKS

6.1 Introduction

Six algorithms that are suitable for use on HLS were implemented by template and automation processes to Zedboard. Results were verified using the UART (Universal asynchronous receiver/transmitter) port in the SDK terminal, and the result.txt is printed and saved using the SD card. I compared the resource and execution time of each algorithm. Through this, I analyzed which factors are important when developing algorithms using HLS. Before the experimental result, I explain setup the experimentation.

6.2 Experimental Setup

It is executed by Xilinx tools versions 2017.3. Also, I used Zedboard Zynq-7000 ARM/FPGA Soc Development Board and has Zynq-7000 AP Soc XC7Z020-CLG484 (part number) and dual-core ARM Cortex-A9, 512MB DDR Memory. SD card/UART/Ethernet interface are included. I have executed it on Windows OS. This setup needs two USB ports. One is used to download bitstream and hardware file. Another one is uses UART interface for checking the results on SDK terminal. Also, I used SD card to transfer input/output and golden output data as shown in Figure 6.1.

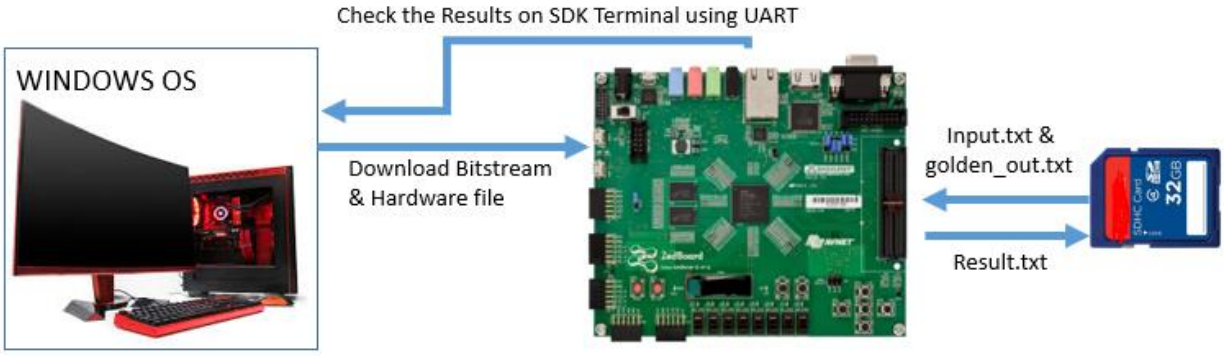


Figure 6.1. Experimental Setup

6.3 Sobel filter

A Sobel filter is a type of edge-detection algorithm used for image processing/computer vision. An image put through this filter is provided in Figure 6.1. Recently, it has been widely used in automobile lane edge detection by utilizing. This is calculated by applying the "Isotropic 3x3 Image Gradient Operator" to the desired image's horizontal and vertical axes respectively [26].

In this analysis, the results were verified using the 512 pixels by 512 pixels of Lena512.bmp files. Result after execution using a Sobel filter, are shown in Figure 6.1.

Figure 6.2. Result from Sobel filter

6.3.1 Resource Utilization

Table 6.1. Resource of Sobel Filter

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	4870	53200	9.15%
Number of LUT-Memory	613	17400	3.52%
Number of LUT-FF pairs	5361	106400	5.04%
Block ram tile	130	140	92.86%
Number of DSP	2	220	0.91%
Number of BUFG	1	32	3.13%

The Sobel filter input a total of 262,144 pixels, which were processed and printed out. Among these, several large variable declarations were made at HLS. As discussed in Chapter 2, this variable declaration consumes memory in final design, resulting in excessive memory use. So, it used around 93% of BRAM in FPGA on Zedboard.

6.3.2 Execution Time

Table 6.2. Execution Time of Sobel Filter

	SW	SW-HW		Difference		Difference (%)	
Execution time	0.4 sec	Unroll	roll	Unroll	roll	Unroll	Roll
		Not possible	0.16 sec	-	0.24 sec	-	150 %

Execution time shows that SW-HW with FPGA-ARM was about 150% faster than SW. HLS supports unroll on loop, which can improve performance. However, unrolling loop was not possible due to a lack of memory and resource in this algorithm. This is because the Sobel filter has lots of iteration in this algorithm in the FPGA.

6.4 KNN

KNN (K-Nearest Neighbors) is one of the most important algorithms and plays a role in the following: text selection, proactive analysis, data mining and image collection [27]. We are not performing classification, instead we are simply implementing the search of k-nearest neighbors when an arbitrary point is presented in a d-dimensional space. In total, the number of rows is 655,36 and number of columns is 8. This algorithm was performed to find eight values that are close to the reference values.

6.4.1 Resource Utilization

Table 6.3. Resource of KNN

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	5100	53200	9.59%
Number of LUT-Memory	721	17400	4.14%
Number of LUT-FF pairs	5631	106400	5.29%
Block ram tile	4.5	140	3.21%
Number of DSP	3	220	1.36%
Number of BUFG	1	32	3.13%

Although the KNN algorithm compared 524,288 integer data, only eight were used for each variable analysis. Therefore, memory usage in variable declaration was lower than Sobel as shown in Table 6.4.

6.4.2 Execution Time

Table 6.4. Execution Time of KNN

	SW	SW-HW		Difference		Difference (%)	
Execution time	0.14 sec	**Unroll**	**Roll**	**Unroll**	**roll**	**Unroll**	**roll**
		0.03 sec	0.06 sec	0.11 sec	0.08 sec	266%	133 %

Execution time shows that SW-HW with FPGA-ARM was about 133% faster with roll on loop than SW. If you can put this pragma directive "#pragma HLS PIPELINE" in top function, this directive tells the compiler to attempt unroll for all loop in the function. After that, when HLS compiler synthesize c-based source with this directive, it can apply unrolled loop if it is possible. However, as one can determine from the Sobel filer result, the unrolled loop may not be applied due to the lack of memory and resources. In such a case, the compiler will explain making an entry in a warning log through the window. If the unrolled loop is done well, the result will be better than the rolled loop.

6.5 AVE8

AVE8 is an algorithm that returns the average by grouping the 24 input integer data into 8 ones in sequence.

6.5.1 Resource Utilization

Table 6.5. Resource of AVE8

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	4837	53200	9.09%
Number of LUT-Memory	762	17400	4.38%
Number of LUT-FF pairs	5489	106400	5.16%
Block ram tile	2	140	1.43%
Number of BUFG	1	32	3.13%

Ave8 processed just 168 data for calculating average 8 data and this is a very simple algorithm using for loop and comparison. So, it used small resource of FPGA.

6.5.2 Execution Time

Table 6.6. Execution Time of AVE8

	SW	SW-HW		Difference		Difference (%)	
Execution time	11.02 msec	Unroll	roll	Unroll	roll	Unroll	Roll
		5.35 msec	11.08 msec	5.67 msec	-0.06 msec	105%	-0.5 %

Hardware acceleration is 0.5% slower than SW because it is a very simple program and AXI interface is used for IP core. In this case, SW processing is more efficient than hardware acceleration. However, After HLS optimization (unrolled loop), hardware accelerations reduce processing time.

6.6 FIR Filter

It is implemented by HLS for fir filter which is a 10- tap FIR filter algorithm [28]. Total number of input data is 50 designed with 8-bit integer and output number of data is 10.

6.6.1 Resource Utilization

Table 6.7. Resources of FIR Filter

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	4666	53200	8.77%
Number of LUT-Memory	640	17400	3.68%
Number of LUT-FF pairs	5458	106400	5.31%
Block ram tile	2.5	140	1.79%
Number of DSP	2	220	0.91%
Number of BUFG	1	32	3.13%

FIR filter processed just 50 data for filtering and this is a very simple algorithm using multiplication. So, very few resources of FPGA were used in FIR filter algorithm.

6.6.2 Execution Time

Table 6.8. Execution Time of FIR Filter

	SW	SW-HW		Difference		Difference (%)	
Execution time	5.03 msec	Unroll	Roll	Unroll	roll	Unroll	roll
		4.5 msec	5.5 msec	0.53 msec	-0.47 msec	11%	-8.5 %

This is also a very simple algorithm for both SW and SW-HW. SW-HW used the AXI interface for IP cores, which execution time is slowed down a little bit with rolled loop. After unrolled loop for optimization, execution time of this algorithm on the hardware acceleration can be reduced by 11% than SW.

6.7 Quick sort

Quick sort is an efficient sorting algorithm. And It is commonly designed with ascending order. If implemented well, it is twice as fast as a heap sort or merge sort.

6.7.1 Resource Utilization

Table 6.9. Resources of Quick Sort

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	5023	53200	9.44%
Number of LUT-Memory	657	17400	3.78%
Number of LUT-FF pairs	5478	106400	5.15%
Block ram tile	4	140	2.86%
Number of BUFG	1	32	3.13%

Quicksort processed just 160 data for sorting and this is a very simple algorithm using for loop and comparison. It used small resources of FPGA for hardware acceleration.

6.7.2 Execution Time

Table 6.10. Execution Time of Quick Sort

	SW	SW-HW		Difference		Difference (%)	
Execution time	29.3 msec	Unroll	roll	Unroll	roll	Unroll	roll
		12.7 msec	13.3 msec	16.64 msec	16.41 msec	130%	123 %

Hardware acceleration for quicksort is faster over 123% than SW. HLS complier synthesized to apply with optimization for quicksort algorithm. After HLS optimization, execution time can be reduced 130% than SW.

6.8 Findprime

Algorithm for finding the prime number between 1 and 1000. A total of 169 Prime numbers were found between 1 and 1000.

6.8.1 Resource Utilization

Table 6.11. Resource of Findprime

Slice Logic Utilization	Used	Available	Utilization (%)
Number of Slice LUTs	4723	53200	8.88%
Number of LUT-Memory	609	17400	3.50%
Number of LUT-FF pairs	5781	106400	5.43%
Block ram tile	4	140	2.86%
Number of BUFG	1	32	3.13%

Findprime algorithm processed just 1169 data for loop and comparison. Small resources were used to implement hardware acceleration of findprime algorithm.

6.8.2 Execution Time

Table 6.12. Execution Time of Findprime

	SW	SW-HW		Difference		Difference (%)	
Execution time	4,612 msec	Unroll	roll	Unroll	roll	Unroll	roll
		5.43 msec	8.68 msec	4606msec	4603 msec	84,835%	53,033 %

Here we can find the great advantage of hardware acceleration. Hardware acceleration is faster over 530 times than SW because findprime algorithm should use nested loop statement to search prime number from 1 to 1000 with comparison. This searching and comparison process always

happens with the worst-case algorithm. So, it should run the nested loop with trip size has 1000. However, hardware acceleration can reduce execution time by optimization.

6.9 Compare benchmarks

6.9.1 Resource Utilization (USED)

Table 6.13. Resource Utilization (USED)

	LUT	LUT-Memory	LUT FF pairs	Block Ram Tile	DSPs	BUFG
Sobel	4870	613	5361	130	2	1
KNN	5100	721	5631	4.5	3	1
AVE8	4837	762	5489	2	0	1
FIR	4666	640	5458	2.5	2	1
Qsort	5023	657	5478	4	0	1
Findprime	4723	609	5781	4	0	1

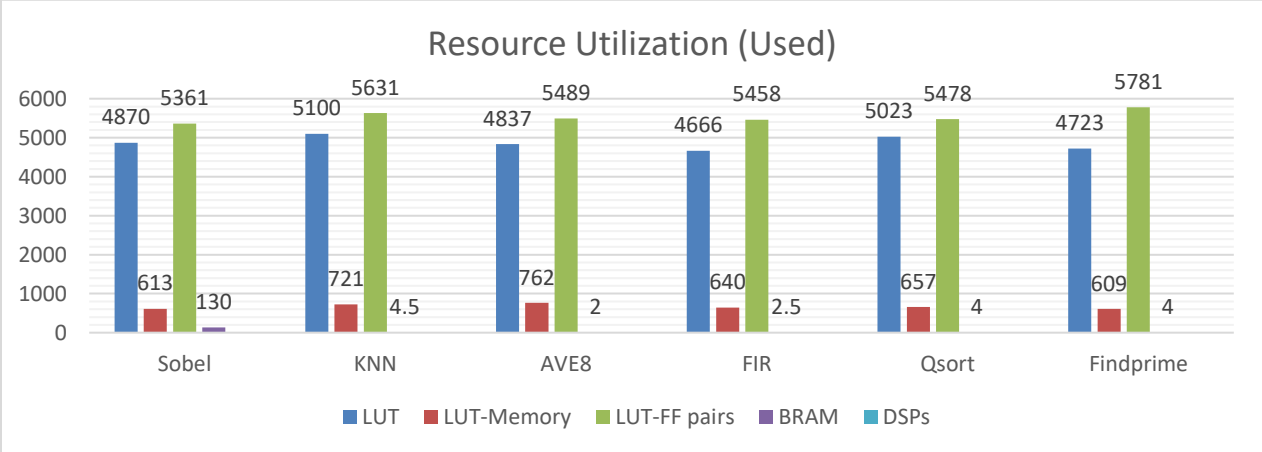


Figure 6.3. Resource Utilization (Used)

All algorithms use small resources of FPGA to implement hardware acceleration. It shows that small FPGA is enough to implement for hardware acceleration.

6.9.2 Resource Utilization (%)

Table 6.14. Resource Utilization (%)

	LUT	LUT-Memory	LUT FF pairs	Block Ram Tile	DSPs	BUFG
Sobel	9.15	3.52	5.04	92.86	0.91	3.13
KNN	9.59	4.14	5.29	3.21	1.36	3.13
AVE8	9.09	4.38	4.38	1.43	0	3.13
FIR	8.77	3.68	5.31	1.79	0.91	3.13
Qsort	9.44	3.78	5.15	2.86	0	3.13
Findprime	8.88	3.5	5.43	2.86	0	3.13

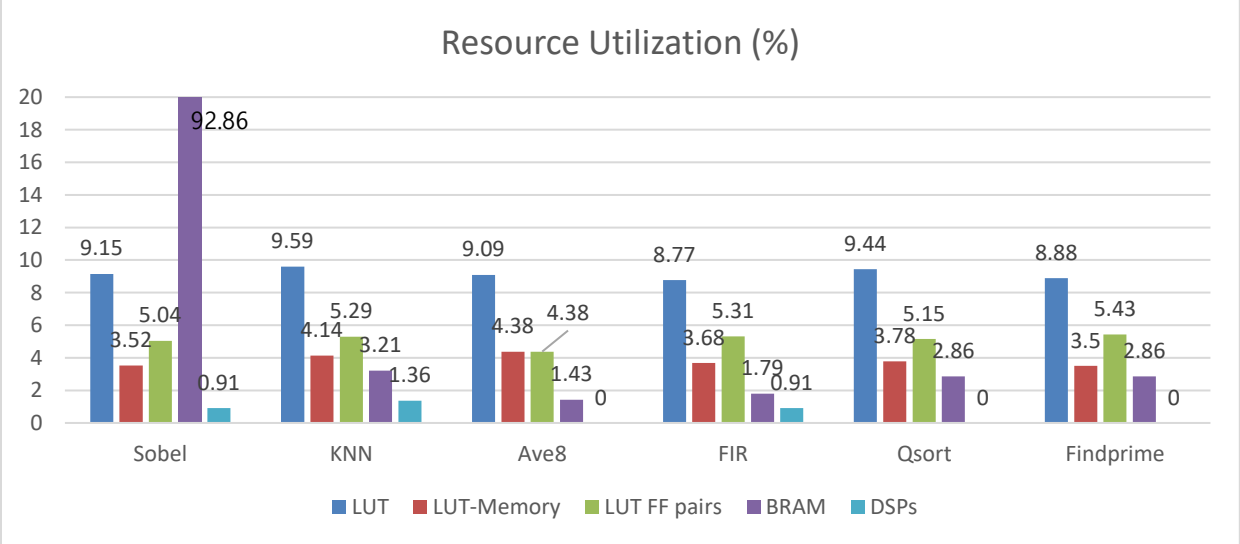


Figure 6.4. Resource Utilization (%)

According to the resource utilization result, all the resources were used in small potion of resource utilization excluding BRAM of Sobel algorithm. As this is already mentioned in 6.2.1 resource utilization, Sobel declared 262,144 array variables in several times. This declaration consumes memory in final design [4]. Therefore, it used around 93% of BRAM in FPGA.

6.9.3 Execution Time

Table 6.15. Execution Time of All Benchmarks

	SW	SW-HW		Difference		Difference (%)	
		unroll	roll	Unroll	roll	unroll	Roll
Sobel	0.4 sec	Not possible	0.16 sec	-	0.24	-	150%
KNN	0.14 sec	0.03 sec	0.06 sec	0.11 sec	0.08 sec	366%	133%
FIR	5.03 msec	4.5 msec	5.5 msec	0.11 msec	-0.47 msec	11%	-8.5%
Qsort	29.3 msec	12.7 msec	13.3 msec	16.64 msec	16.41 msec	130%	123%
AVE8	11.02 msec	5.35 msec	11.08 msec	5.67 msec	-0.06 msec	105%	-0.5%
Findprime	4,612 msec	5.43 msec	8.68 msec	4,606 msec	4,603 msec	84,835%	53,033%

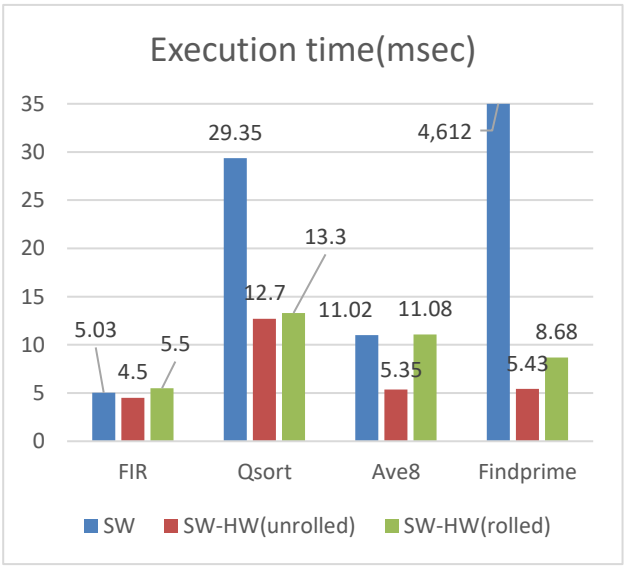


Figure 6.5. Execution time (sec), Execution time (msec)

According to the results of execution time, hardware acceleration has a great advantage in reducing execution time. In particular, processing speed in calculations for processing large amounts of data that is about 1.3 times faster than SW. In addition, with HLS optimization, there was an average speed up of over 1.5 times excluding findprime benchmark.

6.10 Summary

In this chapter, we discussed experimental results using automating FPGA-based hardware acceleration. This method has a great advantage in term of speed up. Hardware acceleration (SW-HW) is faster than SW. The Hardware acceleration (SW-HW) can reduce execution time at least 11% from experimental result. Thus, HLS optimization (unrolled loop) is a very effective way on Hardware acceleration.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, we have proposed and implemented new design methodology for automating FPGA-based hardware acceleration. This method allows SW engineer and FPGA beginner to easily access FPGA-based design. In addition, it can save the time to study about HLS using implemented HLS template and proposed automating design method.

The experiment results using six benchmarks found that the execution time on FPGA-based hardware acceleration was average around 1.5 times faster than the SW. Even for algorithms such as sobel filter and KNN that process large amounts of data (100,000 or more), the results of FPGA-based hardware acceleration are average about 1.3 times faster than the SW.

7.2 Future work

- Developing a fully automated template for Vivado HLS. It is needed to parsing from user desired C-based algorithm.
- Applying this automatic FPGA-based hardware acceleration for synthesizable SystemC language in S2Cbenchmarks.
- Implementing for machine learning algorithms such as CNN (Convolutional Neural Network) with automating FPGA-based hardware acceleration.

APPENDIX

SOURCE CODES

This Appendix includes the C-based source template and testbench template file for HLS using code generator. Also, it contains vivado design suite.tcl file for automation.

A.1 C-based source for HLS

Source file

```
1:  #include <hls_stream.h>
2:  #include <ap_axi_sdata.h>
3:  #define SIZE 100
4:  // Declare 32-bit integer with side-channel (Includes TLAST signal)
5:  typedef ap_axis<32,2,5,6> AXI_VALUE;
6:  unsigned int INPUT[SIZE];
7:  unsigned int OUTPUT[SIZE];
8:  void example(hls::stream<AXI_VALUE> &inStream,hls::stream<AXI_VALUE>
&outStream){
9:  #pragma HLS INTERFACE axis port=outStream
10: #pragma HLS INTERFACE axis port=inStream
11: #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
12: AXI_VALUE aValue;
13: AXI_VALUE bValue;
14: //INTERFACE HLS WITH HARDWARE FOR READING DATA
15: read_A: for(i=0; i< SIZE; i++){
16: #pragma HLS PIPELINE
17:      inStream.read(aValue);
18:     INPUT[i] = aValue.data;
19:  //This is for your algorithm PART!!
20:  //INTERFACE HLS WITH HARDWARE FOR WRITING DATA
21:  write_result: for(i=0; i<SIZE; i++){
22:     #pragma HLS PIPELINE
23:     bValue.data = OUTPUT[i];
24:     bValue.last  = (i==OUTPUT_SIZE-1)? 1 : 0;
25:     bValue.strb  = aValue.strb;
26:     bValue.keep = aValue.keep;
27:     bValue.user = aValue.user;
28:     bValue.id   = aValue.id;
29:     bValue.dest = aValue.dest;
30:     outStream.write(bValue);
31:      }
```

```
32: }
```

A.2 C-based testbench template for HLS

Testbench file

```
1 #include <hls_stream.h>
2  #include <ap_axi_sdata.h>
3  #define SIZE 100
4  // Declare 32-bit integer with side-channel (Includes TLAST signal)
5  typedef ap_axis<32,2,5,6> AXI_VALUE;
6  unsigned int INPUT[SIZE];
7  unsigned int OUTPUT[SIZE];
8  void example(hls::stream<AXI_VALUE> &inStream,hls::stream<AXI_VALUE>
&outStream);
9  int main(){
10  //Define streams for input and output
11   hls::stream<AXI_VALUE> inputStream;
12   hls::stream<AXI_VALUE> outputStream;
13  FILE *fp1, *fp2, *fp3; // fp file pointer
14  int golden_result[SIZE];
15  int result[SIZE];
16  int error=0;
17  AXI_VALUE aValue;
18  //Open input file
19  fp1 = fopen("input.txt","r");
20  if (fp1==NULL){
21      printf("No input file");
22      return NULL;
23      }
24  for (int i=0;i<SIZE; i++){
25      fscanf(fp1, "%d\n", &INPUT[i]);
26      }
27   fclose(fp1);
28  //Open golden_out file
29  fp2 = fopen("golden_out.txt","r");
30  if (fp2==NULL){
31      printf("No Golden output file");
32      return NULL;
33      }
34  for (int i=0;i<SIZE; i++){
35      fscanf(fp2, "%d\n", &golden_result[i]);
36      }
37  fclose(fp2);
```

```
38   //creat result file
39   fp3 = fopen("result.txt","wt");
40   if (fp3==NULL)
41       {
42       printf("No result file");
43       return NULL;
44       }
45 for(i=0; i< SIZE; i++){
46 aValue.data=INPUT[i];
47 inputStream.write(aValue);
48 }
49 // Call top function of IP
50 example(inputStream,outputStream);
51 for(i=0; i< SIZE; i++){
52 outputStream.read(aValue);
53 OUTPUT[i] = aValue.data;
54 }
55 //Compare with result and golden output
56 for (int idxOut = 0; idxOut < SIZE; idxOut++){
57   if ((OUTPUT[idxOut]-golden_result[idxOut])!=0){
58       printf("%d index has error",idxOut);
59       break;
60       error++;
61       }
62   }
63 //write result to result.txt
64 for (int k=0;k<SIZE; k++){
65    fprintf(fp3, "%d\n", OUTPUT[k]);
66    }
67   fclose(fp3);
68   //Checking Error
69   if (error != 0)
70    printf("Mismatch between golden and result\n");
71   else
72      printf("Match Data!!\n");
73   return 0;
74 }
```

A.3 vivado design suite.tcl file for automation

Example3 script.tcl for vivado design suite

```
1   ## START script
2   create_project -force test1 D:/UTD/15.Design/script/test1 -part xc7z020clg484-1
```

```
3   set_property board_part em.avnet.com:zed:part0:1.3 [current_project]
4   set_property target_language VHDL [current_project]
5   set_property source_mgmt_mode None [current_project]
6   # Set the project name
7   set project_name "test1"
8   # Set the directory path for the new project
9   set proj_dir [get_property directory [current_project]]
10 ## Create Blcok design ##
11 create_bd_design "design_1"
12 ## ADD IP Repository ##
13 set_property  ip_repo_paths  D:/UTD/15.Design/Test_sobel_original/sobel_11_22_1
   [current_project]
14 update_ip_catalog
15 ## ADD Zynq7 processing System##
16 create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5 processing_system7_0
17 apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config
   {make_external "FIXED_IO, DDR" apply_board_preset "1" Master "Disable" Slave
   "Disable" }  [get_bd_cells processing_system7_0]
18 ## ADD SOBEL IP Block
19 create_bd_cell -type ip -vlnv xilinx.com:hls:sobel:1.0 sobel_0
20 apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Master
   "/processing_system7_0/M_AXI_GP0" intc_ip "New AXI Interconnect" Clk_xbar "Auto"
   Clk_master "Auto" Clk_slave "Auto" }  [get_bd_intf_pins
   sobel_0/s_axi_CONTROL_BUS]
21 ## SET PROPERTY Zynq7 processing System##
22 set_property -dict [list CONFIG.PCW_USE_S_AXI_HP0 {1}] [get_bd_cells
     processing_system7_0]
23 ## ADD DMA Block
24 create_bd_cell -type ip -vlnv xilinx.com:ip:axi_dma:7.1 axi_dma_0
25 set_property -dict [list CONFIG.c_include_sg {0} CONFIG.c_sg_length_width {23}
     CONFIG.c_sg_include_stscntrl_strm {0} CONFIG.c_m_axis_mm2s_tdata_width {8}
     CONFIG.c_include_mm2s_dre {1} CONFIG.c_include_s2mm_dre {1}] [get_bd_cells
     axi_dma_0]
26 ## CONNECT Instream/OutStream
27 connect_bd_intf_net [get_bd_intf_pins sobel_0/outStream] [get_bd_intf_pins
   axi_dma_0/S_AXIS_S2MM]
28  connect_bd_intf_net [get_bd_intf_pins sobel_0/inStream] [get_bd_intf_pins
   axi_dma_0/M_AXIS_MM2S]
29 ## AUTOMATION OF DMA Block
   apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Master "
   /processing_system7_0/M_AXI_GP0" intc_ip "/ps7_0_axi_periph" Clk_xbar "Auto"
   Clk_master "Auto" Clk_slave "Auto" }  [get_bd_intf_pins axi_dma_0/S_AXI_LITE]
30 apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Master
   "/axi_dma_0/M_AXI_MM2S" intc_ip "Auto" Clk_xbar "Auto" Clk_master "Auto"
```

```
   Clk_slave "Auto" }  [get_bd_intf_pins processing_system7_0/S_AXI_HP0]
31 apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Slave
   "/processing_system7_0/S_AXI_HP0" intc_ip "/axi_smc" Clk_xbar "Auto" Clk_master
   "Auto" Clk_slave "Auto" }  [get_bd_intf_pins axi_dma_0/M_AXI_S2MM]
32 ## ADD concat
   create_bd_cell -type ip -vlnv xilinx.com:ip:xlconcat:2.1 xlconcat_0
   set_property -dict [list CONFIG.NUM_PORTS {3}] [get_bd_cells xlconcat_0]
33 ## SET PROPERTY Zynq7 processing System##
   set_property -dict [list CONFIG.PCW_USE_FABRIC_INTERRUPT {1}
   CONFIG.PCW_IRQ_F2P_INTR {1}] [get_bd_cells processing_system7_0]
34 ## CONNECT INTERRUPT
35 connect_bd_net [get_bd_pins sobel_0/interrupt] [get_bd_pins xlconcat_0/In0]
36 connect_bd_net [get_bd_pins axi_dma_0/mm2s_introut] [get_bd_pins xlconcat_0/In1]
37 connect_bd_net [get_bd_pins axi_dma_0/s2mm_introut] [get_bd_pins xlconcat_0/In2]
38 connect_bd_net [get_bd_pins xlconcat_0/dout] [get_bd_pins
   processing_system7_0/IRQ_F2P]
40 ## VALIDATE BD DESIGN
41 validate_bd_design -force
42 ## MAKE WRAPPER
43 make_wrapper -files [get_files
   D:/UTD/15.Design/script/test1/test1.srcs/sources_1/bd/design_1/design_1.bd] -top
44 add_files -norecurse
   D:/UTD/15.Design/script/test1/test1.srcs/sources_1/bd/design_1/hdl/design_1_wrapper.vhd
45 ## SAVE BLOCK DESIGN
46 save_bd_design
47 ## Set 'sources_1' fileset file properties for local files
48 set file "hdl/design_1_wrapper.vhd"
49 set file_obj [get_files -of_objects [get_filesets sources_1] [list "*$file"]]
50 set_property -name "file_type" -value "VHDL" -objects $file_obj
51 ## Set 'sources_1' fileset properties
52 set obj [get_filesets sources_1]
53 set_property -name "top" -value "design_1_wrapper" -objects $obj
54 ## Create 'constrs_1' fileset (if not found)
55 if {[string equal [get_filesets -quiet constrs_1] ""]} {
   create_fileset -constrset constrs_1
   }
56 # Set 'constrs_1' fileset object
57 set obj [get_filesets constrs_1]
58 ## Empty (no sources present)
59 ## Set 'constrs_1' fileset properties
60 set obj [get_filesets constrs_1]
61 ## Create 'sim_1' fileset (if not found)
62 if {[string equal [get_filesets -quiet sim_1] ""]} {
   create_fileset -simset sim_1
```

```
    }
63 ## Set 'sim_1' fileset object
64 set obj [get_filesets sim_1]
65 ## Empty (no sources present)
66 ## Set 'sim_1' fileset properties
67 ##Set obj [get_filesets sim_1]
68 set_property -name "top" -value "design_1_wrapper" -objects $obj
69 ## Set 'sources_1' fileset properties
70 set obj [get_filesets sources_1]
71 set_property -name "top" -value "design_1_wrapper" -objects $obj
72 ## SYNTHESIS
73 launch_runs synth_1 -jobs 4
74 wait_on_run synth_1
75 ## Place and route
76 launch_runs impl_1 -jobs 4
77 wait_on_run impl_1
78 #Report timing summary (optional)
79 #report_timing_summary -file impl_timing_summary.rpt
80 ## Write Bitstream
81 launch_runs impl_1 -to_step write_bitstream -jobs 4
82 wait_on_run impl_1
83 ## EXPORT HARDWARE WITH BITSTREAM
84 file mkdir D:/UTD/15.Design/script/test1/test1.sdk
    file copy -force D:/UTD/15.Design/script/test1/test1.runs/impl_1/design_1_wrapper.sysdef
85 D:/UTD/15.Design/script/test1/test1.sdk/design_1_wrapper.hdf
86 ## LAUNCH SDK
87 launch_sdk -workspace D:/UTD/15.Design/script/test1/test1.sdk -hwspec
    D:/UTD/15.Design/script/test1/test1.sdk/design_1_wrapper.hdf
88 exit
```

REFERENCES

[1] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie and X. Zhou, "DLAU: A Scalable Deep Learning Accelerator Unit on FPGA," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 3, pp. 513-517, March 2017.

[2] Introduction to FPGA Design with Vivado High-Level Synthesis - UG998, Xilinx, July. [3] 2017.

[3] R. Camposano, L. F. Saunders and R. M. Tabet, "VHDL as input for high-level synthesis," in IEEE Design & Test of Computers, vol. 8, no. 1, pp. 43-49, March 1991.

[4] Vivado Design Suite User Guide High Level Synthesis - UG902, v2017.3 ed., Xilinx, Oct. 2017.

[5] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: an ASIC and system vendor perspective," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, pp. 1507-1522, Dec. 2000.

[6] R. Nane et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp.1591-1604, Oct. 2016.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473-491, April 2011.

[8] Bringing Ultra High Productivity to Mainstream Systems & Platform Designers - "https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf"

[9] Zynq-7000 All Programmable SoC Technical Reference Manual - UG585, Xilinx, Dec. 2017.

[10] P. Coussy, D. D. Gajski, M. Meredith and A. Takach, "An Introduction to High-Level Synthesis," in IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17, July-Aug. 2009.

[11] S. Sánchez-Solano, M. Brox, E. del Toro, P. Brox and I. Baturone, "Model-Based Design Methodology for Rapid Development of Fuzzy Controllers on FPGAs," in IEEE Transactions on Industrial Informatics, vol. 9, no. 3, pp. 1361-1370, Aug. 2013.

[12] M. Fingeroff, High-level synthesis blue book. Xilbris Corporation, 2010

[13] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," Tree-based Heterogeneous FPGA Architectures, pp. 7-48, 2012.

[14] ″Zedboard.” [Online]. Available http://zedboard.org/product/zedboard

[15] T. Scott, Legacy FPGA Design scan be Migrated to Achieve Better Performance, [online] available http://www.latticesemi.com

[16] Altera, "Architecture brief- what is a soc fpga?" Tech. Rep., 2014.

[17] P. H. W. Leong, "Recent Trends in FPGA Architectures and Applications," 4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008), Hong Kong, 2008, pp. 137-141.

[18] "Zynq-7000 generation ahead backgrounder," Xilinx, Tech. Rep., 2014.

[19] AXI Reference Guide - UG761, v2013.1, Xilinx, Mar. 2011

[20] Vivado Design Suite User Guide Design Flows Overviews - UG892, v2017.2 ed., Xilinx, June. 2017.

[21] A. Kumar, A. Hansson, J. Huisken and H. Corporaal, "An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems on Chip," 2007 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2007, pp. 1-6.

[22] Deming Chen, Jason Cong, and Peichen Pan, “FPGA Design Automation A survey,” Foundations and Trends in Electronic Design Automation, Vol.1, No. 3, November 2006.

[23] R. Ernst, "Codesign of embedded systems: status and trends," in IEEE Design & Test of Computers, vol. 15, no. 2, pp. 45-54, April-June 1998.

[24] H. Ding, S. Ma, M. Huang and D. Andrews, "OOGen: An Automated Generation Tool for Custom MPSoC Architectures Based on Object-Oriented Programming Methods," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 233-240.

[25] Ousterhout, J, “TCL: An Embeddable Control Language”, USENIX Conference, 1990. pp. 133-146

[26] Li Xue, Zhao Rongchun and Wang Qing, "FPGA based Sobel algorithm as vehicle edge detector in VCAS," International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003, Nanjing, 2003, pp. 1139-1142 Vol.2.

[27] "Y. Pu, J. Peng, L. Huang, and J. Chen, “An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl,” in Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on , May 2015, pp. 167–170"

[28] B. Carrion Schafer and A. Mahapatra, "S2cbench: Synthesizable systemc benchmark suite for high-level synthesis," Embedded Systems Letters, IEEE, vol. 6, no. 3, pp. 53-56, Sept 2014

RBIOGRAPHICAL SKETCH

Songseok Choi was born in Busan, South Korea on 7th Oct, 1980. He finished his high school in 1999 from Yongin High School, Busan in 2004. After that, he completed his undergraduate degree (BS) in Computer Science and Electrical Engineering from Handong Global University in 2008. He worked as a FPGA Engineer at LIGNEX1 after completing his bachelors. He joined The University of Texas at Dallas for his Master of Science in Electrical Engineering in August 2016. He has been working with DARClab (Design Automation and Reconfigurable Computing lab) since August 2017.

CIRRICULUM VITAE

Songseok Choi

November 27th, 2018

Contact Information:

Department of Electrical Engineering
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: songseok.choi@utdallas.edu

Educational History:

M.S.E.E, The University of Texas at Dallas, 2018
MSEE Thesis: Automating FPGA-Based Hardware Acceleration
Thesis Advisor: Dr. Benjamin Carrion-Schaefer

B.Science, Computer Science and Electrical Engineering, Handong Global University, South Korea, 2008

Work Experience:

FPGA Engineer, LIGNEX1, South Korea (May'08 – May'13)