TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Aleksei Obuhov 178198IASM

MULTI-CORE ARCHITECTURES WITH HARDWARE ACCELERATORS FOR PARALLEL DATA PROCESSING

Master's thesis

Supervisor: Aleksander Sudnitsõn
PhD

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksei Obuhov 178198IASM

MITME-TUUMALISED ARHITEKTUURID RIISTVARAKIIRENDAJAGA PARALLEELSEKS ANDMETÖÖTLEMISEKS

Magistritöö

Juhendaja: Aleksander Sudnitsõn
PhD

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleksei Obuhov

04.05.2020

Abstract

Data sorting is a common operation in various modern applications. Most filtering and searching operation utilize data sorting. The technology development brought the necessity of fast sorting algorithms. The recently released Multiprocessor System-on-Chip, such as UltraZed-EG, provide to possibilities to design a data sorting embedded systems with high efficiency and lower power consumption.

This thesis provides hardware implementation of an O(1) Time Complexity Two-Step Sorting Network on development board UltraZed-EG. The UltraZed-EG is based on Xilinx Zynq UltraScale+ Multiprocessor System-on-Chip, which are available in the Department of Computer Engineering in Tallinn University of Technology.

Through the process of implementation, the proposed sorting network was thoroughly scrutinized. To achieve the desired goals, the Advanced eXstensible Interface and AXI4-Steram protocol were examined, IP core for the Two-Step Sorting Network was created, the communication between processing logic and implemented sorting network was established. The Two-Step Sorting Network IP Core was written in VHDL hardware description language. For testing purposes, the standalone application and Petalinux application were additionally written in C programming language.

According to the experiment results, the proposed implementation works properly and has better optimization of resource consumption regarding to the results in [1]. However, the experiment results demonstrate the big growth of resource utilization by incrimination of a data set size, which makes the use of this implementation impractical with large data sets.

The implemented Two-Step Sorting Network IP core has multi-module structure, where each module can be separately reused and improved in future works.

This thesis is written in English and is 38 pages long, including 6 chapters, 27 figures and 2 tables. The source code of the proposed design available at [2].

Annotatsioon

Mitme-tuumalised arhitektuurid riistvarakiirendajaga paralleelseks andmetöötlemiseks

Andmete sorteerimine on tavaline toiming erinevate kaasaegsete rakendustes. Enamik filtreerimis- ja otsimistoimingutest kasutatakse andmete sorteerimist. Tehnoloogia areng tõi kaasa kiirte sorteerimisalgoritmide vajadusele. Hiljuti välja antud MPSoC, näiteks UltraZed-EG, pakub suure tõhususega ja väiksema energiatarbega andmesorteerimise manussüsteemide kujundamiseks võimalusi.

See lõputöö pakub O(1) ajaliselt keeruka kaheastmelise sorteerimisvõrgu riistvaralist rakendamist arendusplaadil UltraZed-EG. Tallinna Tehnikaülikooli Arvutisüsteemide instituudi UltraZed-EG arendusplaat, mis põhineb Xilinx Zynq UltraScale+ mitmeprotsessorilisel süsteemil.

Pakutud sorteerimisvõrk oli põhjalikult uuritanud arendamise käiguses. Selgitatakse Advanced eXstensible Interface ja AXI4-Stream protokoll; arendatakse kaheastmelise sorteerimisvõrgu PL tasemel IP-tuma kujul; realiseeritakse side PS ja arendatud sorteerimisvõrgu vahel. Kaheastmelise sorteerimisvõrgu IP-tuma on kirjutatud VHDL riistvarakirjelduskeeles. Standalone ja Petalinux rakendused on kirjutatud C programmeerimiskeeles testimise eesmärgil.

Eksperimendite tulemuste alusel, arendatud kaheastmelise sorteerimisvõrk töötab korralikult. Võrreldes [1] tulemusega ressursi tarbimine on optimeeritud paremaks. Eksperimendite tulemused näitavad ressursi tarbimise suurt kasvu andmete hulka suurendamise tänu, mis muudab selle rakendamise kasutamise ebapraktiliseks.

Arendatud kaheastmelise sorteerimisvõrgu IP-tuum on mitme mooduli struktuur, kus iga moodulit saab tulevastes töödes eraldi uuesti kasutada ja täiustada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 38 leheküljel, 6 peatükki, 27 joonist, 2 tabelit. Lähtekood on saadaval [2].

List of abbreviations and terms

AXI	Advanced eXtensible Interface
BSP	Board Support Packages
C/S	Comparator/Swapper
CLB	Configurable Logic Block
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GPU	Graphics Processing Unit
HP	High Performance
ILA	Integrated Logic Analyzer
IP	Intellectual Property
LUT	LookUp Table
MPSoC	Multiprocessor System-on-Chip
OS	Operating System
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
PSoC	Programmable System-on-Chip
RTL	Register Transfer Level
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
UART	Universal Asynchronous Receiver-Transmitter
VHDL	Very high speed integrated circuits Hardware Description Language

Table of contents

1 Introduction 10
1.1 Motivation 10
1.2 Scope 10
1.3 Thesis Outline 11
2 Background 12
2.1 Sorting networks 12
2.2 An O(1) Time Complexity Two-Step Sorting Network 13
2.3 Design environment and hardware 16
3 System design solution for Two-Step Sorting Network 17
3.1 AXI Direct Memory Access IP core 18
3.2 AXI-Stream protocol 19
4 Two-Step Sorting Network IP Core implementation 21
4.1 AXI4-Stream receiver implementation 22
4.2 AXI4-Stream sender implementation 23
4.3 Two-Step Sorting Network implementation 25
5 Experiment results and analysis 29
5.1 Post-implementation results and resources utilization 29
5.2 Two-Step Sorting Network monitoring with Integrated Logic Analyzer 31
5.3 Two-Step Sorting Network execution result at Petalinux 32
6 Conclusion and Future work 34
References 35
Appendix 1 – AXI Direct Memory Access IP Core customization page 36
Appendix 2 – Configuration and building of Petalinux boot image 37

List of figures

Figure 1.1 Simplified architecture of system. 11
Figure 2.1 (a) A comparator/swapper block. (b) in Knuth notation. 12
Figure 2.2 Odd-even transposition sorting network. 13
Figure 2.3 Structure of the O(1) Time Complexity Two-Step Sorting Network. 14
Figure 2.4 Two-step sorting network elements comparison depicted as graph 14
Figure 3.1 Final architecture of system with Two-Step Sorting Network IP Core. 17
Figure 3.2 The final block design diagram in Vivado. 18
Figure 3.3 Block diagram of AXI DMA connections. 18
Figure 3.4 AXI4-Stream interface interconnection between master and slave IP cores. 19
Figure 3.5 Example of AXI4-Stream protocol transmitting process. 20
Figure 4.1 Two-Step Sorting Network IP core structure. 21
Figure 4.2 FSM of AXI4-Stream receiver. 22
Figure 4.3 Writing TDATA payload to FIFO data input buffer 23
Figure 4.4 FSM of AXI4-Stream sender. 23
Figure 4.5 Reading data from data output buffer. 24
Figure 4.6 Structure of implemented Two-Step Sorting Network. 25
Figure 4.7 Optimization of the Edge Matrix. 25
Figure 4.8 Hardware representation of the Edge Computer for 4-input configuration. . 26
Figure 4.9 Reading of the Binary Vector representation in an upper triangular matrix form. 27
Figure 4.10 Sorting Network internal entities in Vivado. 28
Figure 4.11 Simulation of implemented Two-Step Sorting Network. 28
Figure 5.1 The result of utilization LUTs with different data widths 29
Figure 5.2 The result of utilization LUTs with different sizes of data set 30
Figure 5.3 Integrated Logic Analyzer connections. 31
Figure 5.4 Sorting network IP core internal signal monitoring. 32
Figure 5.5 Transferring set of 8 test data by “axidmatransfer” application. 33
Figure 5.6 Validation of the result of sorting on hardware. 33

List of tables

Table 1 The result of utilization LUTs with different data widths. 30
Table 2 The result of utilization LUTs with different sizes of data set. 30

1 Introduction

This thesis explores hardware implementation of an O(1) Time Complexity Two-Step Sorting Network on a field-programmable gate arrays (FPGA). It also addresses investigation of resource consuming of proposed sorting method.

1.1 Motivation

Data sorting is a common operation in data processing area, such as image and video processing, network communications and digital signal processing. Most filtering and searching operation utilize data sorting.

Many data sorting methods require parallel data processing and high repetition of operations. In case of large amounts of data, the execution of such sorting methods on a CPU leads to consuming of a large amount of CPU resources. Sorting networks on a hardware accelerators are intended to solve this problem. Implementation such of system can be achieved on FPGA and programmable systems on chip (PSoC), because of their flexibility, durability and availability. Those platforms allow us to design a data sorting embedded systems with high efficiency and lower power consumption.

1.2 Scope

The current thesis is focused on the hardware implementation of an O(1) Time Complexity Two-Step Sorting Network on UltraZed-EG Starter Kit that is based on the Xilinx Zynq UltraScale+ MPSoC. The proposed sorting network is designed, analysed in this work.

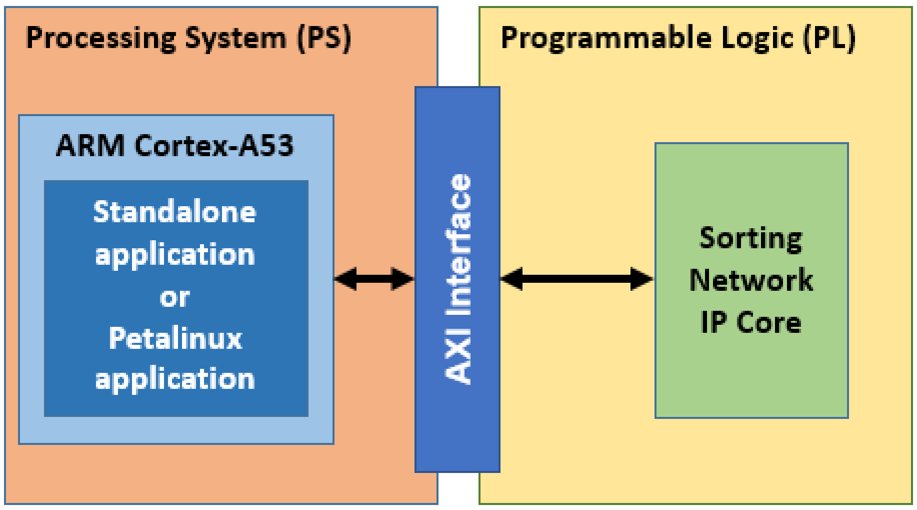


Figure 1.1 Simplified architecture of system.

The main idea is to implement scalable the sorting network as Intellectual Property (IP) core in Xilinx Zynq Programmable Logic (PL) that communicates with Processing System (PS) through AXI interface to find limitations of proposed sorting network. The simplified system architecture view is shown in Figure 1.1. To generate data for sorting, a standalone application or Petalinux application is used. [3]

1.3 Thesis Outline

The remaining part of the thesis contains 5 chapters.

Chapter 2 introduces the background information on sorting networks, an O(1) Time Complexity Two-Step Sorting Network and tools that were used in current thesis.

Chapter 3 presents the proposed system design solution for Two-Step Sorting Network. It also describes the AXI Direct Memory Access IP core and AXI4-Stream protocol.

Chapter 4 presents the implementation of Two-Step Sorting Network IP Core. It also overviews internal components of the proposed implementation and explains their work.

Chapter 5 provides a method of doing experiment, shows and analyse the experimental results of the proposed implementation.

Chapter 6 concludes the thesis and discusses the directions for the further work.

2 Background

2.1 Sorting networks

The implementation of parallel sorting method is a common problem in data processing field. This problem has many solutions such as Parallel QuickSort, Parallel Radix Sort, Sample Sort and algorithmic methods are based on sorting networks. The latter presents a great interest for hardware acceleration, because such as algorithms requires high repetition of simple operations which is can be optimized at hardware level.

Simple sorting network is comprised of wires and comparators/swappers (C/S). A comparator/swapper, shown in Figure 2.1 (a), is a block with two inputs, A and B, and two outputs, the top output is result of function max(A,B) and the bottom output is result of function min(A,B). For simplification, uses Knuth notation in Figure 2.1 (b). [4]

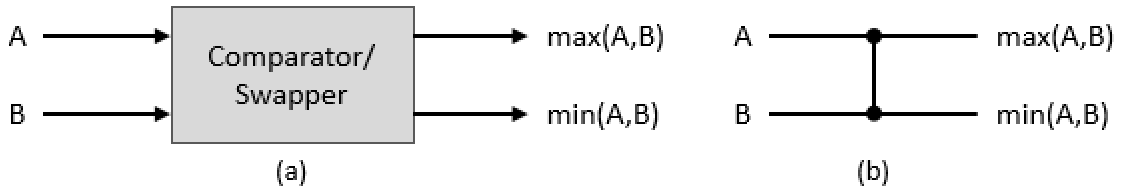


Figure 2.1 (a) A comparator/swapper block. (b) in Knuth notation.

The sorting network represents a pipeline operation, where an unsorted data propagates through the wires and C/S from left to right to produce the sorted data vector on the outputs. One of the most simplest examples of sorting network is Odd-Even Transposition sorting network depicted in Figure 2.2. [4]

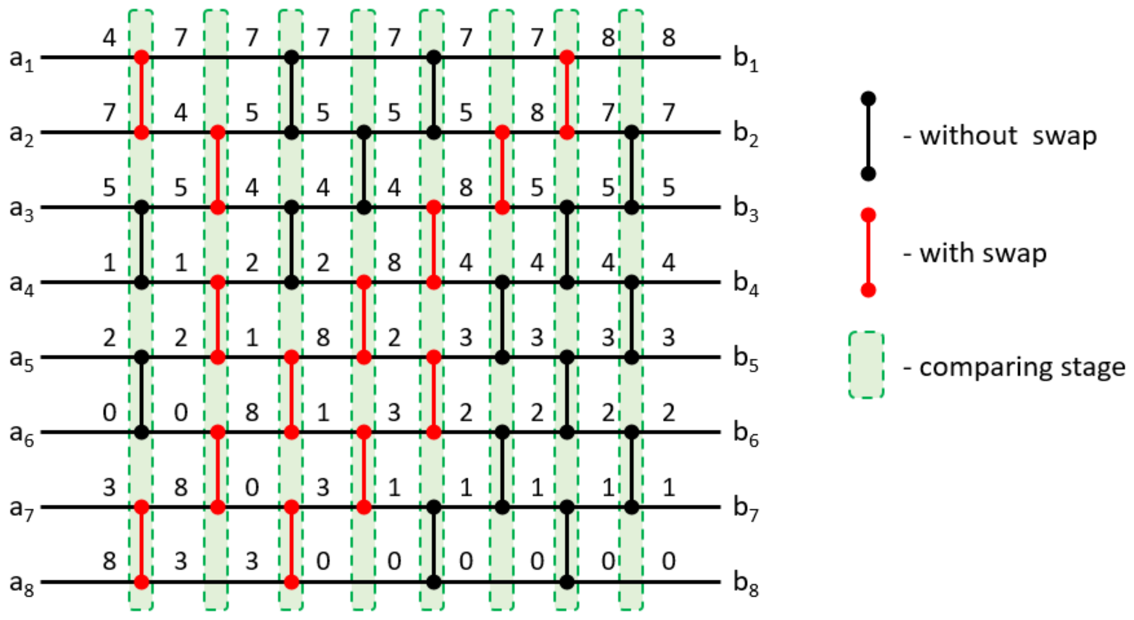


Figure 2.2 Odd-even transposition sorting network.

Odd-even transposition sorting network for n input data consists of n comparator stages. The number of comparators for this sorting network can be calculated as:

$$number\ of\ comparators/swappers = n\frac{n-1}{2} \tag{1}$$

Number of clock cycles required of sorting n inputs by this sorting network equals number of inputs. All aforementioned factors taken into account, 8 clocks and 28 C/S are required to sort 8 input data. This type of sorting network is less efficient than other types of sorting networks, but is considered more reliable and simpler. [4]

2.2 An O(1) Time Complexity Two-Step Sorting Network

An O(1) Time Complexity Two-Step Sorting Network was introduced in [1], which is based on the graph theory concept, and in contrary to conventional sorting networks it consists of edges instead of C/S. Furthermore, its sorting time is independent from the number of inputs, therefore for n data inputs requires an $O(1)$ clock cycles. Structure of proposed sorting network consists of three main blocks as shown in Figure 2.3. It is including Edge Computer, Rank Computer and Data Router.

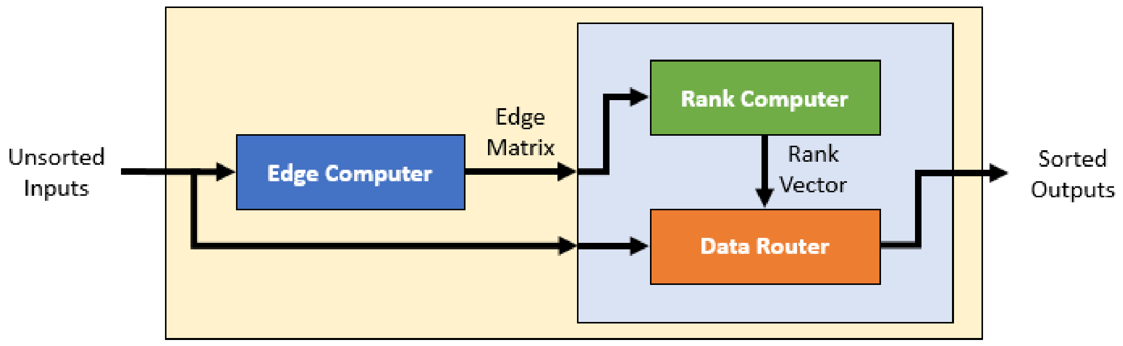


Figure 2.3 Structure of the O(1) Time Complexity Two-Step Sorting Network.

The unsorted inputs are directly connected to the Edge Computer and the Data Router. The Edge Computer block consists of comparators, by which all inputs are compared against each other, that can be depicted as graph for four-input configuration in Figure 2.4.

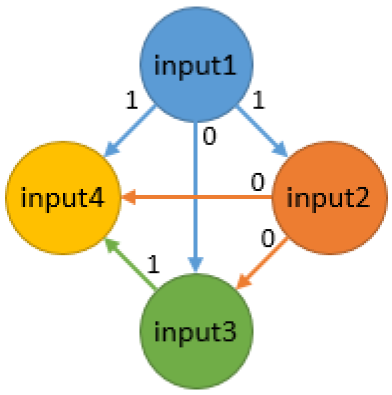


Figure 2.4 Two-step sorting network elements comparison depicted as graph.

The total number of comparators for n-input configuration can be calculated as:

$$number\ of\ comparators = n\frac{n-1}{2} \qquad (2)$$

The Edge Matrix is formed in the output of the Edge Computer according to results of comparisons. The Rank Computer block produces Rank Vector by calculating sum of the elements of each row in the Edge Matrix. The Data Router block directs the unsorted inputs to their respective sorted positions at the output, according to information from the Rank Vector. The Edge Computer finishes processing in a single clock pulse, while the next two other blocks also require another clock pulse for finish processing. Consequently, this sorting network requires only two clock cycle and the processing time does not depend on the amount of input data. [1]

The Rank Computer can be described mathematical expression of two matrices multiplication (3), where R denotes the Rank Vector, E represents the Edge Matrix and I is the Identity Matrix.

$$R = E \times I \tag{3}$$

Each element of matrix E, that is $E(i,j)$, represents the result of comparing the $input(i)$ and the $input(j)$. In the case when the $input(i)$ is greater than or equal to the $input(j)$, "1" is written into the matrix at $E(i,j)$, otherwise "0". [1]

The $E(i,j)$ values extracts out of the elements comparison graph, where considering to the next condition: when the outgoing edge value is "1" or the incoming one is "0", the $E(i,j)$ will be "1", otherwise it will be "0". The diagonal of the E matrix will be filled with zeros, since there is no need to compare the inputs with itself. With respect to the graph depicted in Figure 2.4, matrix E is extracted as follows:

$$E = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{4}$$

According to (3) and (4) the Rank Vector for this inputs can be calculated as follow:

$$R = E \times I = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 3 \\ 1 \end{bmatrix} \tag{5}$$

Each element of the rank vector represents the position index in sorted order for correspondent input. [1]

2.3 Design environment and hardware

The project is implemented on UltraZed-EG Starter Kit in Xilinx Vivado Design Suite environment.

The UltraZed-EG Starter Kit is based on Xilinx Zynq UltraScale+ XCZU3EG-1SFVA625 MPSoC with four ARM Cortex A53 cores, two Cortex R5 cores, a Mali-400 GPU, and FPGA fabric with 154K system logic cells and 71K CLB LUTs. The system memory of this system is 2GB DDR4 SDRAM. [5]

3 System design solution for Two-Step Sorting Network

The main idea is to design and implement scalable hardware design solution of an O(1) Time Complexity Two-Step Sorting Network on Xilinx Zynq UltraScale+ XCZU3EG-1SFVA625 MPSoC such way to be able easily reconfigure system for different sorting configurations such as number of sorting data or data-unit width.

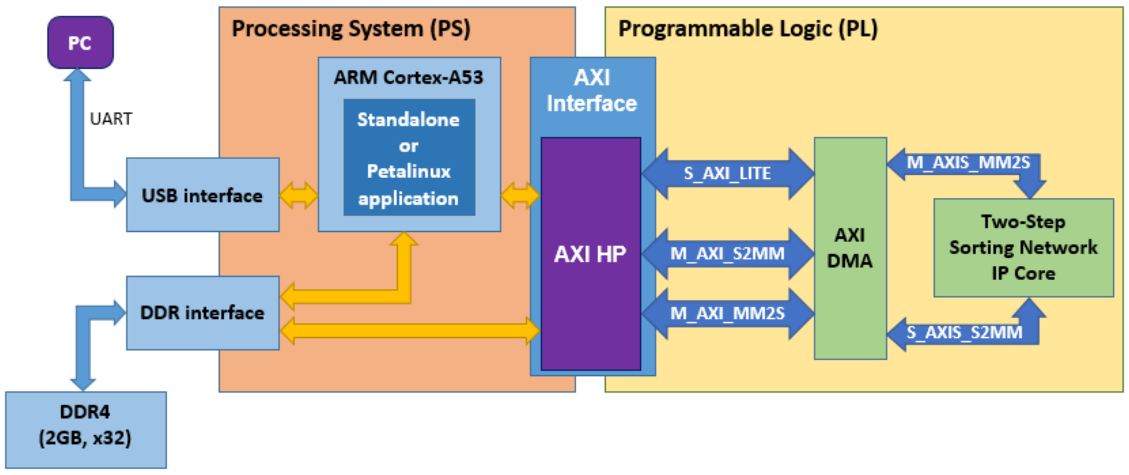


Figure 3.1 Final architecture of system with Two-Step Sorting Network IP Core.

Figure 3.1 presents the proposed architecture of system is based on Xilinx Zynq UltraScale+ MPSoC with using Two-Step Sorting Network IP Core as hardware accelerator. We assume that the unsorted data generated by standalone or Linux application are transferred through AXI HP (High Perfomance) port toward Direct Memory Access (DMA) channel, where AXI DMA IP core stream this data to the Two-Step Sorting Network IP Core through AXI4-Stream port. When Two-Step Sorting Network IP Core has completed processing, it sends the sorted data back to the AXI DMA, where the AXI DMA will transferred the sorted data back to the application. Personal Computer (PC) communicates with PS side through UART interface and allows us to control Petalinux or launch and debug Standalone applicaion.

The final block design diagram with only interface connections is shown in Figure 3.2. The AXI DMA is described in detail in Section 3.1. The implementation of Two-Step Sorting Network IP Core is described in Chapter 4.

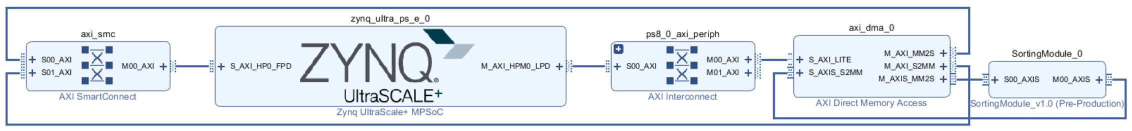


Figure 3.2 The final block design diagram in Vivado.

3.1 AXI Direct Memory Access IP core

The AXI Direct Memory Access IP core provides high-performance burst transfers between Processing System DRAM and the Programmable Logic.

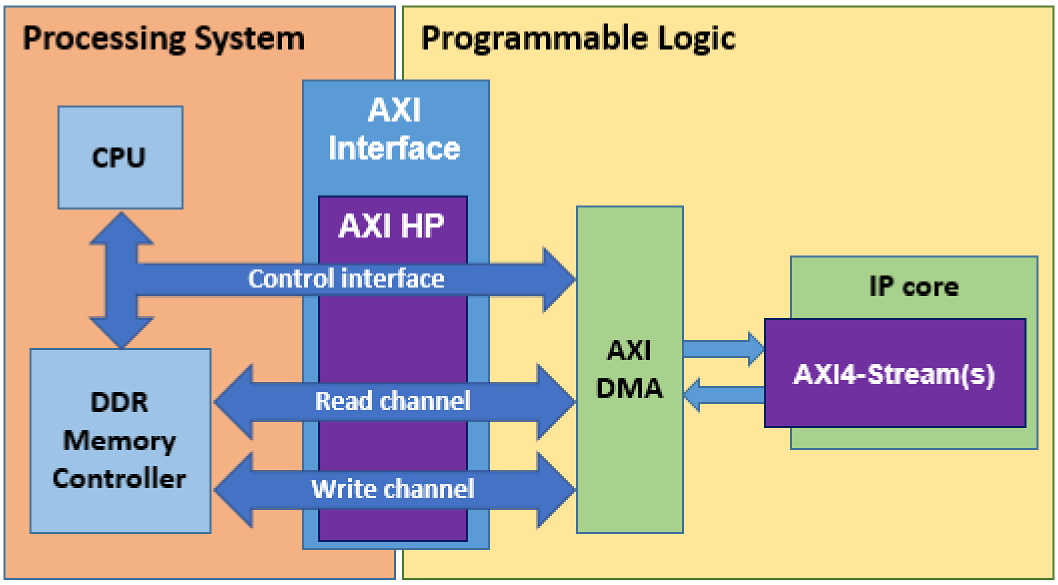


Figure 3.3 Block diagram of AXI DMA connections.

The AXI DMA has a AXI-Lite control interface, and a read and write channel which provides access to the memory location, and a AXI4-Stream port for connecting to an IP Core. The read channel reads from PS DRAM, and writes to a stream. The write channel performs the opposite operation: reads from a stream and writes to PS DRAM. In case of Xilinx Zynq UltraScale+ MPSoC the AXI DMA performs all commutation with PS side through AXI HP port as depicted in Figure 3.3. [6] [7] [8]

The Xilinx Vivado Design Suite environment allows us to configure the AXI DMA IP Core. The customization page is shown in Appendix 1 – AXI Direct Memory Access IP Core customization page. The Stream Data Width parameter will be changed during experiments. [6]

3.2 AXI-Stream protocol

Advanced eXstensible Interface (AXI) is the standard for communication between Xilinx IP and it form the interface between the PS Processing System and Programmable Logic in the Xilinx Zynq UltraScale+ MPSoC. For transmitting an arrays of data between PS and Two-Step Sorting Network selected AXI4-Stream protocol. This type of AXI protocol interface enables high-speed transmission of big volumes of data. AXI4-Stream protocol can burst an unlimited amount of data and designed to transport arbitrary unidirectional data streams. [9] [10]

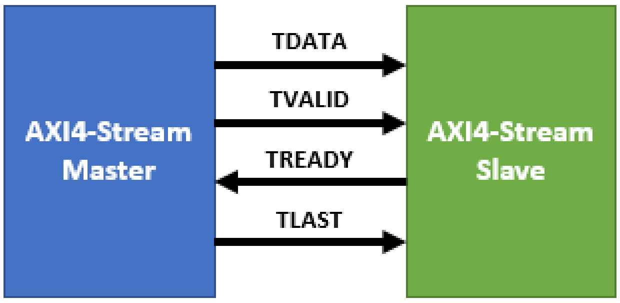


Figure 3.4 AXI4-Stream interface interconnection between master and slave IP cores.

Figure 3.4 depicts the AXI4-Stream interface interconnection between master and slave IP cores, which consists of four signals. The TDATA signal is a payload transferred per clock cycle, which the width can be defined in the range from 8 to 1024 bits. The TVALID signal indicates that the master is driving a valid transfer. The TREADY signal indicates that the slave can accept a transfer in the current cycle. A transfer takes place when both TVALID and TREADY are asserted. The TLAST signal indicates the boundary of a packet. [11]

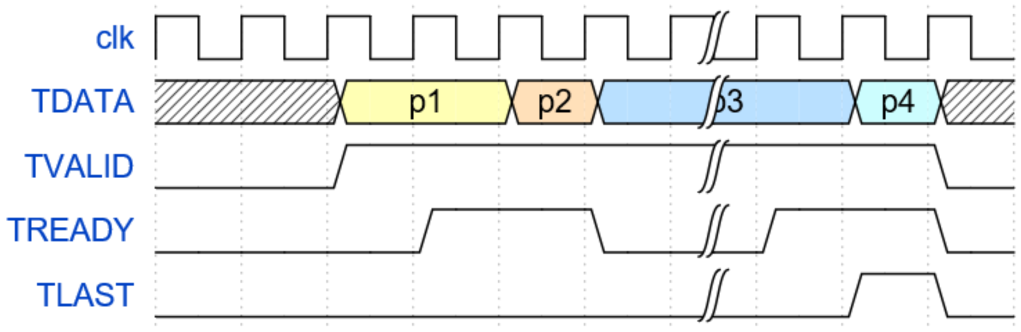


Figure 3.5 Example of AXI4-Stream protocol transmitting process.

Figure 3.5 depicts the example of AXI4-Stream protocol transmitting process, where master performs sending four data parts by one packet. Handshake is executed as shown in Figure 3.4 at sending "p1" payload: the master sets high TVALID signal and on the next clock cycle the slave sets high TREADY signal as well. At this moment “p1” payload is considering as read. After receiving “p2” payload, the slave cannot receive one more payload and sets low TREADY. When the slave is ready to resume reception, it sets high TREADY. At “p4” payload, the master sets high TLAST to indicate the boundary of the packet and to finish the packet transmitting process. [12]

4 Two-Step Sorting Network IP Core implementation

The implementation of the proposed sorting network IP Core consists of five main entities as shown in Figure 4.1. This IP Core is implemented to be compatible with AXI4-Stream protocol and to be configurable at design. The AXI4-Stream receiver enables serial receiving an array of unsorted data from AXI DMA through Slave AXI4-Stream Port and collect them into the data input FIFO buffer. The AXI4-Stream sender enables serial sending an array of sorted data from Data output buffer to AXI DMA through Master AXI4-Stream Port.

Assuming that sorting is done in two clock cycles, the sending procedure has an internal delay before sending. After receiving a signal about the arrival of new data, the AXI4-Stream receiver send the control signal to the AXI4-Stream sender, it takes one clock cycle, then AXI4-Stream waits one more clock cycle and initialize sending an array of sorted data. The receiving of new data packet is stopped until the current packet finish was processed and sent back to AXI DMA. The Two-Step Sorting Network reads and writes whole array of data at ones.

The implementation of each entity is described in detail in the following sections.

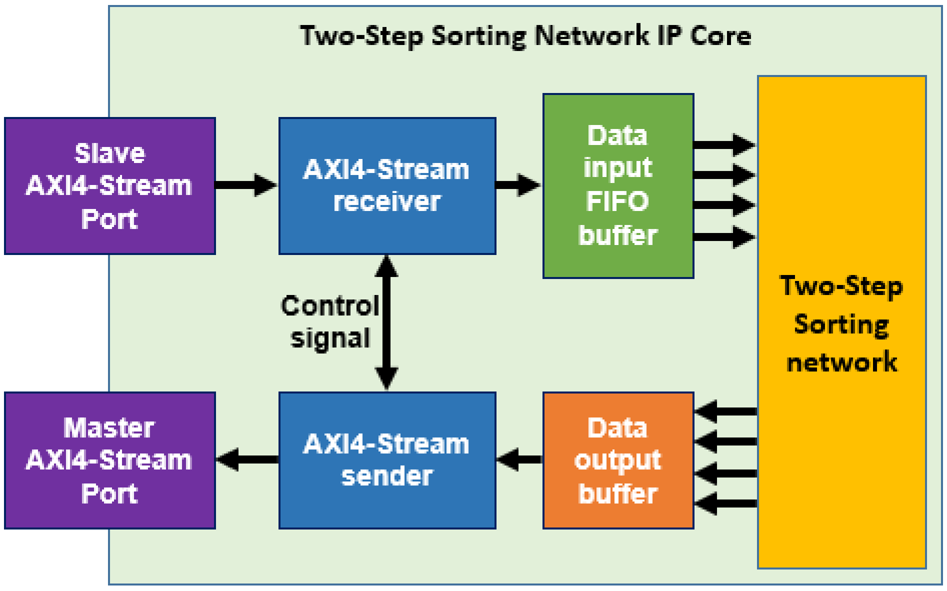


Figure 4.1 Two-Step Sorting Network IP core structure.

4.1 AXI4-Stream receiver implementation

The implementation of AXI4-Stream receiver was executed to be compliant with AXI4-Stream protocol as slave. The receiving procedure controlled by internal finite state machine (FSM) as shown in Figure 4.2. [13]

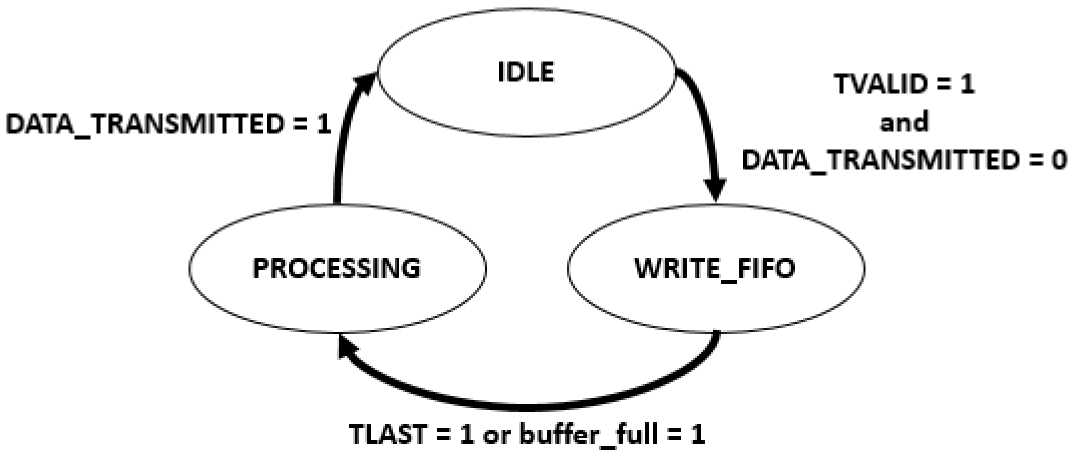


Figure 4.2 FSM of AXI4-Stream receiver.

The FSM of AXI4-Stream receiver consists of three states: “IDLE”, “WRITE_FIFO” and “PROCESSING”. At the “IDLE” state the receiver is waiting for the start of a new packet transmission from the AXI DMA, which corresponds to a rising edge TVALID signal, and low DATA_TRANSMITTED signal on the AXI4-Stream sender, which is described in detail in the section 4.2. Moreover, during this state the receiver sends high reset signal to the data input FIFO buffer. When conditions of receiving a new packet are met, the receiver sets high TREADY signal and moves to “WRITE_FIFO” state. At “WRITE_FIFO” state, the receiver is enabling writing in the data input FIFO buffer by every clock cycle from TDATA as shown in Figure 4.3.

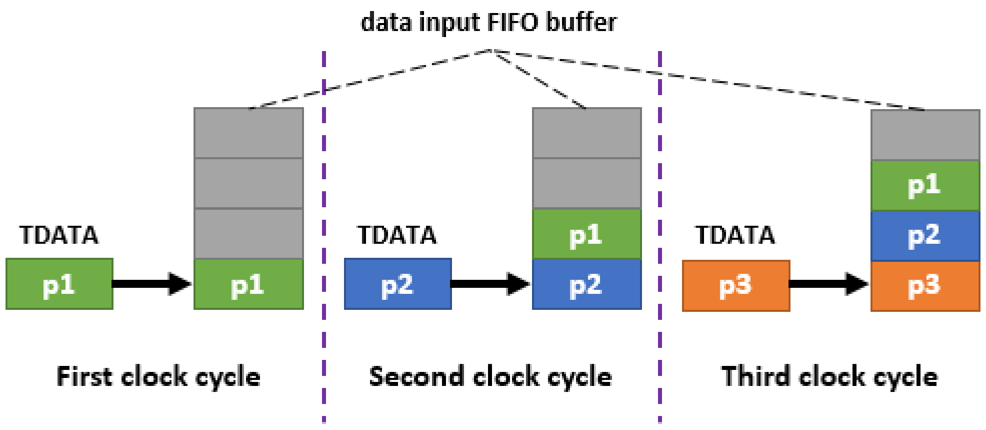


Figure 4.3 Writing TDATA payload to FIFO data input buffer.

After getting high signal TLAST or overflow of the data input FIFO buffer, it interrupts receiving by setting low TREADY signal, then sets high NEW_DATA_READY signal and moves to “PROCESSING” state. At this state, AXI4-Stream receiver is waiting high DATA_TRANSMITTED signal from the AXI4-Stream sender to moves back to “IDLE” state, sets low NEW_DATA_READY signal and is ready to receive a new packet.

4.2 AXI4-Stream sender implementation

The implementation of AXI4-Stream receiver was executed to be compliant with AXI4-Stream protocol as master. The receiving procedure controlled by FSM shown in Figure 4.4.

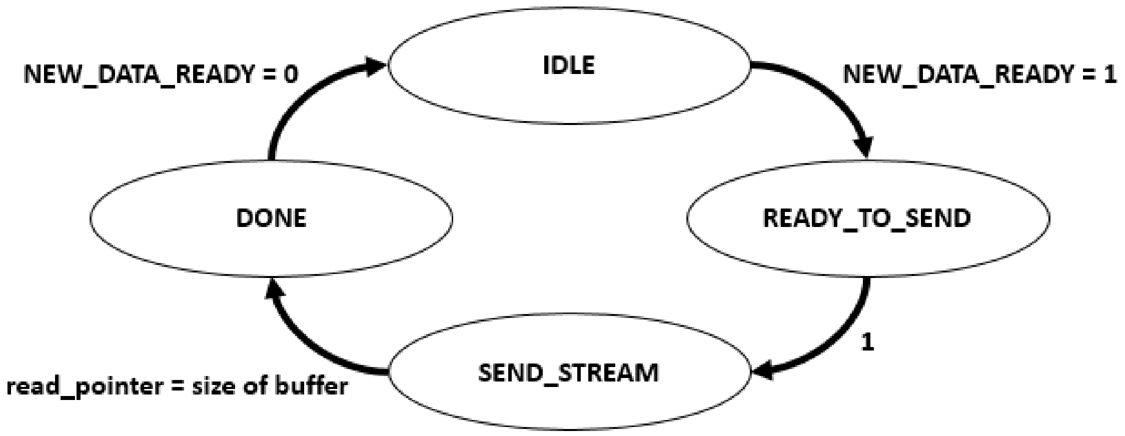


Figure 4.4 FSM of AXI4-Stream sender.

In comparison with the FSM of the AXI4-Stream receiver, the FSM of the AXI4-Stream sender consists of four states: “IDLE”, “READY_TO_SEND”, “SEND_STREAM” and “DONE”. At the “IDLE” state the sender is waiting the finish of a new packet reception by the AXI4-Stream receiver, which corresponds to a rising edge “READY_TO_SEND” signal. The “READY_TO_SEND” state represents a one clock cycle delay. The transition from “IDLE” to “READY_TO_SEND” state and the unconditional transition from “READY_TO_SEND” to “SEND_STREAM” state takes a total of two clock cycles, which are necessary for Two-Step Sorting Network to finish its job. At the “SEND_STREAM” state the sender is performing a transmission to AXI DMA through AXI4-Stream protocol according to the protocol specification. The sender uses the pointer to read and send data from output data buffer, which is incremented at every transmission clock cycle as shown in Figure 4.5.

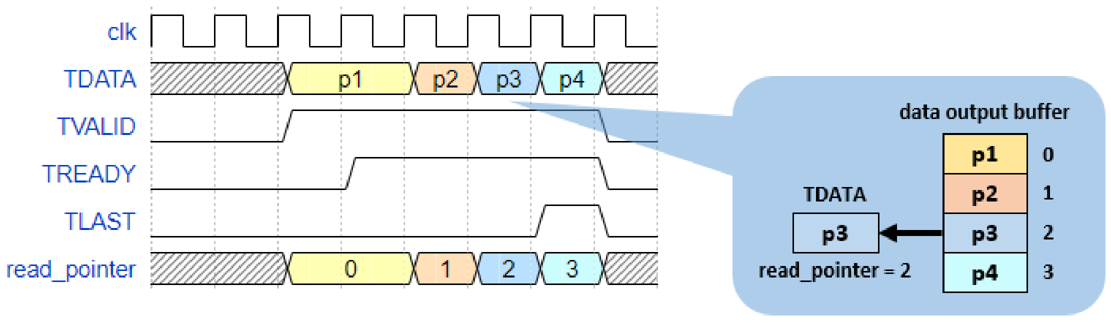


Figure 4.5 Reading data from data output buffer.

When the read pointer reaches the size of the data output buffer, the sender is finalizing transmitting the packet by setting high TLAST signal and moving to “DONE” state. At this state AXI4-Stream sender is waiting low NEW_DATA_READY signal from the AXI4-Stream receiver to moves back to “IDLE” state, sets low M_DATA_TRANSMITTED signal and ready for sending a new packet.

4.3 Two-Step Sorting Network implementation

The implementation of the proposed sorting network is based on idea of an O(1) Time Complexity Two-Step Sorting Network from [1] and consists of three main entities as shown in Figure 4.6.

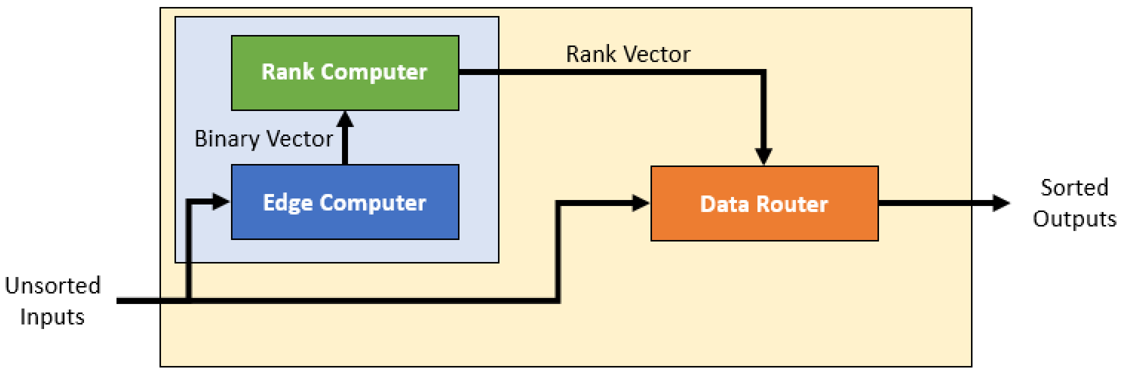


Figure 4.6 Structure of implemented Two-Step Sorting Network.

The Edge Computer simultaneously compares elements from the unsorted data array with each other. The result of comparison is the Binary Vector, which width equals of total number of comparisons between the elements of data array. Total number of comparisons for n-elements can be calculated as follows:

$$number\ of\ comparisons = n\frac{n-1}{2} \tag{6}$$

The Binary Vector is optimized form of the Edge Matrix. The Edge Matrix optimization process depicted in Figure 4.7.

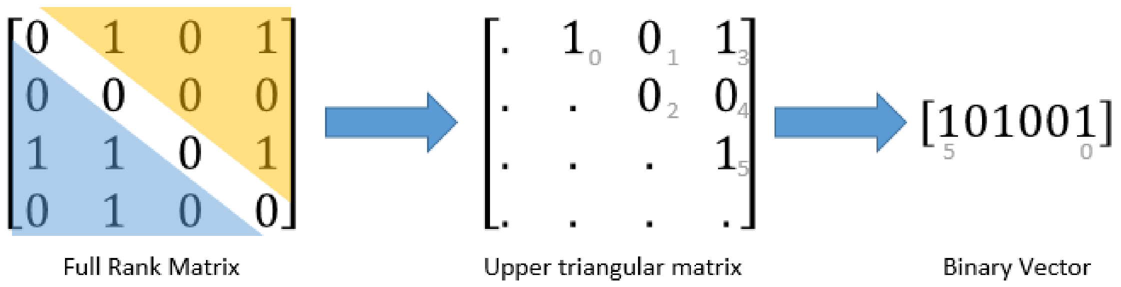

$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$
Full Rank Matrix
$\begin{bmatrix} . & 1_0 & 0_1 & 1_3 \\ . & . & 0_2 & 0_4 \\ . & . & . & 1_5 \\ . & . & . & . \end{bmatrix}$
Upper triangular matrix
$[1_5 0 1 0 0 1_0]$
Binary Vector

Figure 4.7 Optimization of the Edge Matrix.

In optimization purposes, comparison between array elements will be executed only once and in one direction, the comparison result in opposite direction is the result of inversion of corresponding bit. In accordance with the foregoing, an upper triangular matrix was obtained. On the next step, the upper triangular matrix is transformed into a Binary Vector by reading each column from left to right and from up to down, with the exception of the matrix diagonal.

Implementation the Edge Computer on hardware level represents a set of comparators as shown in Figure 4.8.

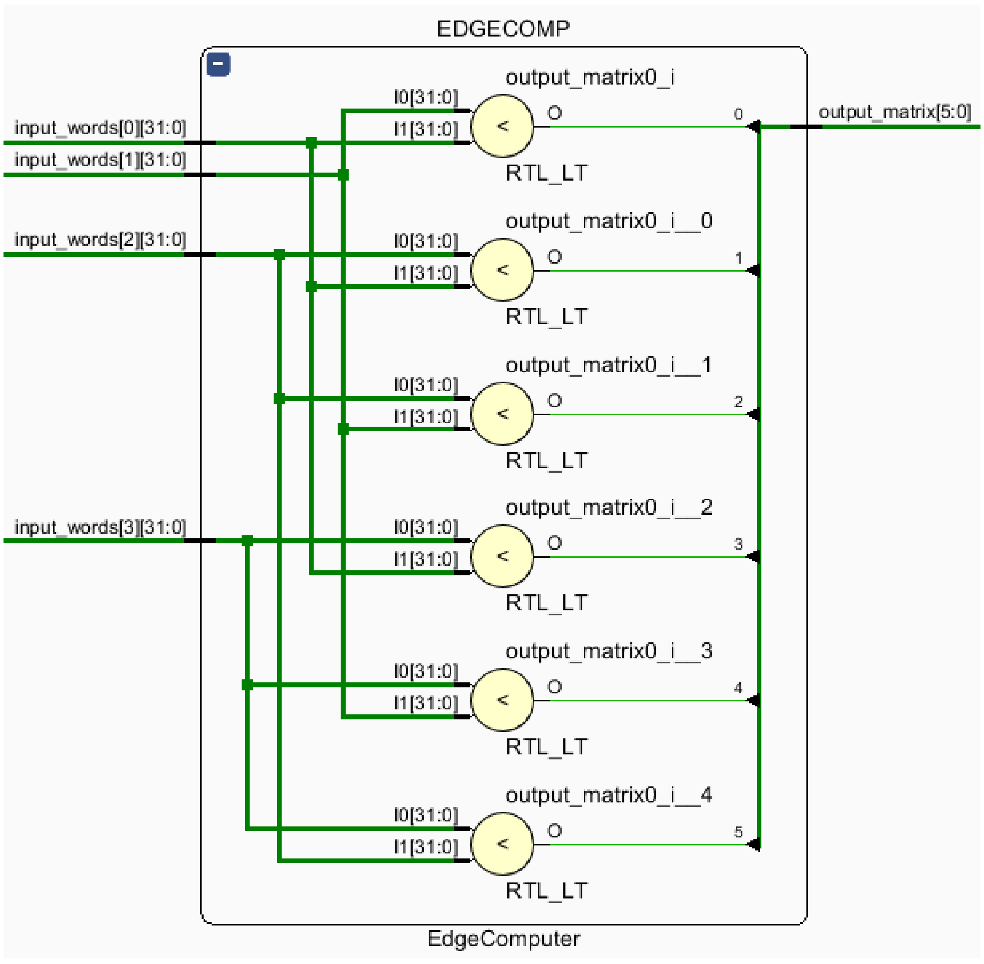


Figure 4.8 Hardware representation of the Edge Computer for 4-input configuration.

The Rank Computer simultaneously calculates and counts number of ones for each data element in the Binary Vector. Reading of the Binary Vector by the Rank Computer is depicted in Figure 4.9 as an upper triangular matrix, in order to better understanding.

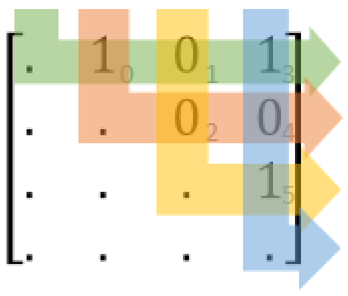


Figure 4.9 Reading of the Binary Vector representation in an upper triangular matrix form.

In upper triangular matrix representation, reading starts from the second column and going down the matrix till matrix diagonal. In upper triangular matrix representation, reading starts from the left column and going down the matrix till matrix diagonal, where the reading is going from to right with inversion of bits as depicted in Figure 4.9. The reading result of the comparison results for the second element from Figure 4.9 example, can be expressed as following:

$$[i_0\bar{\imath}_2\bar{\imath}_4] = [1\bar{0}\bar{0}] = [111] \tag{7}$$

The final rank result for the second element from the unsorted data array equals 3. After computation of all elements ranks, the Rank Vector is propagated to the Data Router.

Lastly, the Data Router is routing inputs according to a rank value of corresponding element from the Rank Vector and outputs the data array in descending order. The implemented Two-Step Sorting Network is shown in Figure 4.10. [14]

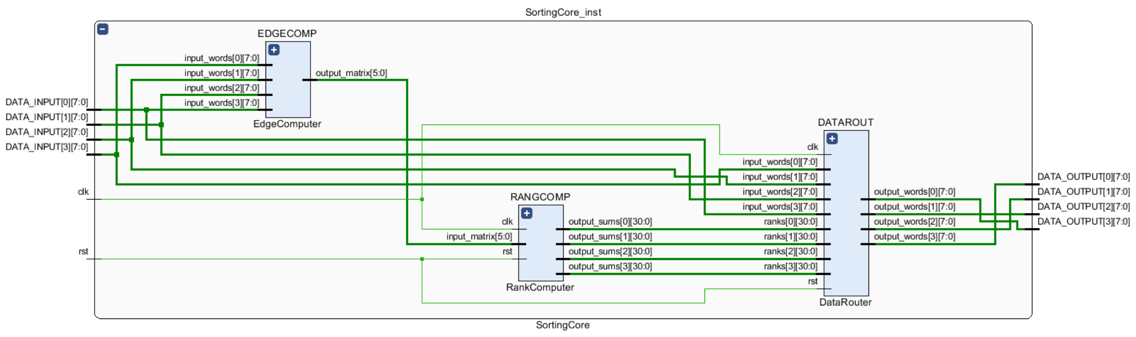
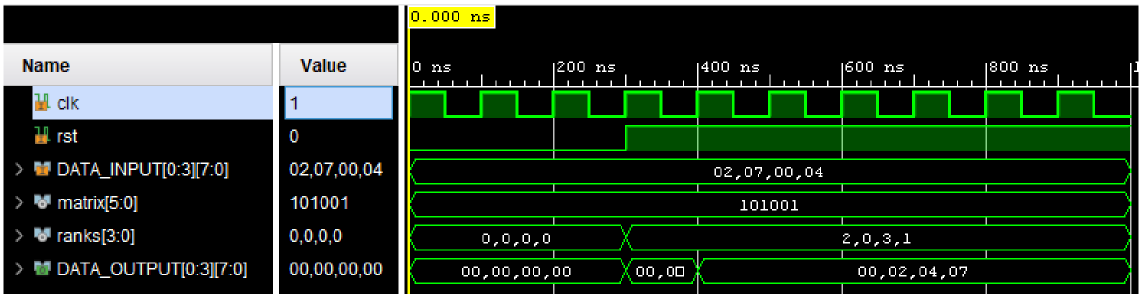


Figure 4.10 Sorting Network internal entities in Vivado.

This implementation takes two clock cycles to sort incoming data array. At first clock cycle Edge Computer and Rank Computer calculates Binary Vector and Rank Vector. At second clock cycle, the Data Router executes routing of unsorted data.

For testing purposes, the width of data and size of array is fully configurable for this implementation. Simulation of implemented Two-Step Sorting Network with 8-bit 4-input configuration is shown in Figure 4.11. [15]



Name	Value
clk	1
rst	0
DATA_INPUT[0:3][7:0]	02,07,00,04
matrix[5:0]	101001
ranks[3:0]	0,0,0,0
DATA_OUTPUT[0:3][7:0]	00,00,00,00

Figure 4.11 Simulation of implemented Two-Step Sorting Network.

5 Experiment results and analysis

This chapter describes experimental results of Two-Step Sorting Network IP Core implemented in Chapter 4 in scope of system designed in Chapter 3. Evaluation of the proposed implementation has been done through a set of experiments with different system configuration.

5.1 Post-implementation results and resources utilization

Evaluation of the post-implementation results and resources utilization has been done through a set of experiments with different data width and data array size for sorting. During this experiment we counted only the number of look-up tables (LUTs) utilized by Two-Step Sorting Network IP Core, which are the primary PL/FPGA resources.

In the first set of experiments was selected four data sets with widths of 8, 16, 32 and 64 bits and constant size of array (8 items). The results are shown in Table 1 and Figure 5.1.

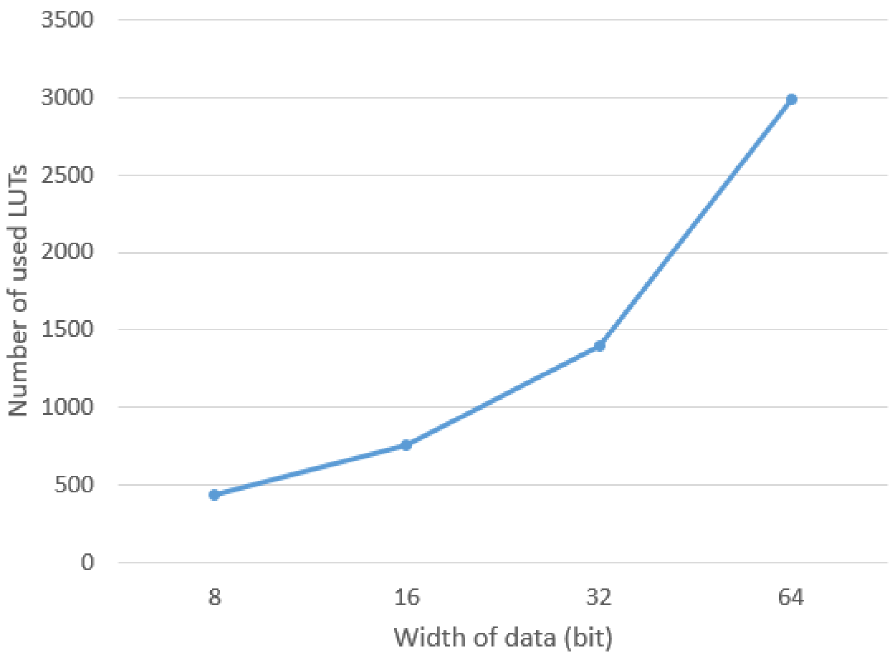


Figure 5.1 The result of utilization LUTs with different data widths.

Table 1 The result of utilization LUTs with different data widths.

Width of data (bit)	**Number of used LUTs**
8	441
16	762
32	1399
64	2984

In the second set of experiments was selected four data sets sizes of 8, 16, 32 and 64 items, with constant width of data (32 bits). The results are shown in Table 2 and Figure 5.2.

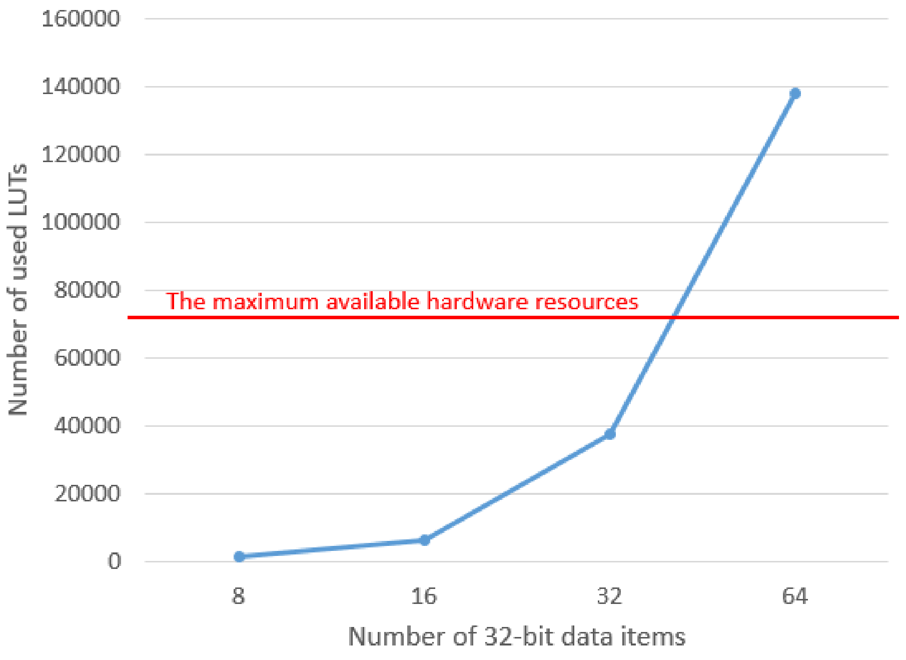


Figure 5.2 The result of utilization LUTs with different sizes of data set.

Table 2 The result of utilization LUTs with different sizes of data set.

Number of 32-bit data items	**Number of used LUTs**
8	1399
16	6452
32	37449
64	138030

The following conclusions can be drawn from the first and the seconds experiments:

- In proposed implementation of Two-Step Sorting Network IP Core utilizes less number of LUTs for sorting 8 items 8-bit data set than the implementation in [1], which utilizes 713 LUTs for the same configuration.
- The changing of the sorting data width leads to expected moderate exponential growth of utilized LUTs.
- The changing size of sorting dataset leads to very intensive growing of utilized LUTs. At 64 32-bit data set, the proposed implementation of sorting network utilized more resources than available on Xilinx Zynq UltraScale+ XCZU3EG-1SFVA625 MPSoC, which has only 71K LUTs.
- This implementation is not suitable for big data volumes, according to the intensive growth of utilized resources with growth of data set size.

5.2 Two-Step Sorting Network monitoring with Integrated Logic Analyzer

For analysing of correctness Two-Step Sorting Network implementation work at signal level, additionally was added debug signals to the IP Core, which were connected to the Integrated Logic Analyzer (ILA) IP Core, provided by Xilinx, as shown in Figure 5.3.

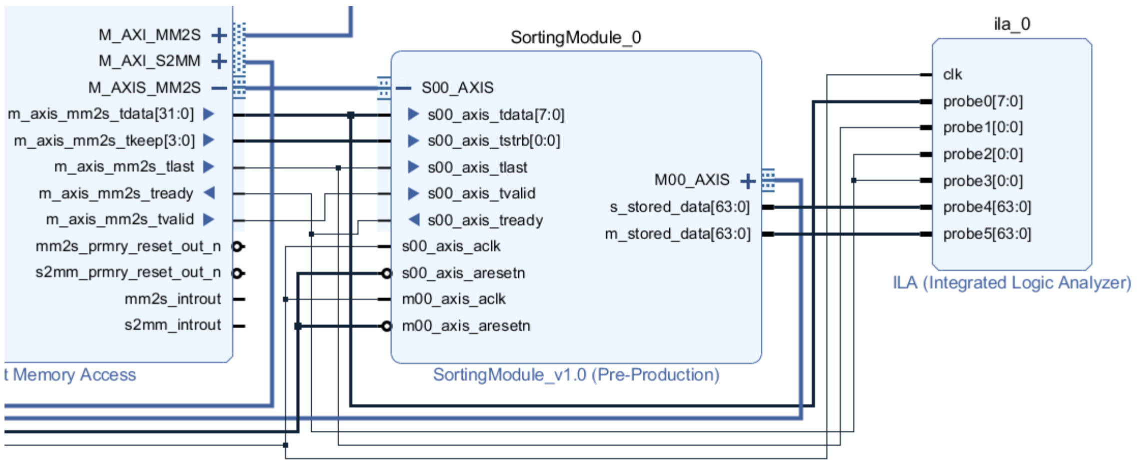


Figure 5.3 Integrated Logic Analyzer connections.

For testing purposes the standalone application was written in C programming language. Its sends single set of 8 8-bit test data into Two-Step Sorting Network IP Core through AXI DMA. The result of monitoring receiving and sorting by ILA is shown in Figure 5.4.

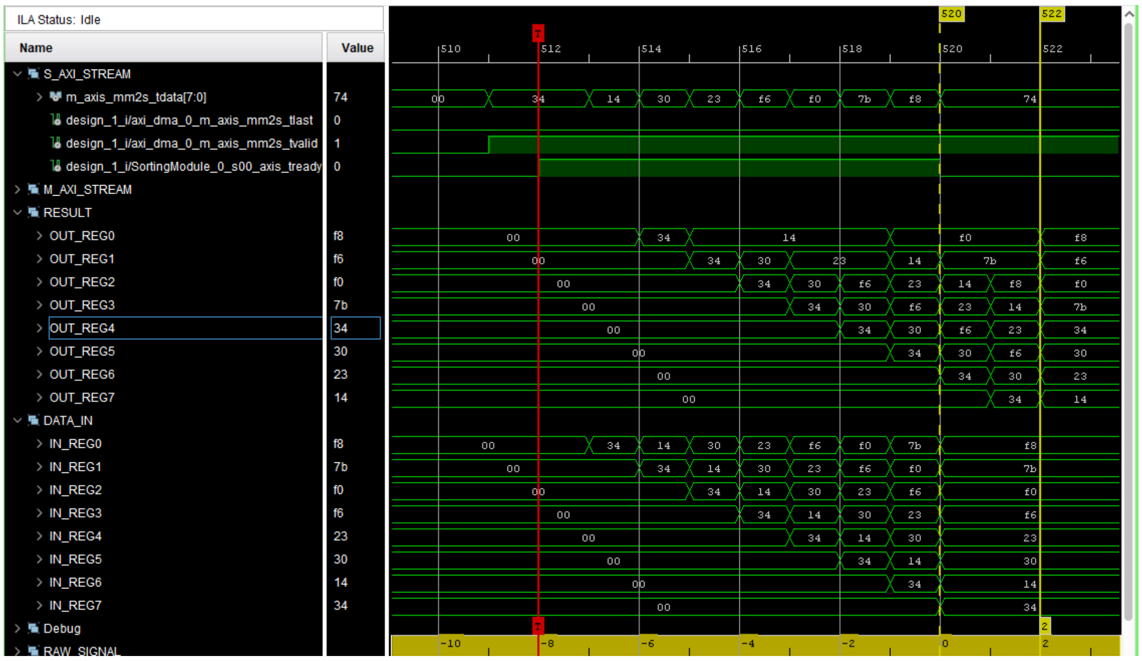


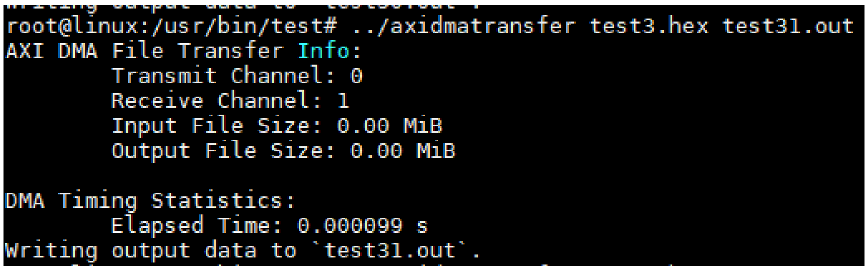
Figure 5.4 Sorting network IP core internal signal monitoring.

The Two-Step Sorting Network IP Core works as expected and performs sorting in two clock cycle.

5.3 Two-Step Sorting Network execution result at Petalinux

For analysing of correctness two directional communication between Processing System running on Petalinux OS and Two-Step Sorting Network IP Core through AXI DMA, was performed the sorting on Two-Step Sorting Network IP Core and result validation by additional developed application. The configuration and building of the Petalinux boot image were described in Appendix 2 – Configuration and building of Petalinux boot image. [16] [17]

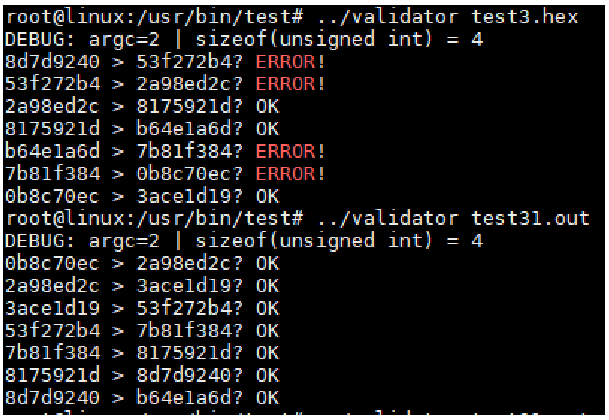
The one of the result of execution sending and receiving set of 8 32-bit test data to Two-Step Sorting Network IP Core is shown in Figure 5.5. Result of validation the unsorted set of data and sorted set of data after transfer execution is shown in Figure 5.6.



```
root@linux:/usr/bin/test# ../axidmatransfer test3.hex test31.out
AXI DMA File Transfer Info:
        Transmit Channel: 0
        Receive Channel: 1
        Input File Size: 0.00 MiB
        Output File Size: 0.00 MiB

DMA Timing Statistics:
        Elapsed Time: 0.000099 s
Writing output data to `test31.out`.
```

Figure 5.5 Transferring set of 8 test data by “axidmatransfer” application.



```
root@linux:/usr/bin/test# ../validator test3.hex
DEBUG: argc=2 | sizeof(unsigned int) = 4
8d7d9240 > 53f272b4? ERROR!
53f272b4 > 2a98ed2c? ERROR!
2a98ed2c > 8175921d? OK
8175921d > b64e1a6d? OK
b64e1a6d > 7b81f384? ERROR!
7b81f384 > 0b8c70ec? ERROR!
0b8c70ec > 3ace1d19? OK
root@linux:/usr/bin/test# ../validator test31.out
DEBUG: argc=2 | sizeof(unsigned int) = 4
0b8c70ec > 2a98ed2c? OK
2a98ed2c > 3ace1d19? OK
3ace1d19 > 53f272b4? OK
53f272b4 > 7b81f384? OK
7b81f384 > 8175921d? OK
8175921d > 8d7d9240? OK
8d7d9240 > b64e1a6d? OK
```

Figure 5.6 Validation of the result of sorting on hardware.

Were performed 10 transferring and calculated the average elapsed time per transfer to and from Two-Step Sorting Network IP Core, which equals 74.1 microseconds. Assuming that the PL runs at 100 MHz and sorting takes two clock cycles, the time consumes on sorting equals 20 nanoseconds or 0.02 microseconds. Moreover, transfer of the packet with 8 data items through AXI4-Stream in one direction, at the same frequency, will take 90 nanoseconds (8 clock cycles for data and plus one clock cycle for handshake). Two transmitting and sorting will take 200 nanoseconds or 0.2 microseconds. As result, 73.9 microseconds were consumed by driver and operation system.

In accordance with the foregoing calculations, using Two-Step Sorting Network IP Core with small volume of data is impractical due to the large time consumption on transfer.

6 Conclusion and Future work

This thesis explored hardware implementation of an O(1) Time Complexity Two-Step Sorting Network on FPGA. This chapter summarizes the main thesis contributions and outlines the directions for the future work.

The hardware accelerator proposed in this thesis is in very high demand in many fields and especially in those where time and resource consumption is critical. Fast sorting is vital task in many real-time systems.

The main contribution of the presented work is wider exploring an O(1) Time Complexity Two-Step Sorting Network proposed in [1]. The results of experiments demonstrate the big growth of resource utilization by incrimination of data set size for proposed Two-Step Sorting Network implementation. Furthermore, using this implementation with Petalinux is impractical due to the large time consumption on transfer. However, the implemented sorting network works properly and has better optimization of resource consumption for set of 8 8-bit data regarding to the results in [1] for the same configuration.

The following future work are possible to do:

- According to the AXI4-Stream protocol, the data transferred in serial. The proposed solution can be more optimized for the work with AXI4-Stream protocol, to dispose of two clock cycle delay for processing.
- Applying others sorting network for current implementation AXI4-Stream based IP Core is possible. Proposed IP Core is based on module design and each module easily can be replaced.

References

[1] P. TaghiniaJelodari, M. ParsaKordasiabi, S. Sheikhaei and B. Forouzandeh, An O(1) Time Complexity Two-Step Sorting Network with Hardware, IEEE Transactions on Very Large Scale Integration, 2019.

[2] A. Obuhov, *Project source code git repository.*

[3] *Zynq UltraScale+ MPSoC Data Sheet,* Xilinx, 2019.

[4] D. E. Knuth, The Art of Computer Programming vol. III, Addison-Wesley, 2011.

[5] D. Saveski, UltraZed-EG SOM Hardware User Guide, AVNET, 2017.

[6] *AXI DMA v7.1 LogiCORE IP Product Guide,* Xilinx, 2019.

[7] "PS/PL Interfaces," Xilinx, 2018. [Online]. Available: https://pynq.readthedocs.io/en/v2.5/overlay_design_methodology/pspl_interface.html. [Accessed March 2020].

[8] Zynq UltraScale+ MPSoC: Embedded Design Tutorial, Xilinx, 2019.

[9] J. Lant, C. Concatto, A. Attwood, J. A. Pascual, M. Ashworth, J. Navaridas, M. Lujan and J. Goodacre, *Enabling shared memory communication in networks of MPSoCs,* Wiley, 2017.

[10] AXI Reference Guide, 2011: Xilinx.

[11] AXI4-Stream Infrastructure IP Suite v3.0, Xilinx, 2018.

[12] L. Vosandi, "Arbitrary data streams," [Online]. Available: https://lauri.xn--vsandi-pxa.com/hdl/zynq/axi-stream.html. [Accessed February 2020].

[13] S. Baranov, Logic and System Design of Digital Systems, TUT Press, 2008.

[14] *Vivado Design Suite: Creating and Packaging Custom IP,* Xilinx, 2018.

[15] V. A. Pedroni, Circuit design and simulation with VHDL, MIT press, 2004.

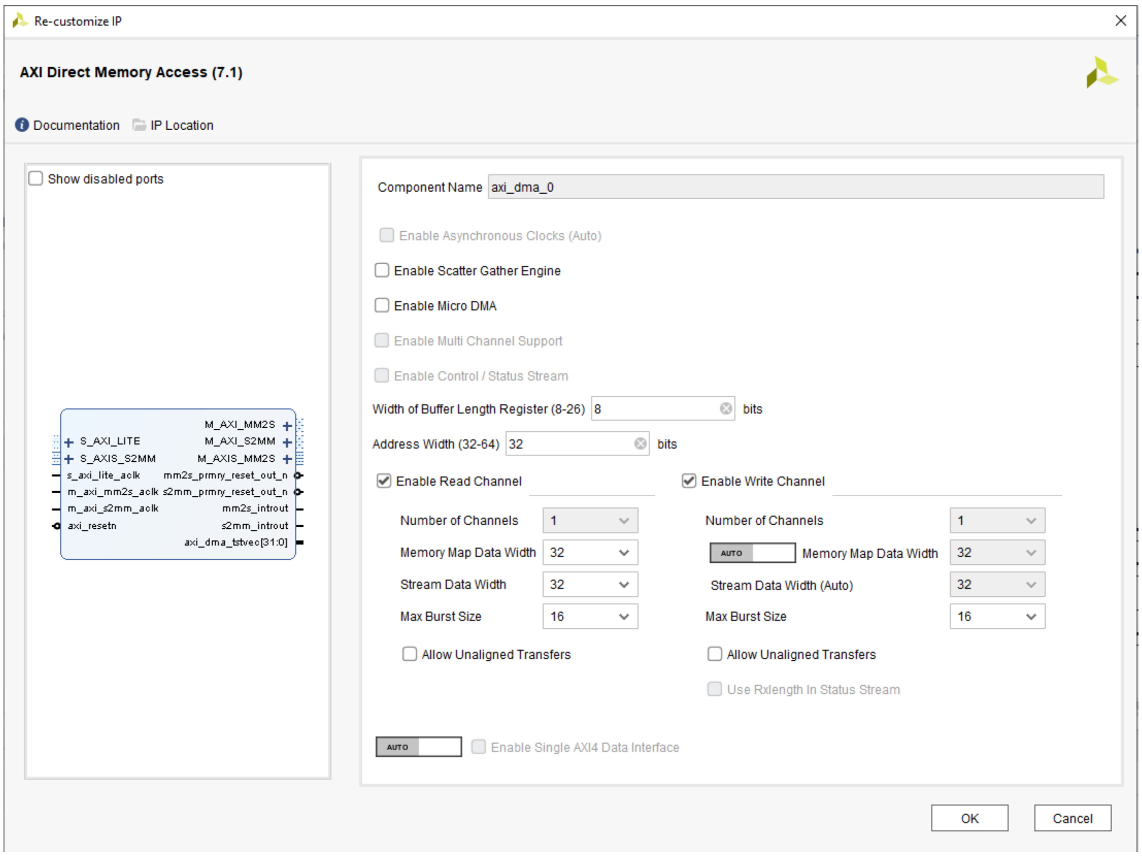
[16] PetaLinux Tools Documentation, Xilinx, 2019.

[17] PetaLinux SDK User Guide, Xilinx, 2013.

[18] B. Perez, "Linux driver for Xilinx's AXI DMA," GitHub, [Online]. Available: https://github.com/bperez77/xilinx_axidma. [Accessed 2020].

[19] V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson and C. Cardoso, Hardware/Software Co-design for Programmable Systems-on-Chip, TUT PRESS, 2014.

Appendix 1 – AXI Direct Memory Access IP Core customization page



AXI Direct Memory Access (7.1)

☐ Show disabled ports

Component Name axi_dma_0

☐ Enable Asynchronous Clocks (Auto)

☐ Enable Scatter Gather Engine

☐ Enable Micro DMA

☐ Enable Multi Channel Support

☐ Enable Control / Status Stream

Width of Buffer Length Register (8-26) 8 bits

Address Width (32-64) 32 bits

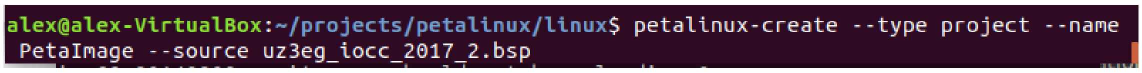
✓ Enable Read Channel		✓ Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	32	AUTO Memory Map Data Width	32
Stream Data Width	32	Stream Data Width (Auto)	32
Max Burst Size	16	Max Burst Size	16
☐ Allow Unaligned Transfers		☐ Allow Unaligned Transfers	
		☐ Use Rxlength In Status Stream	

AUTO ☐ Enable Single AXI4 Data Interface

Appendix 2 – Configuration and building of Petalinux boot image

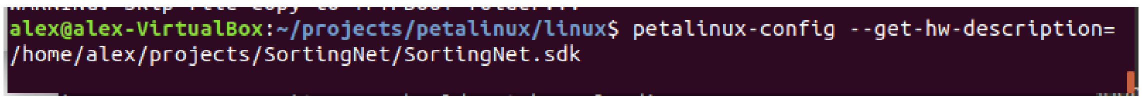
All operations in this chapter are done in PC with Ubuntu 20.04 LTS and installed PetaLinux Tool 2018.3.

At the first step, was created the PetaLinux project from BSP available on AVNET home page, using next command:



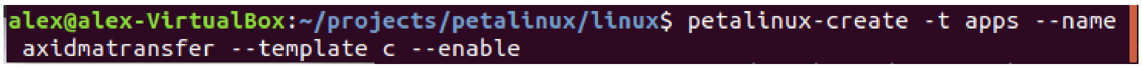
```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-create --type project --name
 PetaImage --source uz3eg_iocc_2017_2.bsp
```

On the next step, the created project was configured according the generated hardware description by Xilinx SDK, using next command. The settings were left by default.

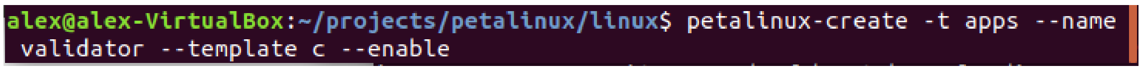


```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-config --get-hw-description=
/home/alex/projects/SortingNet/SortingNet.sdk
```

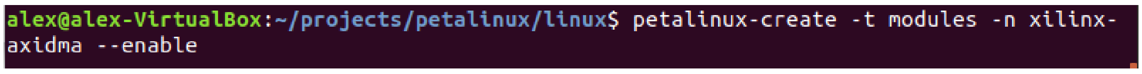
After configuration generation, was initialized two additional Petalinux application and one additional Linux kernel module by following commands:



```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-create -t apps --name
 axidmatransfer --template c --enable
```



```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-create -t apps --name
 validator --template c --enable
```

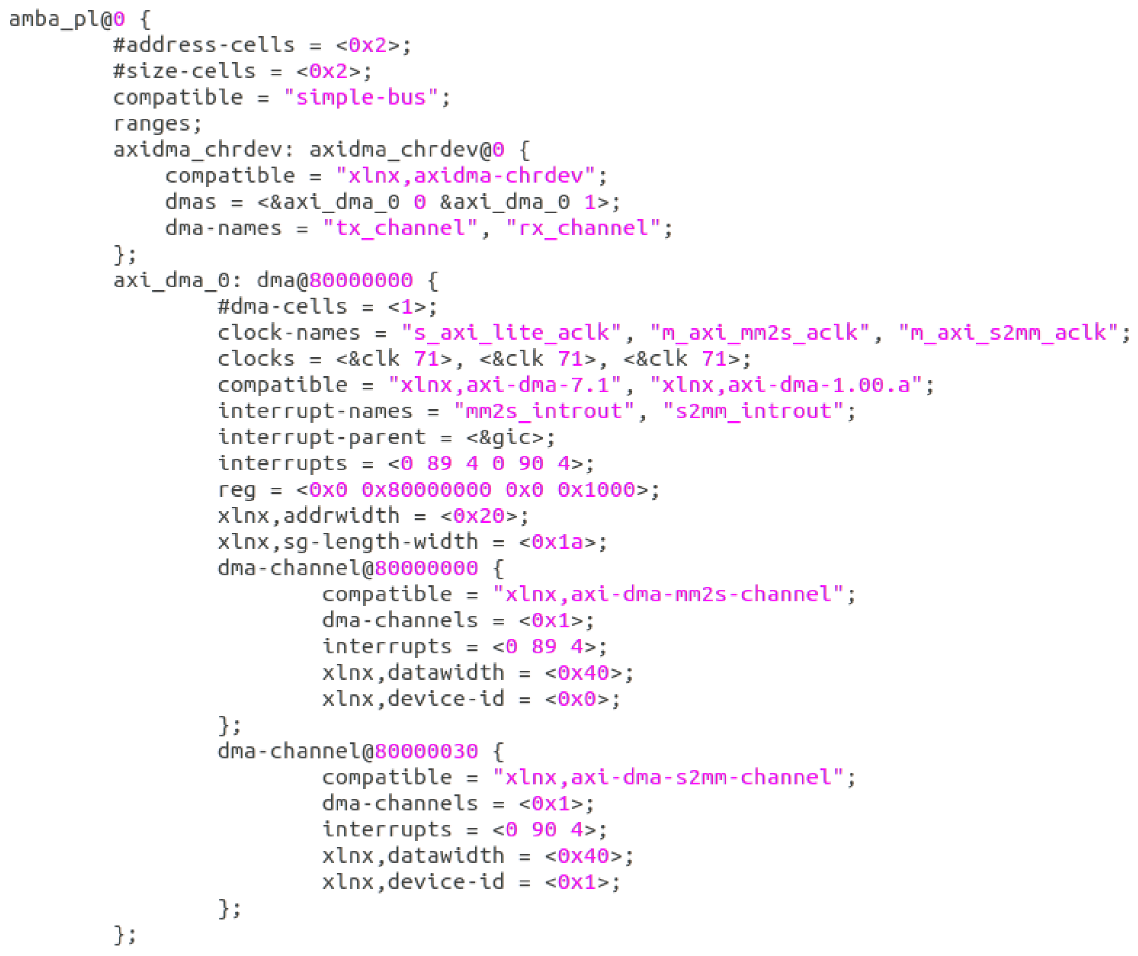


```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-create -t modules -n xilinx-
axidma --enable
```

In the Linux kernel module and the axidmatransfer application directory were added sources from xilinx_axidma git repository. [19]

In the validator application was added separately developed source code, which is available in project git repository. [2]

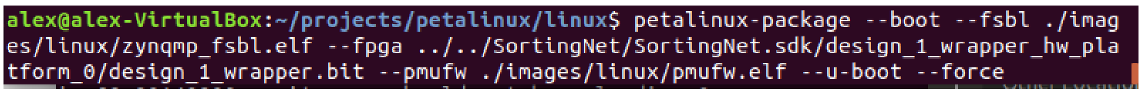
The last step before building the kernel, was added custom configuration for the AXI DMA device tree to "*system-user.dtsi*" file in *"~/projects/petalinux/linux/project-spec/meta-user/recipes-bsp/device-tree/files*" directory as follow:



```
amba_pl@0 {
        #address-cells = <0x2>;
        #size-cells = <0x2>;
        compatible = "simple-bus";
        ranges;
        axidma_chrdev: axidma_chrdev@0 {
            compatible = "xlnx,axidma-chrdev";
            dmas = <&axi_dma_0 0 &axi_dma_0 1>;
            dma-names = "tx_channel", "rx_channel";
        };
        axi_dma_0: dma@80000000 {
                #dma-cells = <1>;
                clock-names = "s_axi_lite_aclk", "m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
                clocks = <&clk 71>, <&clk 71>, <&clk 71>;
                compatible = "xlnx,axi-dma-7.1", "xlnx,axi-dma-1.00.a";
                interrupt-names = "mm2s_introut", "s2mm_introut";
                interrupt-parent = <&gic>;
                interrupts = <0 89 4 0 90 4>;
                reg = <0x0 0x80000000 0x0 0x1000>;
                xlnx,addrwidth = <0x20>;
                xlnx,sg-length-width = <0x1a>;
                dma-channel@80000000 {
                        compatible = "xlnx,axi-dma-mm2s-channel";
                        dma-channels = <0x1>;
                        interrupts = <0 89 4>;
                        xlnx,datawidth = <0x40>;
                        xlnx,device-id = <0x0>;
                };
                dma-channel@80000030 {
                        compatible = "xlnx,axi-dma-s2mm-channel";
                        dma-channels = <0x1>;
                        interrupts = <0 90 4>;
                        xlnx,datawidth = <0x40>;
                        xlnx,device-id = <0x1>;
                };
        };
```

At this moment, all preparation for building the PetaLinux boot image were done. Building the PetaLinux boot image is done with "*petalinux-build*" command.

After, to finalizing building of the PetaLinux boot image, was executed last command:



```
alex@alex-VirtualBox:~/projects/petalinux/linux$ petalinux-package --boot --fsbl ./imag
es/linux/zynqmp_fsbl.elf --fpga ../../SortingNet/SortingNet.sdk/design_1_wrapper_hw_pla
tform_0/design_1_wrapper.bit --pmufw ./images/linux/pmufw.elf --u-boot --force
```

Where "zynqmp_fsbl.elf" and "pmufw.elf" were generated in Xilinx SDK.

Final Linux boot image consists of three files: BOOT.BIN, image.ub and system.dtb.