UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Institute for Applied Microelectronics
Information and Communication Systems

Master of Science in Telecommunication Technologies

Master's Thesis

Design and Implementation of a MapReduce architecture for Big Data applications using High-Level Synthesis design flow

Author: Julian Spahr
Supervisor(s): Dr. Pedro Pérez Carballo
Prof. Antonio Núñez Ordóñez

Date: July 2017

IUMA
t +34 928 451 086 iuma@iuma.ulpgc.es Campus Universitario de Tafira
f +34 928 451 083 www.iuma.ulpgc.es 35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Institute for Applied Microelectronics
Information and Communication Systems

Master of Science in Telecommunication Technologies

Master's Thesis

Design and Implementation of a MapReduce architecture for Big Data applications using High-Level Synthesis design flow

Signatures

Author:	Julian Spahr	Signature:
Supervisor(s):	Dr. Pedro Pérez Carballo	Signature:
	Prof. Antonio Núñez Ordóñez	Signature:
Date:	July 2017	

IUMA
t +34 928 451 086 iuma@iuma.ulpgc.es
f +34 928 451 083 www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Institute for Applied Microelectronics
Information and Communication Systems

Master of Science in Telecommunication Technologies

Master's Thesis

Design and Implementation of a MapReduce architecture for Big Data applications using High-Level Synthesis design flow

Evaluation

Grade: .....

President: Signature:

Secretary: Signature:

Member: Signature:

Date: July 2017

IUMA
t +34 928 451 086 iuma@iuma.ulpgc.es
f +34 928 451 083 www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

ACKNOWLEDGEMENTS

I want to take the opportunity to thank my supervisors, Pedro Pérez Carballo and Antonio Núñez Ordóñez, for their special dedication and persistence during the development of this work. Thanks to their efforts I have been able to carry out this Master's Thesis.

I want to also thank my colleague and friend Yubal Barrios Alfaro, for his suggestions and propositions that have helped me improve this work, and sort my ideas whenever I doubted myself.

Lastly, I thank my family, their commitment and patience, through which I was able to finalize this master's degree. And specially my mother, for her endless curiosity and devotion.

RESUMEN

El presente proyecto abarca el diseño e implementación de una plataforma *hardware* que integra algoritmos críticos de procesamiento en aplicaciones para Big Data sobre un SoC de la serie Xilinx Zynq, mejorando de forma significativa su eficiencia con respecto a implementaciones *software* del mismo tipo. De forma específica, se ha diseñado una plataforma MapReduce aplicada al algoritmo *WordCount*, cuya arquitectura permite el reemplazo de bloques IP para modificar la aplicación de una plataforma a otra, siempre y cuando cumpla un esquema del tipo MapReduce.

Este trabajo comienza haciendo referencia a los aspectos claves del concepto de Big Data y técnicas para el análisis de datos masivos en sistemas modernos. Con ello se abarca el modelo de programación MapReduce, para procesar y generar grandes cantidades de datos por medio de técnicas de paralelismo sobre *clusters*. Seguidamente se abarca la serie Xilinx Zynq y la metodología de diseño empleada para la creación de plataformas flexibles y configurables para desembocar en las herramientas empleadas en el presente proyecto, entre las cuales se destaca el entorno de desarrollo de sistemas basados en IP, Xilinx Vivado Design Suite.

Finalmente, el proyecto se ha sometido a una fase de validación con objeto de estimar los parámetros del sistema final, entre los cuales se subrayan parámetros de latencia, *throughput* y nivel de utilización del sistema MapReduce. En base a estos resultados, se concluye que la plataforma diseñada representa una buena solución en lo que respecta al análisis de datos masivos para aplicaciones Big Data debido a su alta velocidad de cómputo, bajo consumo de potencia y nivel de utilización y en última instancia debido a su flexibilidad a ser adaptado a otras aplicaciones del tipo MapReduce de características similares.

ABSTRACT

This Master's thesis comprises the design and implementation of a hardware platform that integrates critical algorithms of Big Data application processing on a Xilinx Zynq series SoC, improving significantly it's efficiency in comparison to software implementations. Specifically, a MapReduce platform has been designed using the classic WordCount application to establish a generic architecture that allows for easy IP block substitution to modify the application into another, if it complies with the MapReduce outline.

This work starts presenting key aspects of the Big Data concept and techniques for massive data analysis in modern systems. Through this, we cover the MapReduce programming model, for large data processing and generation through parallelization on clusters. Following up, the used Xilinx Zynq Design Methodology is explained, that allows for the design of flexible and configurable platforms, leading finally into the tools used in this thesis, among which we emphasize the Xilinx Vivado Design Suite for SoC FPGA.

Lastly, the designed prototype is put through a validation phase to estimate the performance parameters of the system, i.e. latency, throughput and utilization of the MapReduce system. Based on these results, we conclude that the designed platform is a good solution with respect to massive data analysis for Big Data applications, due to its high computing speed, low power consumption and utilization and ultimately due to its flexibility to be adapted for other MapReduce applications of similar characteristics.

INDEX

FIGURE INDEX

TABLE INDEX

ACRONYMS

A

B

C

D

F

G

H

I

J

K

L

M

P

Chapter 1. INTRODUCTION

In this first chapter, the reader is presented with the necessary background and motivations that condition the work done in this thesis, including the objectives pursued by the authors and the structure that the current document holds. By careful reading of this chapter, the reader acquires a basic knowledge about the origins and expected results of this work.

1.1. BACKGROUND

1.1.1. Global IP traffic

Every day we use devices and systems that take advantage of data oriented infrastructure such as mobile phones, tablets and PC's, that process and generate large volumes of information. This includes traffic from social networks, search engines, e-mail exchange systems and a data derived from sensor monitorization systems used in public transport and infrastructure, vehicles such as airplanes, cars, etc.… [1], [2]. Figure 1 shows global Internet Protocol (IP) derived traffic, which is in continuous growth, with a 2020 forecast of 30.4 to 34.9 % traffic generated by North America and Asia Pacific regions, based on a 22 % Compound Annual Growth Rate (CAGR) in between the years 2015 and 2020 [2]. This implies global IP traffic of a volume around 110 exabytes (10^{18} bytes) per month, in 2017 only [2].

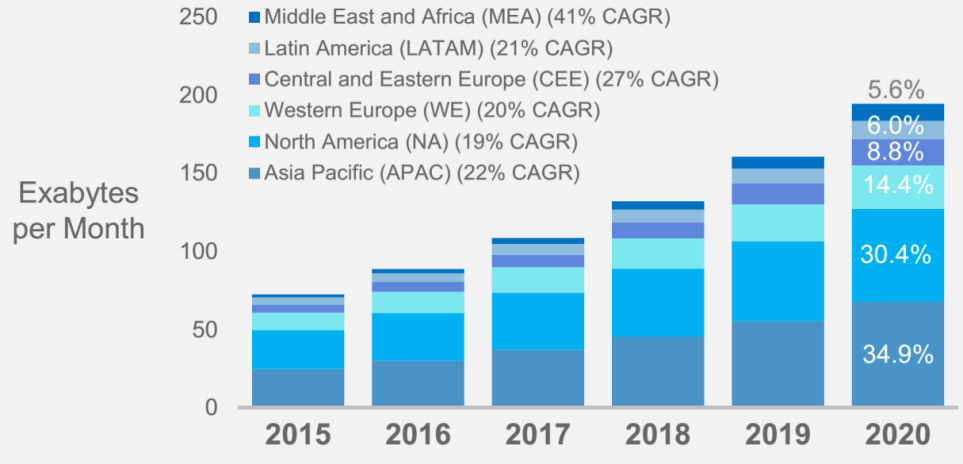

Figure 1. - Global IP traffic growth forecast per region [2]

To provide a satisfactory level of Quality of Service (QoS) and guarantee efficient network traffic, it is necessary to understand the underlying behaviour of IP networks and it's data processing systems and techniques, as the global IP traffic grows larger each day [3].

1.1.2. An introduction to Big Data and Map-Reduce

The term Big Data is understood as the traffic of information that flows in immeasurable quantities between sources, including social networks, public databases, mobile devices, bank and insurance transaction data, etc.… as it is shown in Figure 2.

Figure 2. - Big Data sources

Ultimately, Big Data is collected, structured, organized and indexed for its analysis and useful data extraction; since the elements that compose the datasets itself have no value on their own, we use sophisticated algorithms, comprised in the Data Mining and Big Data Analysis fields, to find recurring patterns and conceive useful information [1].

A classic approach to information processing in Big Data for large cluster organization is the MapReduce (MR) programming model, as proposed by Google and used to process vast amounts of data using a high amount of processing units [4]–[7]. The MapReduce Algorithm comprises two main phases: the mapping phase, where the data is processed and translated to useful information as a so called key-value (KV) pair, and the reduce phase, where the key-value structured data is merged based on a given criteria [6], [7]. This criterion could be a repetition counting system of either the key's or value's, a combination of both or arithmetic operations of any of them, as it depends on its ultimate application. A generic MapReduce architecture, that complies with the afore mentioned explanation, is shown in Figure 3. In this example, the data is firstly split into multiple data chunks and given to the mapping stage. Once the mapping is done, the information is shuffled and given to the reduce stage, where every reduce element represents a reduce criterion or application. The data is merged at the end of the reduce stage, to comprise useful, processed information.

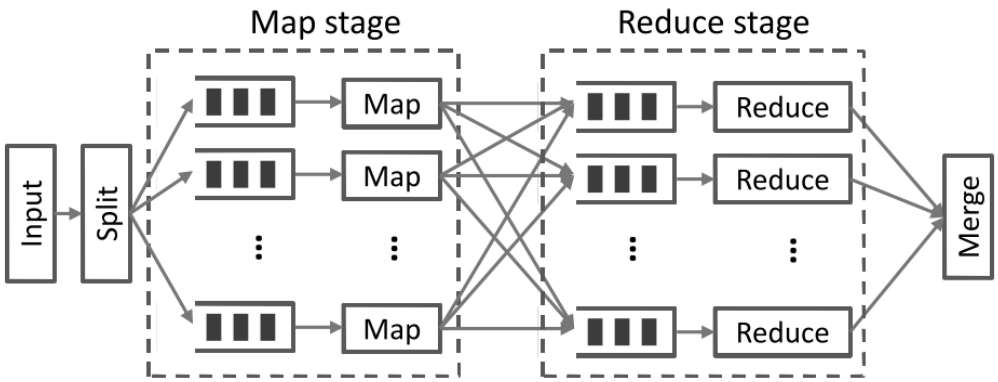

Figure 3. – Basic MapReduce architecture *[6]*

The MapReduce framework finds a wide range of applications, from matrix arithmetic (Matrix Multiplication), text processing (String Matching, Word Count), image processing (RGB Histogram) and a variety of Machine Learning (ML) and Artificial Intelligence (AI) kernels (Linear Regression, PCA, Kmeans, etc.…) [4]–[6].

Systems dedicated to data processing, analysis and organization, are computation intensive, which implies a non-negligible computation derived latency, high memory storage requirements and high power consumption. For this reason, it is necessary to design and develop custom solutions that try to attack these problems [6], [8], [9].

In order to address this need, as a result of recent research and developments in the field, it has been concluded that hybrid architectures, between General Purpose Processor (GPP) machines and Field-Programmable Gate Arrays (FPGA) on a single System On a Chip (SoC) based platform can address this problem, creating solutions with high computing capability and low power consumption [6], [8], [9].

1.1.3. FPGA based flexible platforms

From an architectural point of view, we can distinguish two main implementation types: software based programs and solutions on a multicore GPP or hardware based implementations on Application-Specific Integrated Circuits (ASIC) or FPGA [6], [9]. In between both of the presented solutions, FPGA based platforms implement efficiently complex electronic systems with low power consumption, although its flexibility is inferior to other applications, such as Digital Signal Processors (DSP), Graphics Processing Units (GPU) or GPP's, as it is shown in Figure 4.

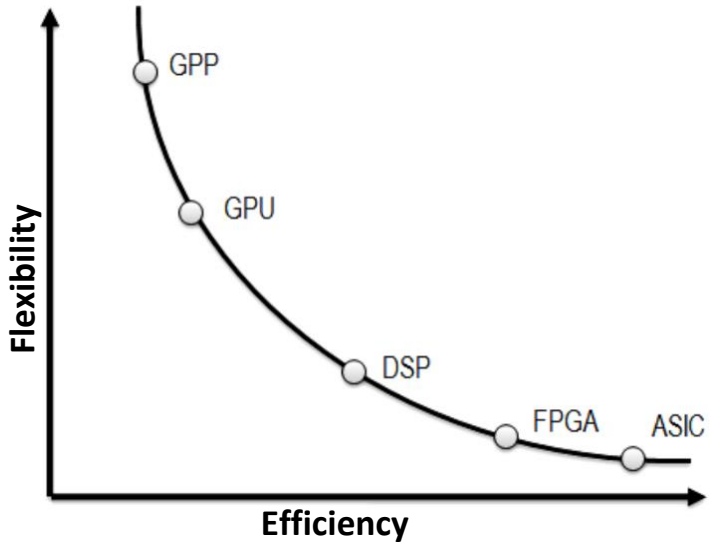

Figure 4. - Flexibility and efficiency trade-off for different computation solutions *[10]*

To achieve maximum flexibility between general purpose and particular solutions for a given system, we can incorporate GPP processors altogether with an FPGA core, thus

creating a system with the computation capability of addressing the desired task. Such a device is considered a FPGA based SoC.

From a commercial point of view, we can find configurable platforms of this kind (GPP + FPGA) from multiple different companies on the market in the field of programmable devices. This thesis focuses on solutions provided by Xilinx, particularly focusing on the Zynq All Programmable (AP) SoC series.

Xilinx Zynq-7000 AP SoC's, are made up of two main blocks: first, an ARM Cortex-A9 MPCore operating at a maximum frequency of 1 GHz, and secondly, a 7 series Xilinx FPGA made through TSMC's 28-nm HPL process [11]. Additionally, the mentioned SoC integrates multiple high-speed transceivers and peripherals to allow for a wide range of solutions with a single device [12]–[14]. Figure 5 shows the architecture of the Zynq-7000 All Programmable SoC device.

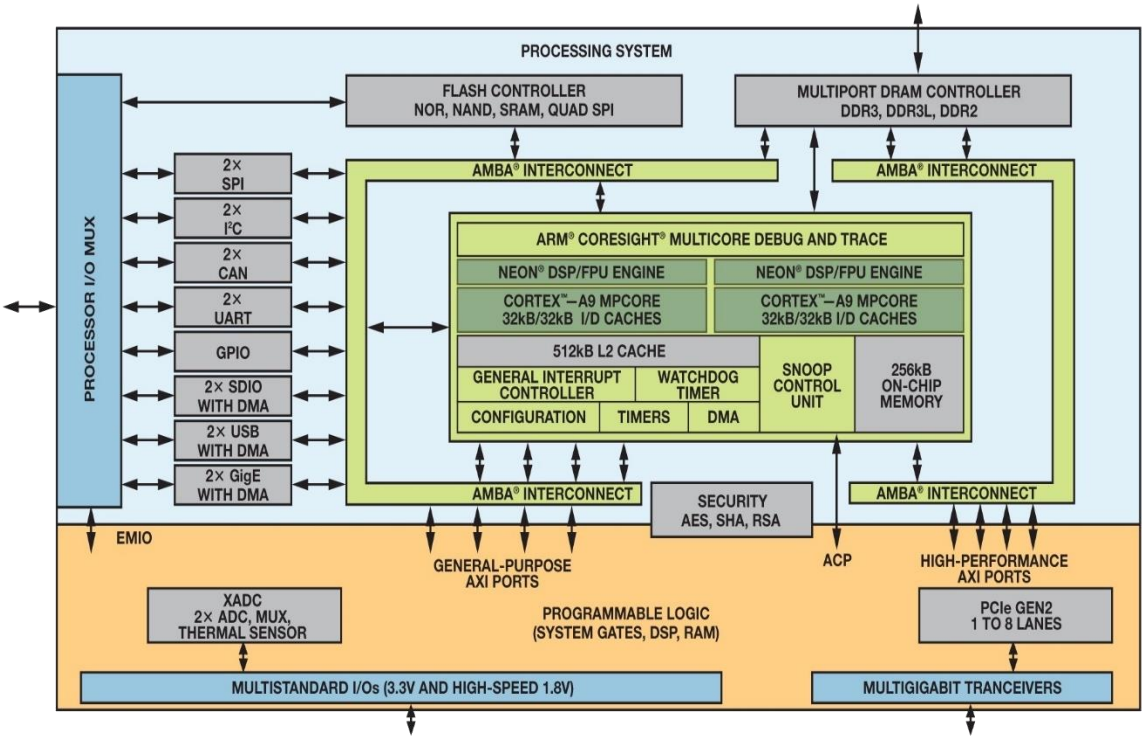

Figure 5. – The Zynq-7000 All Programmable SoC architecture

As shown in the previous figure, the Zynq-7000 devices are composed of Programmable Logic (PL, yellow) and the Processing System (PS, blue). The Zynq PL owns its features to the 7 series FPGA's from Xilinx and is connected to the PS through standard

Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) and Accelerator Coherency Port (ACP) interfaces.

This Master's Thesis uses the ZC706 Evaluation Kit, which employs two 32 bit ARM Cortex A9 Central Processing Units (CPU) and a 7 series Kintex FPGA [15]. The employed device (XC7Z045 FFG900-2, Figure 6) has enough resources to integrate a Big Data platform.

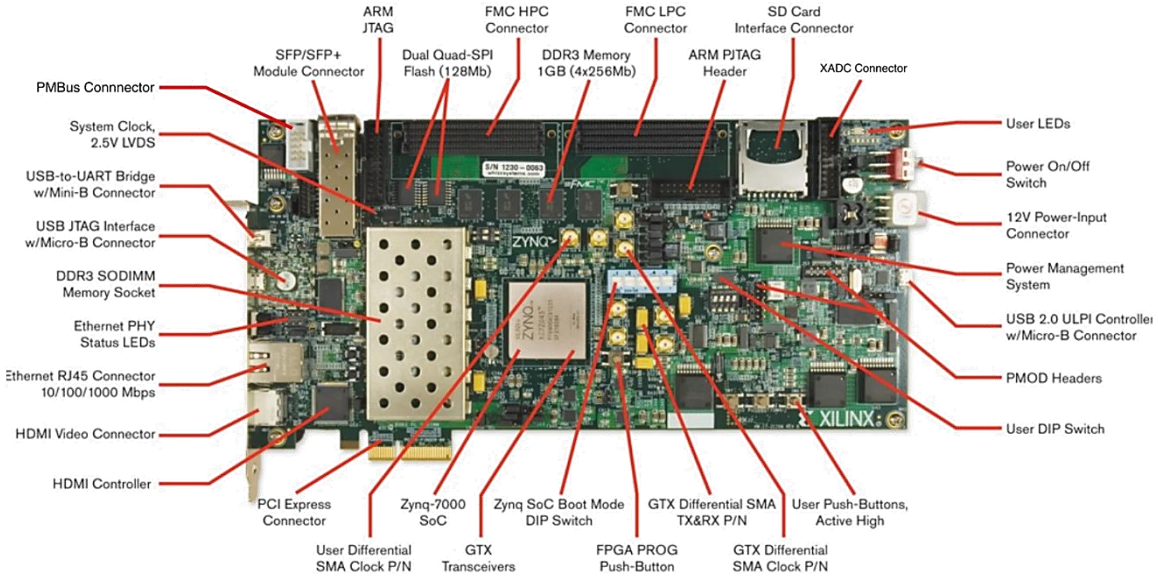

Figure 6. – ZC706 Board Features

1.1.4. State of the art

Previous work has been done in the MapReduce acceleration through FPGA based solutions. One of the most important contributions is the fairly recent work done by Diamantopoulos and Kachris in [6] by substituting high power consuming multi-core processors involved in MapReduce tasks with Virtex-7 FPGA's, improving the overall throughput by a factor of 4.3 and reducing power consumption by a factor of two orders of magnitude. Diamantopoulos and Kachris propose the so called HLSMapReduceFlow, a reconfigurable MapReduce accelerator that can be scaled up to data centers with high programming efficiency by using the Vivado HLS design flow [6], as presented in Figure 7.

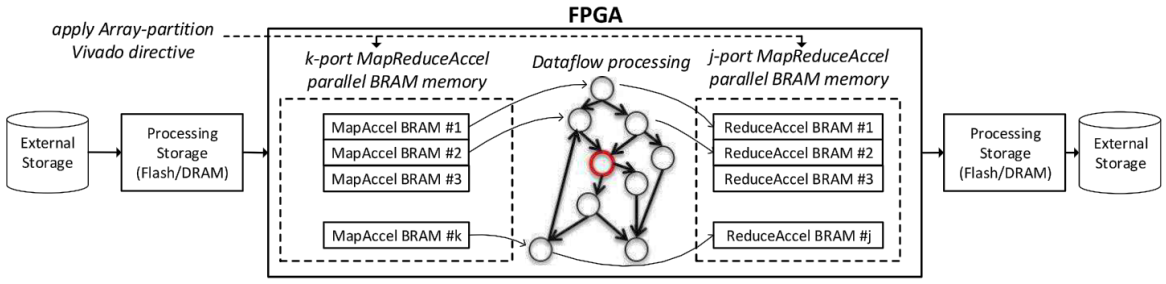

Figure 7. – HLSMapReduceFlow dataflow architecture *[6]*

In this case, the input data is split into fragments for the MapReduce architecture located in the FPGA hardware, and merged once the data reaches the end of a reduce stage. The mentioned MR architecture is put through a thorough verification phase by using Histogram, Matrix Multiplication, String Match, Linear Regression, PCA, Kmeans and Word Count applications. Simple applications (Word Count or Linear Regression) achieve an average computing time reduction by a factor of 2 while power consumption is reduced by a factor of 25 for all applications.

This architecture is fairly typical in such applications, as shown in [16] by Ranger, Raghuraman, Penmetsa et al., and used to present the Phoenix programming API, based on Google's MapReduce model [7]. The Phoenix implementation bases itself on the same basic principles upon which the MapReduce framework is constructed, which is: the map function processes the input data and generates a set of intermediate Key-Value pairs, and the reduce function merges the intermediate pairs which have the same key [16]. Phoenix differs from the original MR framework in that it targets shared-memory systems such as multi-core chips.

As of today, Phoenix' modern iteration receives the name of Phoenix++, which addresses limitations of the original Phoenix implementation, such as an improvement to the key-value storage, and correction to an inefficient combiner behavioural code that caused memory access overhead and an improvement to the overall MapReduce task chunking [17]. Phoenix++ however, employs the same basic data flow architecture as it's original, as presented in Figure 8.

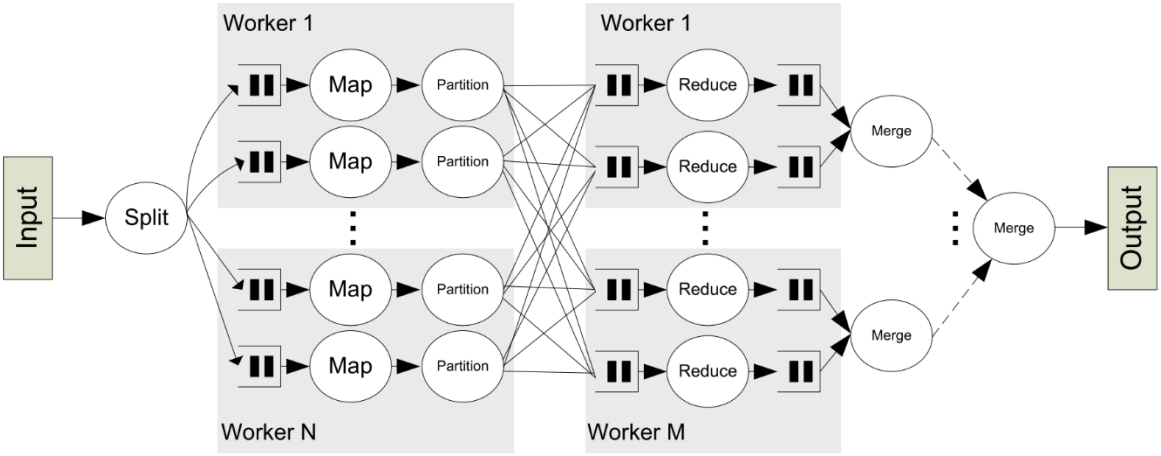

Figure 8. – Basic data flow architecture for Phoenix [17]

Phoenix and Phoenix++ utilize similar verification techniques as the aforementioned work by Diamantopoulos and Kachris in [6], by using Histogram, Linear regression, Kmeans, PCA and Word Count applications in order to estimate the efficiency of the implementation.

Although it is evident that Phoenix++ has its applications in shared-memory systems with predominantly a software implementation, we want to stress that the frameworks behaviour, approaches to latency and throughput issues and means of verification, serve as a backbone to this work to implement a hardware based MapReduce solution.

Another recent work done in the field of software implementations of a MapReduce framework, is presented in [18] by Lu, Zhan, Huynh et al., where an optimized MR framework is used on an Intel Xeon Phi processor and compared to the Phoenix++ model, evaluating both performances with the mentioned applications. Cao, Cui, Shi et al. [19] employ the MapReduce framework on a back-propagation (BP) neural network in combination with a Particle Swarm Optimization (PSO) algorithm to improve the PSO itself, optimize the design of the MR related tasks and therefore achieve a higher classification accuracy. Aly, Montréal, Shaban et al. [20] apply a MapReduce framework, employed on a Google Cloud Platform (GCP) to analyse data generated by industrial systems. Melia is a scalable MR framework utilized on OpenCL based FPGA's, as developed by Wang, Zhang, He et al. [21], that uses MapReduce + OpenCL to reduce significantly power and time consumption, as their Altera Stratix V GX FPGA is compared to CPU and GPU implementations with the same functionality and application.

In terms of heterogeneous accelerator architectures, Ghasemi and Chow [22] demonstrate a CPU + FPGA based dataflow that takes advantage of the Apache Spark cluster-computing framework in combination with a custom accelerator inside an FPGA; by comparing this solution with an Intel Xeon processor, the authors achieve a throughput improvement by a factor of 2, as the number of acceleration workers increase to 4.

Lastly, Neshatpour, Sasan and Homayoun [23] employ a heterogeneous accelerator architecture to develop a MapReduce implementation in an Apache Hadoop environment. The described architecture exploits a CPU + FPGA with a Xilinx Artix-7 and AMBA AXI interfaces in between workers to reduce the overhead caused by interactions between the cores. The employed architecture is presented in Figure 9 for a single-node accelerator.

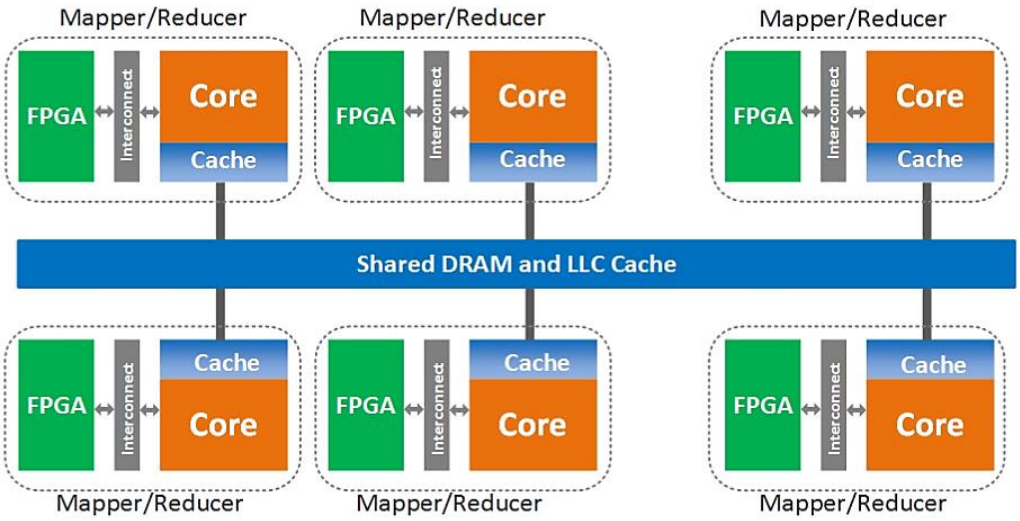

Figure 9. - Single-node heterogeneous accelerator architecture [23]

The shown solution allows for easy scalability as every node can be connected to a neighbour through a multi-channel gigabit switch [23], as shown in Figure 10.

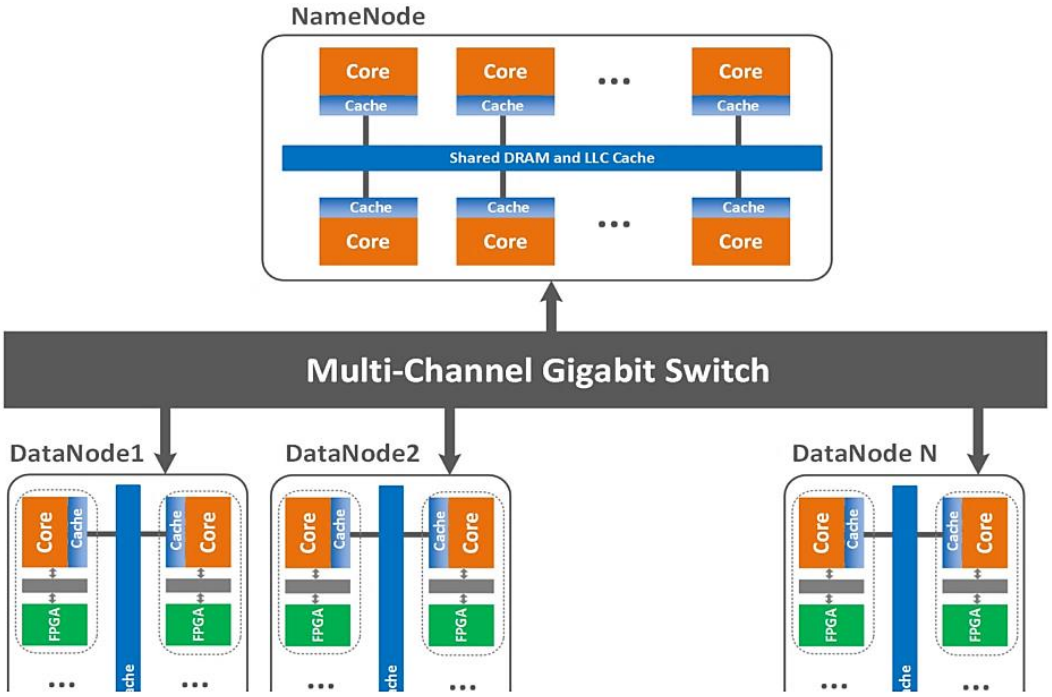

Figure 10. – Multi-node heterogeneous accelerator architecture [23]

Its efficiency was explored by executing Naïve Bayes, Kmeans, KNN and SVM applications, achieving a speedup factor between 1.008 and 4.561, and a power consumption reduction factor between 1.07 and 22.20.

As a conclusion of the presented state of the art exploration, we want to stress the growing trend of using heterogeneous architectures to implement MapReduce based frameworks for multiple applications to obtain energy and time efficient solutions. Although the scope of this thesis does not comprise the implementation of a multi-node solution, we follow the related work to obtain a MR friendly FPGA based node.

1.2. Objectives

The main objective of this thesis, is the implementation of a hardware based accelerator on FPGA based SoC, to migrate different elements of the main algorithms that comprise a Big Data application such as a MapReduce framework, therefore significantly improving the efficiency of the system through custom hardware design. Consequently, is important to study the different programming models that exist in Big Data applications, and particularly in MapReduce frameworks, that allow for massive data processing.

First and foremost, one of the main objectives of the current thesis is the study of the State of the Art in Big Data and MapReduce applications that are relevant in the field.

Secondly, we propose the development of a MapReduce worker platform that responds to the main needs of the field, which includes low power consumption, flexibility, and high throughput. To do so, it is essential to develop the underlying architecture that holds the MapReduce worker on a Xilinx Zynq AP SoC.

With regards to design the proposed architecture, we develop the acceleration Kernels required, both in hardware and in software, that are used in the final solution. The MapReduce worker platform will integrate the main MapReduce functionality in a custom hardware solution while a software kernel is developed to manage the former.

Lastly, we perform the hardware-software integration between the hardware based acceleration cores and its software application, to both verify the system and to make parameter measurements and reporting its performance.

This thesis is concluded once the final system passes the verification and validation phases, as well as a thorough parameter evaluation phase to model its functional behaviour and performance.

1.3. Petitionary

This Master's Thesis is developed in the Industrial Systems and CAD (SICAD) Division of the Institute for Applied Microelectronics (IUMA), using a Zynq All Programmable SoC device as provided by the SICAD division, in order to design a MapReduce compatible platform and the necessary kernels to accelerate data processing techniques as used in the Big Data field.

1.4. Document structure

In this chapter, we present the necessary background for the reader to understand the purposes and objectives of this work, including a brief covering of basic concepts and statistics, from global IP traffic to the State of the Art in Big Data and MapReduce applications.

Chapter 2 concerns itself with the exposition of the concept of Big Data, Data Mining and the MapReduce programming model. In Chapter 3 we synthesize the employed hardware in this thesis, the Xilinx Zynq ZC706 Evaluation Kit, as well as the used High Level Design flow.

After introducing the hardware and the methodologies to produce our final solution, in Chapter 4 we cover the proposed solution for a MapReduce framework that is implemented on the presented Xilinx Zynq AP SoC. This includes the design constrains and objectives and the underlying architecture that it carries.

Chapter 5 includes the hardware platform design, from its underlying IP blocks to the platform modelling and result evaluation, diving then into Chapter 6, which covers the integration of the mentioned hardware platform design with a software application that uses this hardware accelerating core to its advantage in MapReduce applications.

Chapter 7 concerns the validation of the produced system, by analysing the performance of the MapReduce worker and its functionality. This document concludes with Chapter 8, by stating the overall results and characteristics of the produced system, a brief comparison with similar implementations and suggestions for improvement and future developments that could build upon this work or elements of it.

Chapter 2. BIG DATA

The primary goal of this chapter, is to characterize the concepts of Big Data and MapReduce. Firstly, we cover formal definitions for Big Data, its applications and challenges, and secondly, we relate these ideas to the MapReduce programming model, since a variation of this framework is used in this Master's Thesis.

2.1. DEFINITION

Large chunks of data that belong to IP traffic, are defined as Big Data and it represents vast amount of heterogeneous information, as defined by Sam Madden in [1]:

"Big data means too big, too fast, or too hard for existing tools to process."

Too large, because the data size orders of magnitude are around petabytes and exabytes, as previously stated, and the data itself is comprised of multiple sources; too fast, because the data must be processed on par with its continuous growth; and too complex since the problems that arrive through Big Data cannot be solved through conventional data processing techniques and conventional systems.

Another definition is described by Elgendy and Elragal in [24]:

> *"The term Big Data as the aggregate of large datasets that grow so large that they cannot be processed by traditional database management systems. They are datasets whose size is beyond the ability of commonly used software tools and storage systems to capture, store, manage, as well as process the data within a tolerable elapsed time."*

As discussed in section 1.1.1, and by Elgendy and Elragal, the big data sizes are in a constant growth ranging from a few terabytes (TB, 10^{12} bytes) to many petabytes (PB, 10^{15} bytes) of data in one single data set.

Big Data analytic processing, encompasses four critical requirements: fast data loading, fast query processing, efficient usage of storage space and strong adaptivity to highly dynamic workload patterns [24].

2.2. CHARACTERISTICS

In terms of Big Data characteristics we distinguish three 'Vs': volume, velocity, variety [1]:

- **Volume**, referring to the ever-increasing amount of data.
- **Velocity**, referring to the speed at which data traverses the network, considered from the data's creation, to its transmission, collection and classification.
- **Variety**, referring to the heterogeneous structure of large datasets, comprised of text, audio, images, videos and more unstructured data that can contain many other formats.

The concepts of volume, velocity and variety are not only the main characteristics of Big Data, as they represented the main challenges when talking about acquiring, organizing and analysing large datasets. But it is important to stress that variety of the datasets is what challenges Big Data solutions the most, as the heterogeneous characteristic of Big Data type datasets highlight the need for data classification and play a key role in the selection of the application [1], [24], [25]. Once these challenges have been addressed, the action of organizing, structuring, classifying and extracting useful data or patterns out of the dataset is what we call Data Mining [25].

2.3. DATA MINING CHALLENGES

To provide an efficient Data Mining solution, in this section, we must address multiple design challenges, as evidenced in [25].

Most times, Big Data datasets are found in a distributed memory architectures, which means that data transaction and storage should be taken into careful consideration. Thus, we say that data transaction is expensive, as it depletes time and power to move the datasets so that algorithms can operate on the data. This is an important discussion, since

most algorithms need most of the data to be stored previously on the main memory they operate on.

A second consideration is the privacy of user related datasets. Encrypted data is more than beneficial to the user base as it conceals private information but Big Data applications may not be able to operate on such encrypted data. For example, sensor network data may or may not be critical in privacy related terms, and can therefore be transmitted unencrypted and be easily processed by Big Data oriented solutions, as it does not necessarily represent a harmful threat to the user's privacy. But it is obvious that data such as a mobile user's location is, and might be therefore transmitted encrypted, rendering its content inaccessible for data mining.

Last, we must address the difficulties that arise when trying to extract useful information of a given dataset. When data arrives at a processing algorithm or framework, it finds itself in a raw format, unprocessed, incomplete and from multiple sources, and needs to be processed into subsets containing elements that share similar characteristics with each other, a criterion that might differ in between applications. The processed information must be mined to extract information and construct a model or solution, so that it can be finally tested to obtain feedback on the relevant information (such as the fusion techniques and criteria used in the first phase).

It is obvious that, while large data sets are analysed by multiple different applications and users for different purposes, the underlying system needs to be highly adaptive; MapReduce is a parallel programming model suitable for such a feat [24].

2.4. The MapReduce programming model

To facilitate the data fusion and processing stages, Google employs the already mentioned MapReduce programming model [7], as proposed by Dean and Ghemawat. This framework is a model and implementation for processing and generating large organized datasets. We stress that most of the information discussed in this section, is extracted from the original work done by Dean and Ghemawat in [7], as part of Google's effort to facilitate Big Data processing.

2.4.1. MapReduce dataflow

MapReduce concerns applications that are conceptually straight forward, like String Matching or Wordcount, but whose input data is unusually large and where computations must be distributed across of thousands of machines so that the computation time is reduced to a reasonable amount. To do so the authors conceptualized two main phases in dataset processing: the map phase and the reduce phase. By dividing the processing into multiple tasks, the parallelization endeavours can be abstracted and easily divided over multiple workers or machines.

The model itself takes a set of input key-value pairs (name and content of a file) and produces a set of output key-value pairs (the organized content and its values). The mapping function takes an input pair, where the key is the name of the string to be processed and the value is its content, and produces so called intermediate key-value pairs that are sent to the reduce function, which merges all the Key-Values together who share the same value. One of the main elements to this process is the fact that the intermediate values are supplied to the reduce function through an iterator, which allows for computation of lists that are too large to fit into local memory.

This structured dataflow allows Map invocations to be executed on different systems to produce splits out of a single dataset by using a simple and scalable solution, as shown in Figure 11.

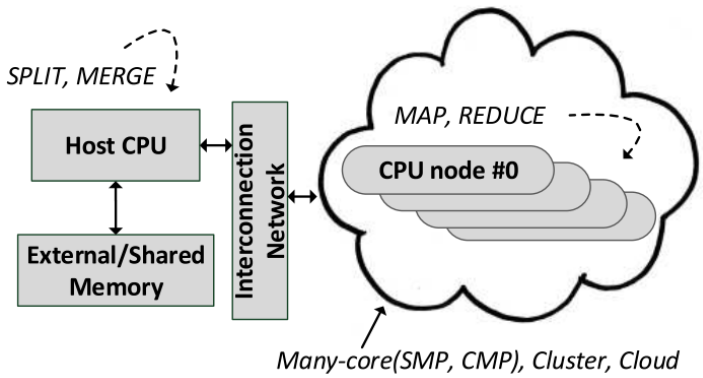

Figure 11. – Topology of classical MapReduce framework *[6]*

A typical MapReduce framework may use a split and merge function, although executed on the host CPU, whilst a CPU node takes over assigned Map and Reduce functions.

2.4.2. Architecture

We take a closer look at the MapReduce architecture by analysing what's inside a typical MapReduce worker, in Figure 12.

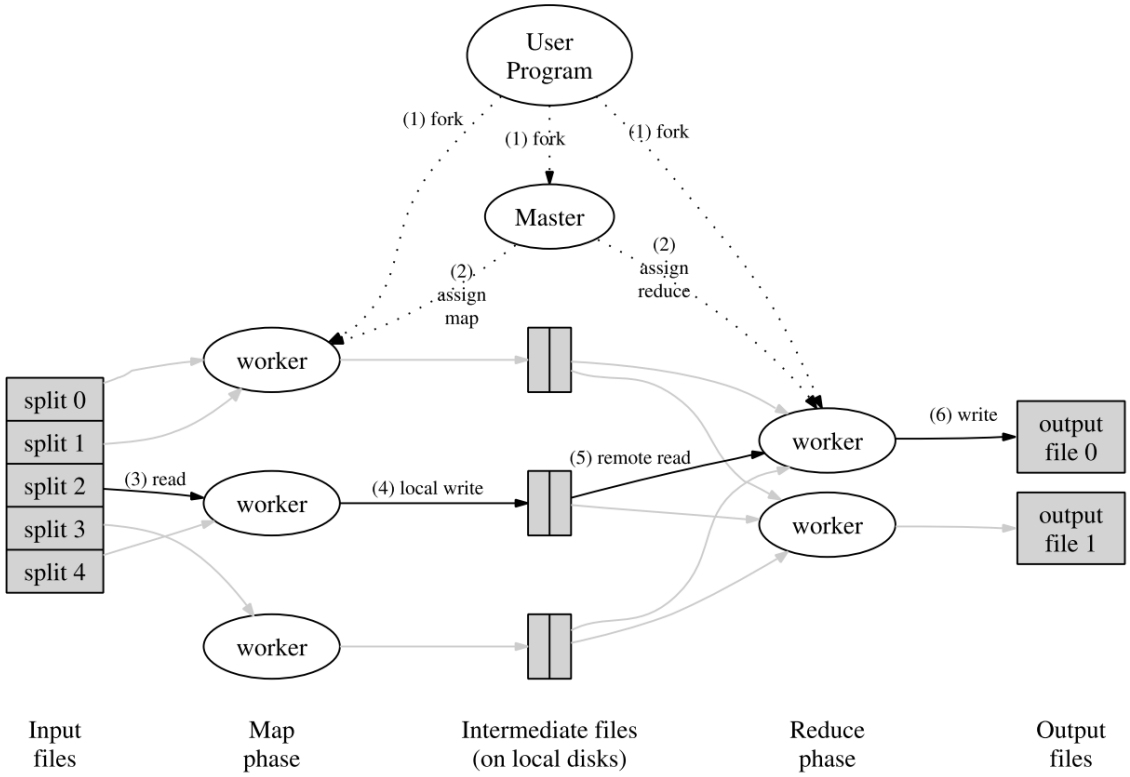

Figure 12. – Original MapReduce architecture proposed by Dean and Ghemawat *[7]*

As we can see, the input data is divided into multiple splits to be processed in a Map phase. The intermediate solution is stored on disk and given to a Reduce worker. The output is stored into an output file, ready for post-processing.

The original MapReduce library splits the input files into chunks of 16 to 64 Megabytes (MB) and starts up a master worker that supervises multiple slave workers on a cluster of machines that execute the Map and the Reduce tasks.

Each of the machines that perform a Map task, may also employ a by-user-specified combiner function that merges multiple key-value pairs before sending them to a reduce function. A combiner is used to do partial combining before applying a Reduce task and avoid throughput dampening when finding a large amount of key repetitions. As also explained in [7], the main difference between the combiner and the reduce function is the output it produces: the combiner produces an intermediate output to a Reduce worker while the reduce function itself produces an output file on the disk.

The MapReduce library is designed to work on large amounts of data and uses therefore fault tolerant techniques. To do so, the master worker pings every worker periodically; if no response is given after a certain time is elapsed, the Map or Reduce worker is reset and becomes eligible for rescheduling.

In conclusion, the MapReduce framework is an easy to use, fault-tolerant, locality optimized and load balancing solution. As pointed out by the authors, a large variety of problems are easily expressed as a MapReduce computation, allowing this framework to be used in a broad range of applications, such as in Google's case data sorting, data mining or machine learning.

MapReduce applications have a simple structure, and are defined by the Map and the Reduce main function bodies. The Map function comprises the main translating functionality, while the Reduce function includes the reduce criteria. Such applications include Histogram (image processing), Matrix Multiplication, String Match, Linear Regression, PCA, Kmeans and Word Count applications.

2.5. CONCLUSION

Big Data traffic processing and Data Mining tasks require solutions to produce memory, latency and throughput efficient results, that ordinary systems cannot conceive. Big Data itself can be defined by its volume, velocity and variety. Volume, due to the sizes of the datasets; velocity, as the data needs to be processed on par of it being received and sent to the next node; variety, since the datasets are heterogeneous and comprised of multiple different sources, such as audio, video, text and any other unstructured data.

A processing framework for Big Data is the MapReduce programming model, which takes up the task of translating (mapping) the data for it to be processed by simple criteria (reduced). The original MapReduce implementation is fault-tolerant, load balancing, memory optimized and easily programmable.

Chapter 3. XILINX ZYNQ ALL PROGRAMMABLE SOC AND HIGH-LEVEL DESIGN FLOW

In this chapter, we present the Xilinx Vivado design methodology that is used in this thesis, as provided by Xilinx Inc. We start by briefly describing the Xilinx Zynq-7000 AP SoC ZC706 Evaluation Kit used for this MapReduce implementation, its architecture and underlying elements

We follow up with the employed design flow to develop C/C++ based IP blocks, hardware platform creation and its use on FPGA's, which can be found in the Xilinx High-Level Design Methodology. To do so, we use Xilinx Vivado for hardware platform design, Vivado HLS for custom IP block design and Xilinx SDK for hardware-software integration and application definition. The latter is also used for final validation and platform parameter measurements.

We also provide the global design flow, from the most basic IP block and platform design to its final validation phase.

3.1. XILINX ZYNQ-7000 ALL PROGRAMMABLE SOC

As every AP SoC out of the Xilinx Zynq-7000 series, this type of SoC is comprised of two main architectural blocks: Processing System and the Programmable Logic. The PS is bases on two ARM Cortex-A9 MPCores which works up to 1 GHz (dependant on the model) [26], while its PL is based on the 7 series FPGA technology, designed with CMOS technology of 28 nm. We also find multiple transceptors that allow for a large variety of flexible solutions [15], [27], [28]. The described architecture can be seen in Figure 13.

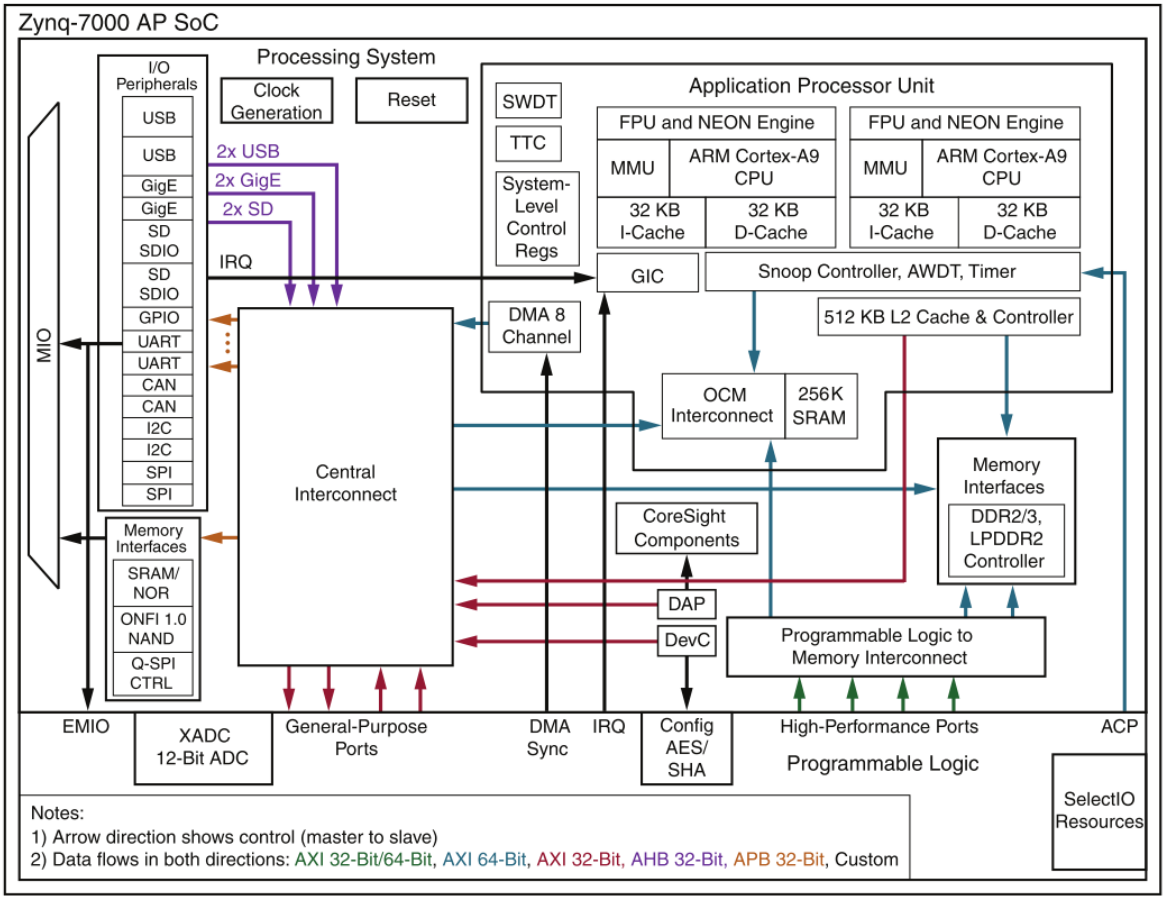

Figure 13. - Zynq 7000 Block Diagram [27]

As shown in the previous figure, the PS comprises four main blocks [27]:

- Application Processor Unit (APU).
- Memory interfaces.
- I/O Peripherals (IOP).
- Multilayered ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI), or AMBA AXI interfaces for short.

The Zynq SoC owns a variety of IOP's, such as Universal Asynchronous Receiver/Transmitter (UART), Controller Area Network (CAN), Gigabit Ethernet (GigE), Universal Serial Bus (USB), Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI) or General Purpose Input-Output (GPIO). Some of these peripherals use a Direct Memory Access (DMA) to allow for high speed applications.

3.1.1. The ZC706 Evaluation Kit

We use the Zynq-7000 AP SoC ZC706 Evaluation Kit, as shown in Figure 14.

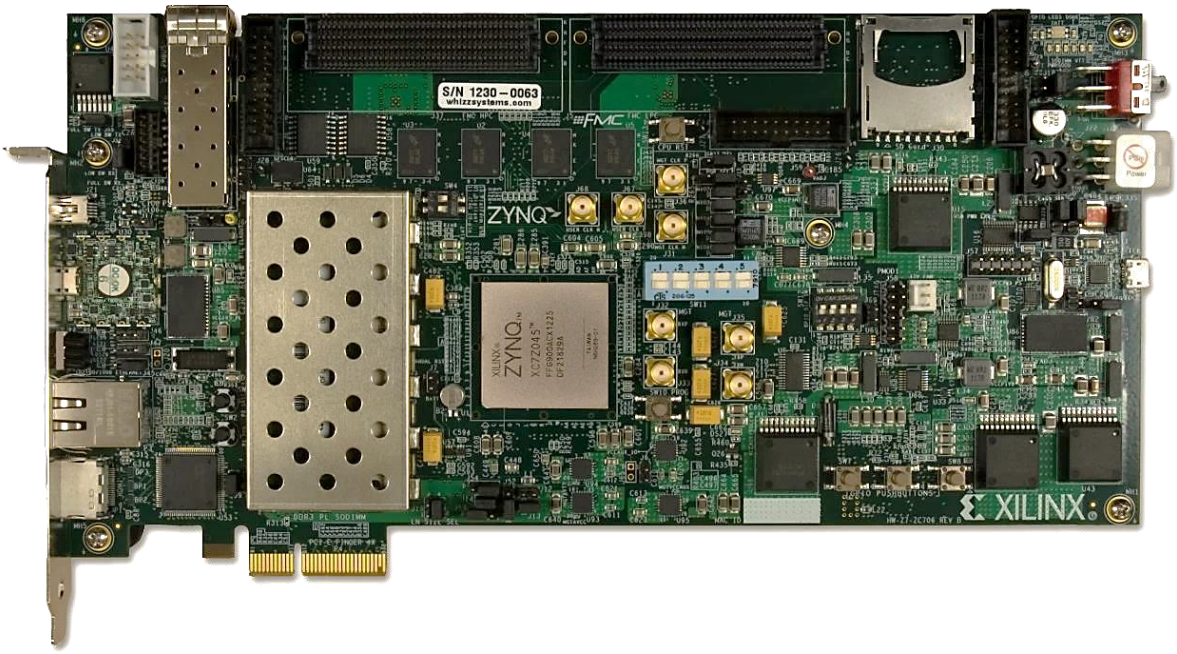

Figure 14. – ZC706 Evaluation Board [15]

Similar as indicated in Figure 13, but summarized, the XC7Z045 AP SoC can be described as follows in Figure 15 [27].

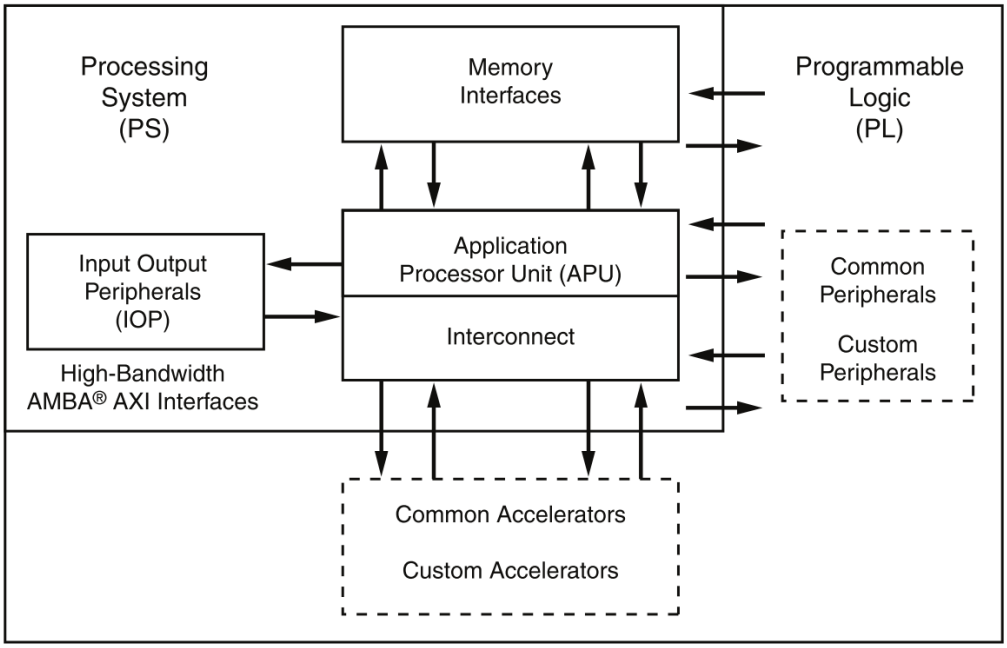

Figure 15. – ZC706 High-Level Block Diagram [27]

The PS section of the SoC, integrates a dual-core ARM Cortex-A9 MPCore that connects to IOP's, custom accelerators and peripherals (that reside in the PL section)

through AMBA AXI interfaces; the PS boots at power-up or reset and runs separately to the PL [27].

3.1.2. Processing System (PS)

The APU integrates a dual-core ARM Cortex-A9, which works at $667\ \mathrm{MHz}$ on the commercial model, $800\ \mathrm{MHz}$ on the extended model and up to $1\ \mathrm{GHz}$ on the industrial model [26]. In this thesis, we use the extended model (XC7Z045-2FFG900I), which works up to $800\ \mathrm{MHz}$.

As extracted from [26], we also find an Accelerator Coherency Port (ACP) which allows for coherent PL to CPU memory space access; the Level 1 cache (L1) employs 32 KB (up to 32 KB per instruction) whilst the Level 2 cache (L2) holds 512 KB in total; both CPU and PL have access to on-chip Random Access Memory (RAM) of 256 KB, although it is designed for low latency access from the CPU.

In terms of interfaces, the ZC706 offers [26], [27]:

- 2x 10/100/1000 tri-mode Ethernet MAC (TEMAC) peripherals.
- 2x UART.
- 2x CAN.
- 2x I2C.
- 2x SPI
- 4x 32b GPIO.

One of the most important features of this SoC, is the 8-channel DMA that supports easy and high throughput transfers between memory and peripherals. 4 of those channels are dedicated solely to the PL communication. Additionally, some of the interfaces work with built-in DMA, which include both USB interfaces, 2x Tri-mode Gigabit Ethernet and 2x Secure Digital Input Output (SDIO) [26].

Lastly, in terms of memory storage and interfaces, the SoC has 1 GB Double Data Rate 3 (DDR3) memory on both, the PL and the PS side, as well as two 128 Mb Quad-SPI flash memories [26]. The DDR is accessible through four separate 64 bit AXI slave ports: two dedicated to PL access, one to the ARM CPU through the L2 cache and a shared AXI slave, employed by all of the AXI masters via the central interconnect [26].

3.1.3. Programmable Logic (PL)

The Zynq PL shares its characteristics with the 7 series FPGA's. Specifically, the ZC706 integrates the XC7Z045-2FFG900I device, which derives from the Kintex-7 FPGA [26], [27], containing:

- 218,600 Look Up Tables (LUTs).
- 437,200 Flip Flops (FF)s.
- 18 KB per Block Ram (BRAM), with a total of 1,090 BRAM's (19.2 Mb).
- 900 Digital Signal Processing (DSP) slices.

3.1.4. PS-PL Communication

To communicate software applications that reside in the PS section of the Zynq with custom hardware of the PL, the Zynq SoC uses ARM AMBA AXI4. The usage of AXI interfaces allows for easy IP integration and connection of custom logic solutions with PS managed software ACP interfaces can be located as well. The PS-PL interconnection can be seen in both, the previous Figure 13 and the following Figure 16.

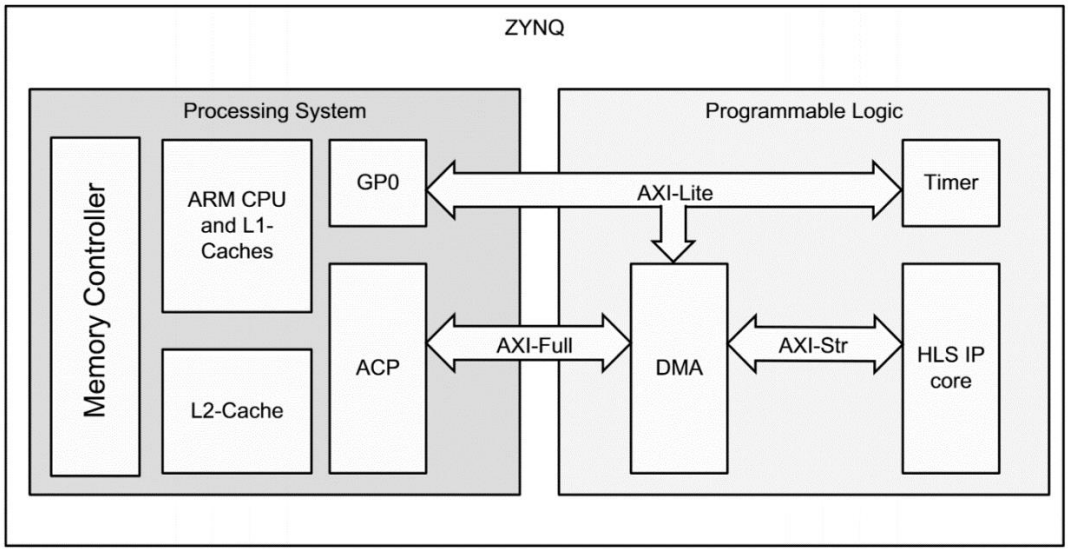

Figure 16. – Zynq device simplified architecture [14]

By using AMBA AXI4 as the primary interface between PS and PL, we can create flexible high-speed solutions with smooth and scalable interconnectivity. To transfer data between memory and custom IP cores located in the PL, we use AXI-Stream interfaces connected to a DMA, that connects back to the PS.

The primary AMBA AXI interfaces are [26]:

- 2x AXI 32b Master and Slave ports.
- 4x AXI 64b/32b Memory ports.
- 1x AXI 64b ACP ports.
- 16 interrupts.

Additional interfaces include four PS clock outputs and four PS reset outputs, a Processor Configuration Access Port (PCAP), a Xilinx Analog to Digital Converter (XADC) and a Joint Test Action Group (JTAG) interface for debugging purposes [26].

Lastly, the ZC706 allows multiple different configuration options [27]:

- Quad-SPI flash memory for PS configuration.
- PS boot from SD card for PS configuration.
- PL configuration through USB JTAG.
- PL configuration through an additional cable header and flying lead header JTAG configuration ports.

3.1.5. Zynq-7000 Memory Map

The Zynq-7000 series supports a total of 4 GB of address space, organized as presented in Figure 17.

Start Address	Size (MB)	Description
0x0000_0000	1,024	DDR DRAM and on-chip memory (OCM)
0x4000_0000	1,024	PL AXI slave port #0
0x8000_0000	1,024	PL AXI slave port #1
0xE000_0000	256	IOP devices
0xF000_0000	128	Reserved
0xF800_0000	32	Programmable registers access via AMBA APB bus
0xFA00_0000	32	Reserved
0xFC00_0000	64 MB - 256 KB	Quad-SPI linear address base address (except top 256 KB which is in OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

Figure 17. – Zynq-7000 Memory Map [26]

Addresses 0x4000_0000 and 0x8000_0000 are reserved for AXI slave ports that allow for easy PL-PS communication and data transfer. Starting at address 0xE000_0000 we find the already mentioned IOP devices. Addresses higher than 0xF000_0000 are reserved for other applications such as AMBA Advanced Peripheral Bus (APB) or Quad-SPI.

It is crucial to understand the Zynq address space about designing, modifying or debugging applications on a designed hardware platform.

3.2. XILINX HIGHLEVEL DESIGN FLOW

The tools and environments used in this work, such as the Vivado Design Suite, utilizes the High-Level Design Methodology, a set of practices put in place to describe a clear and straightforward design methodology for hardware-software solutions on AP SoC's and devices [28], [29]. In short, it describes a design flow that guides the designer through multiple design rules and methods that ease up the entire process and tries to achieve the design goals as quickly and efficiently as possible [29].

3.2.1. Xilinx Vivado and SDK

Both Xilinx Vivado and Xilinx Vivado HLS are comprised in the Vivado Design Suite, and employ the mentioned methodology, which proposes a design flow that can be summarised in Figure 18.

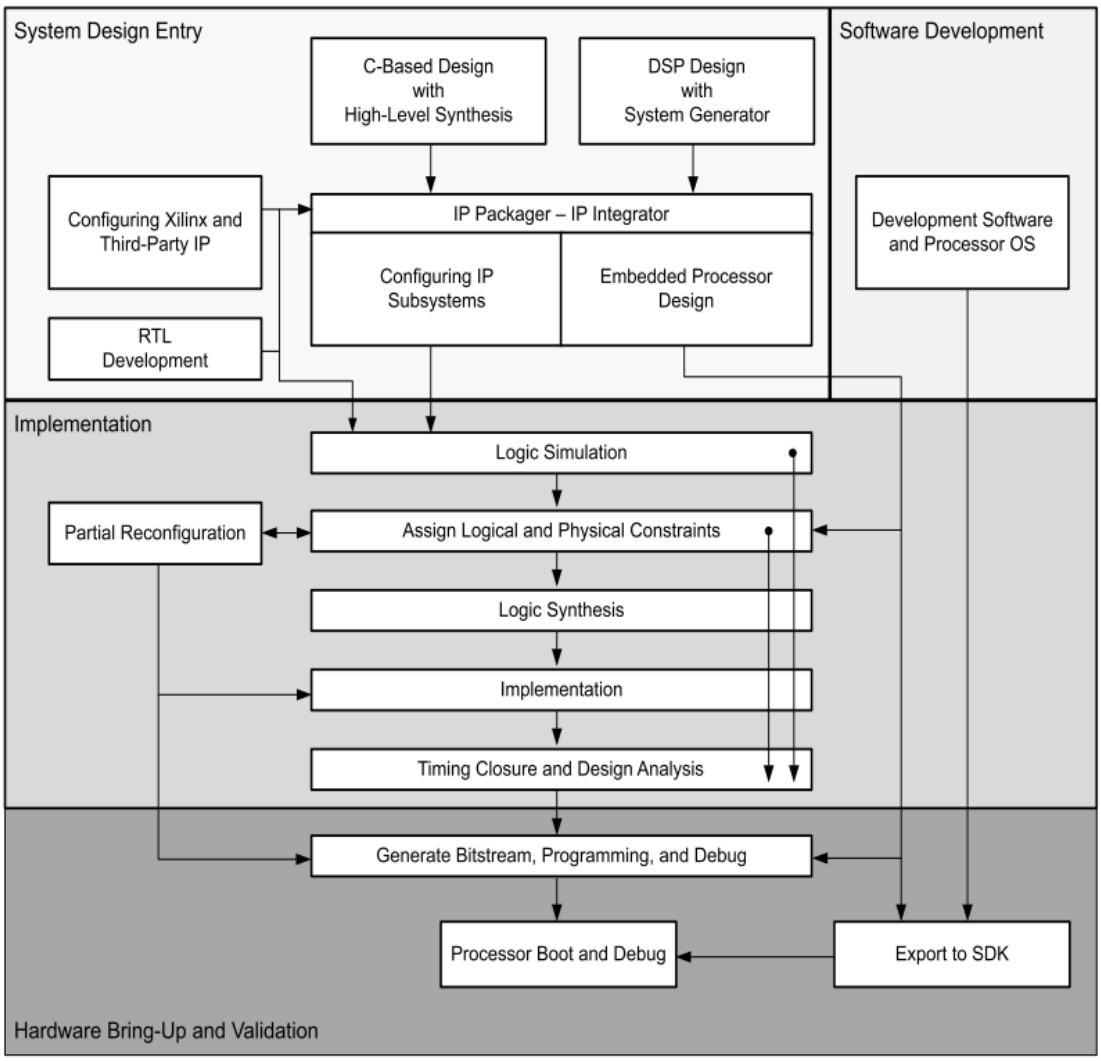

Figure 18. – Vivado Design Suite High-Level Design Flow [30]

Vivado employs a platform design process based on IP blocks, which are extracted out of Xilinx's IP catalog. The mentioned IP blocks can be either custom designed (C-Based design with High-Level Synthesis) or a standard Xilinx IP, if the interfacing between these blocks happens with Vivado supported interfaces or protocols, such as AMBA AXI.

Once the system description is complete, its RTL is synthesized maintaining user defined constraints into a netlist that allows the placement of the platform onto the desired AP SoC, which is the implementation phase and the final stage of the Vivado based platform design.

The hardware platform is given an application by using the Xilinx Software Development Kit (SDK). In this SDK, the user instantiates the designed hardware interfaces and creates a Board Support Package (BSP) that communicates the hardware with the

chosen SoC. The SDK is used to define the software that ultimately governs the instantiated hardware, manage the processor and its busses, the clock, reset and interrupt signals and initialize the RAM settings. Conclusively, the SDK defines the application that runs the entire system.

Verification, debugging and performance measurements are executed by using the SDK in conjunction with the Vivado implementation results. The design flow concludes once the systems behaviour and performance is validated.

3.2.2. Xilinx Vivado HLS

Vivado HLS main task is the translation, optimization and implementation of C/C++ or SystemC algorithmic description [31] (Figure 19).

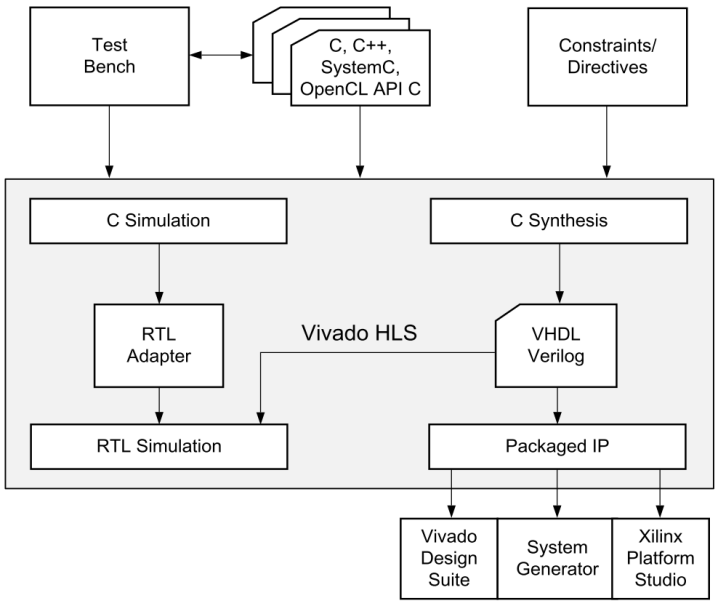

Figure 19. – Vivado HLS Design Flow [31]

The Vivado HLS inputs are of three types:

- the behavioural description (in C/C++, SystemC or OpenCL),
- the testbench that evaluates the behavioural code and
- a constraint or directives file that tunes and optimizes the design.

The design flow converts these inputs into a functional IP block. The algorithmic description is translated into Register Transfer Level (RTL) language through C Synthesis and its behaviour co-simulated with the source code. If both, synthesis and co-simulation results are satisfactory, by complying with latency, throughput and utilization constraints

for the selected SoC, the IP block is implemented and included in the Xilinx IP catalog, ready to be used in the Vivado Design Suite for platform design.

3.3. Conclusion

This thesis employs the Xilinx Zynq-7000 AP SoC ZC706 Evaluation Kit, which integrates a 7-series FPGA with two ARM Cortex-A9 MPCores at $800\ \mathrm{MHz}$. The ZC706 device allows for custom hardware IP solutions on the PL side integrated with a custom software solution on the PS side, in conjunction with a large variety of peripherals and interfaces that allow for high-performance solutions in a large variety of fields.

By using hardware provided by Xilinx, we are able to apply the Xilinx High-Level Design Flow, that allows for C/C++ and SystemC algorithmic descriptions to be implemented as hardware IP's on our device. This design flow's core is the Vivado Design Suite, that ranges from the algorithmic description, to the IP modelling, hardware platform design and the hardware-software integration.

Chapter 4. PROPOSED SOLUTION

Now that the MapReduce architecture has been presented in Chapter 2, as well as the Xilinx Zynq AP SoC in Chapter 3, in this chapter we introduce the reader to the proposed solution, which defines a MapReduce worker framework on a Xilinx Zynq AP SoC.

First, we cover multiple initial design considerations to understand the design space constraints and limitations. Then, we define the global dataflow architecture to later go into the concrete definition of the multiple blocks that confine the whole solution.

4.1. INITIAL CONSTRAINTS AND CONSIDERATIONS

Before any solution proposition can be explained, multiple design choices need to be made.

First and foremost, the hereby presented solution focuses on producing a simple MapReduce worker, based on a typical MapReduce framework. This worker contains multiple entities, such as a Map and a Reduce function, although it is evident that multiple instantiations of both are allowed.

The main objective is to set the groundwork for a generic high-performance framework, as it is MapReduce, for FPGA based SoC environments. By doing so, we create a framework whose application can be substituted with any other desired functionality, and therefore change the area of interest in which this thesis is employed.

We chose to define a MR worker that applies a Word Count application, since it is simple and therefore shortens design time for the IP blocks that ultimately shape the accelerator core. A Word Count application takes a string input, such as a file, maps out the

words that exist in the file into Key-Value pairs and merges all the pairs that share the same Key by increasing the value counter, as presented in Figure 20.

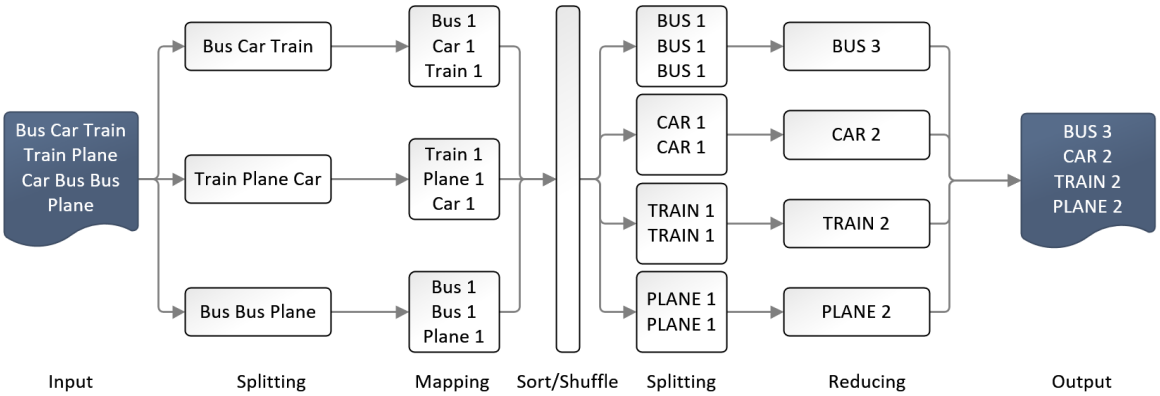

Figure 20. - Simple Word Count application using a MapReduce framework

With regards to memory considerations, since we are interested in the design of a simple MR worker whose parameters and performance are subsequently studied to justify whether it is an efficient solution, we concluded that the unformatted data must be handled by declaring it previously in memory, and sent to the accelerator through a DMA. The formatted data is then being given back by the same means, employing a DMA. The DMA itself is handled by interruptions and not by polling, since the latter may produce additional software overhead and therefore alter the latency of the accelerator core.

Regarding the interfaces of our IP blocks, we use AMBA AXI4 Stream for the data transfer interfaces, since it allows for custom width data transmission between IP blocks and easy protocol AXI protocol handling [26].

With respect to performance measurements, we evaluate the quality of the worker by its PL footprint and its computation latency and throughput.

4.2. MapReduce dataflow architecture

Regardless of its ultimate application, the proposed MR worker produces a given output by consuming an input file. Both those tasks require an initial splitting to happen and a final merging of the output products to produce the complete result. Combining these tasks with the map and reduce function, we propose an architecture such as shown in Figure 21.

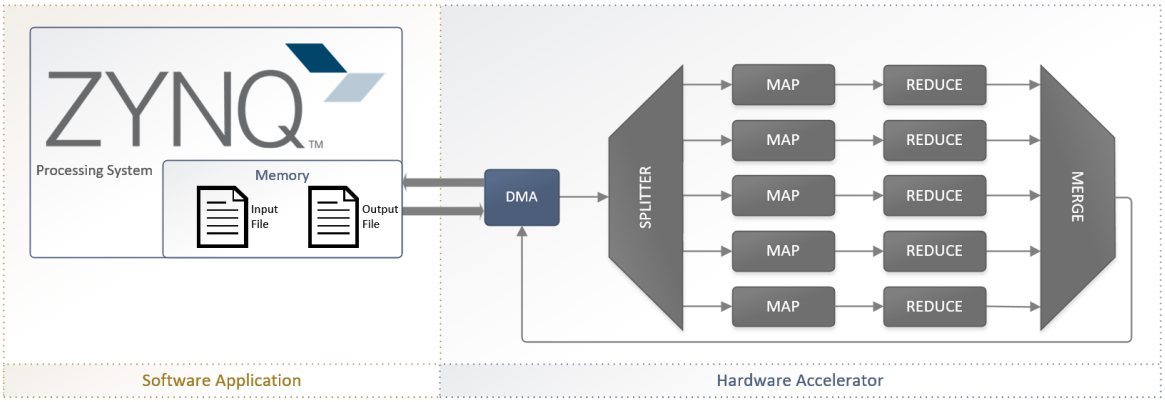

Figure 21. - MapReduce worker architecture - Proposition

The number of Map-Reduce pairs, which from now on are called a 'line' or 'MapReduce line', is arbitrarily chosen to be 8 at the beginning of its design. Although some implementations may favour implementing the Split and Merge function on the software side of the solution, to do so, it is required to redirect the Map and Reduce streams back to the PS side and into the memory, which requires a DMA access. Since this is a difficult task, as we have limited available streaming interfaces to go back from the PL to the PS, and software functionalities usually have lower throughput than their hardware counterpart, we decided to implement the Split and Merge stages as separate hardware IP blocks.

The proposed solution takes advantage of a dataflow architecture, where a splitting function divides the main task into multiple splits that are processed in a parallel fashion. Moreover, such a solution does not retain processed information until the entire task is done, but sends data to the next function whenever possible to reduce latency and increase throughput.

Out of all the presented functions (Split, Map, Reduce, Merge), only the Reduce function requires the data to be fully stored into memory to merge the KV pairs, whereas the Split, Map and Merge do not require to store data. By storing information in memory, we are severely throttling throughput and producing a higher memory footprint. For that reason, we are not concerned about the existing of a shuffle/sort stage between the Map and the Reduce function, as presented in a typical MapReduce framework [7], [17].

4.2.1. Split

The splitters main task is that of dividing the input file into roughly same sized splits so that the workload of each MR line is balanced. By using a Word Count application, the splitter must be cautious in dividing the incoming stream, since it is not allowed to partition words because it would destroy useful information. A splitting example can be found in Figure 22.

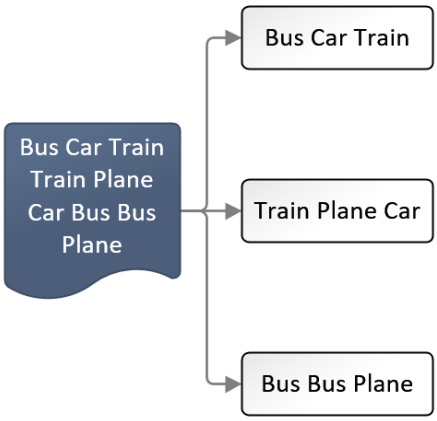

Figure 22. - Data splitting example

To produce a high-throughput implementation of a MR worker, the splitter must, not only divide the stream evenly over the available lines, but also send valuable information to a line as soon as possible. Thus, if we were to split 1 KB of data over 5 lines, the splitter is not allowed to maintain a line idle, as for example in Figure 23.

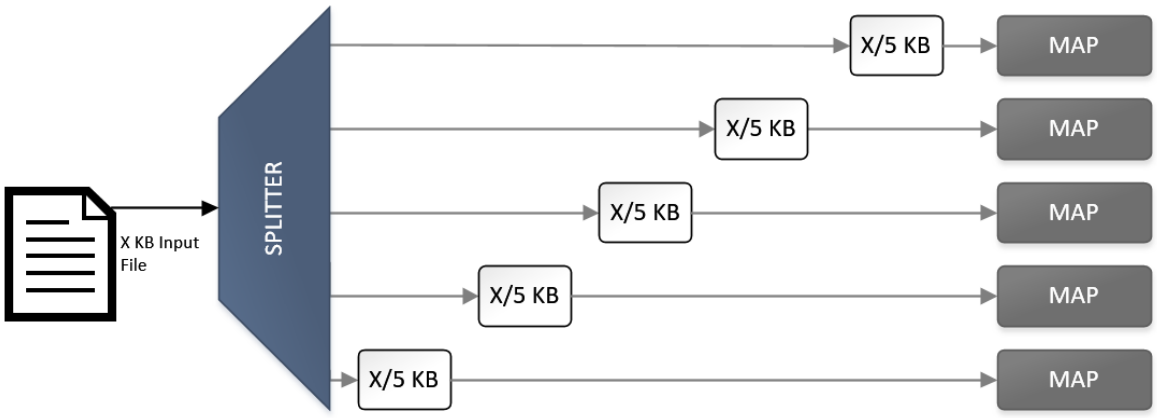

Figure 23. - Inefficient data splitting example

As we can see, while the splitter transmits a fifth of the total data, any other line remains idle, which is inefficient since one of the MR lines could potentially start mapping the output. We therefore chose a dataflow friendly approach, as presented in Figure 24.

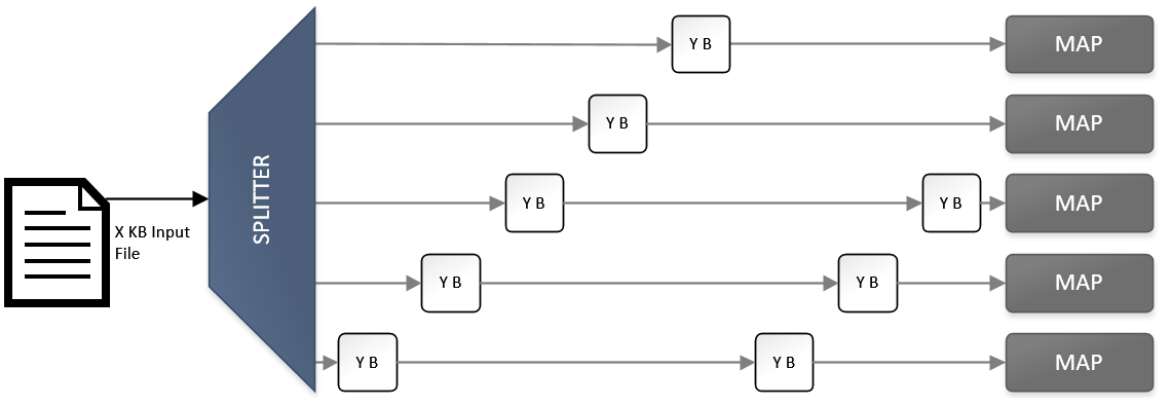

Figure 24. - Dataflow friendly data splitting

We propose a splitting function that divides the incoming data evenly over the given MR lines and sends valuable information to a line whenever data is ready to increase throughput. We do this by producing data splits of smaller proportions; i.e., for an input file of a given size X, the split function must swap interfaces once for every data split of size Y, where $Y \ll X$. By doing so, none of the Map-Reduce lines are idle during the splitting of the input data.

For our purposes, we arbitrarily chose the data to be split into 25 bytes' segments. This means, that the Split function partitions the data and sends it to the current Map function once a non-alphabetic character is found after crossing a 25-Byte threshold.

4.2.2. Map

The mappers key role is that of translating the incoming dataset into useful Key-Value pairs (Figure 25).

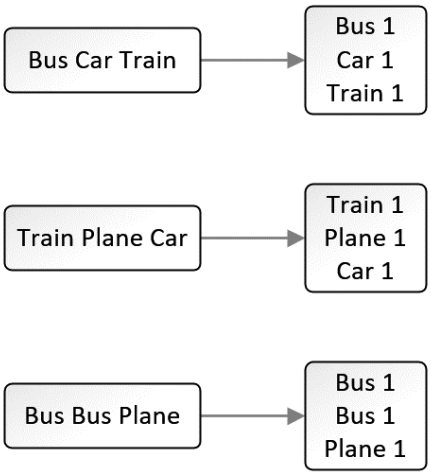

Figure 25. - Map example

Some MapReduce implementations, introduce a combiner function to their designs to merge KV pairs in earlier stages, such as presented in [17], [18], [20] and the original MapReduce proposal [7]. To simplify each of the functions main body and boost its efficiency, we decided not to include such a merging function and leave it to the reduce phase to complete merge Key-Value pairs.

4.2.3. Reduce

The Reduce function, concerns itself with comparing each value of an incoming KV pair. Reduce functions can have multiple criterions to do so, such as only merging all the data or only those with a previously given Key. For our case, we decided to make a Word Count application that merges all the data by a basic criterion of word repetition (Figure 26).

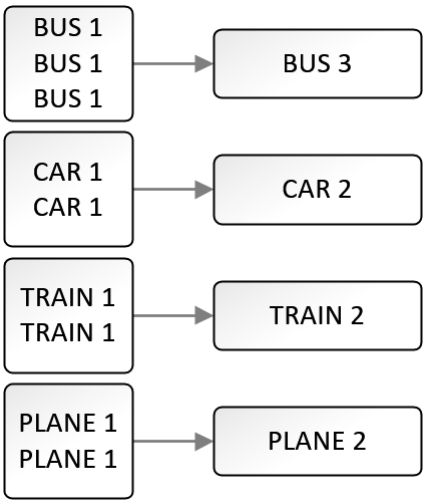

Figure 26. - Reduce example

The output of the reduce phase is a dataset comprised of KV pairs that do not have Key repetitions and are ready to be merged into an output file or stream.

4.2.4. Merge

The merging phase is the last of the MapReduce stages, and combines all the lines' output to create the complete result of the Word Count task (Figure 27).

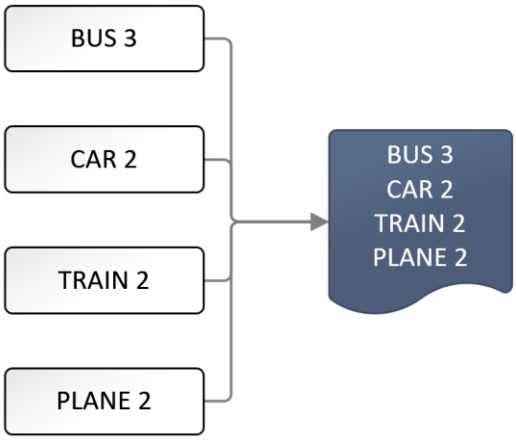

Figure 27. - Merge example

The output stream of the Merge phase is copied into a file for later analytic use.

4.3. STREAMING INTERFACES CONSIDERATIONS

As, already mentioned, the proposed solution employs AMBA AXI4 interfaces to easily stream data between IP blocks. Although AXI4 provides the main protocol for transparent communication, there is a need to establish an understandable and easy protocol to share Key-Value pairs between and inside IP blocks. To do so we established a simple by-tab division of keys and values, as shown in Figure 28.

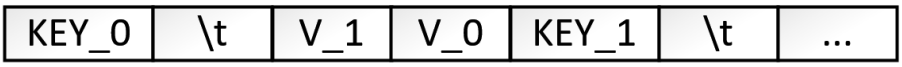

Figure 28. - Key-Value streaming protocol

After every Key (an english word) we find a horizontal tab character ('\t' or ASCII code 9), and the upper and lower part of its value. With this KV streaming protocol, functions can easily understand KV pairs throughout the whole worker, which is that, if in a given Key-Value stream a tab character is found, we are certainly at the end of a Key and at the start of both the Value bytes.

4.4. IP BLOCK DEPTH AND PRELIMINARY WORKER FOOTPRINT

The main MapReduce functions that are implemented in the presented worker must consider a given data input depth and width; depth, to employ large enough FIFOs (if needed) and width, as in data type width.

Since, with exception to the merge phase, all the other functions require a Byte-by-Byte processing to distinguish and compare words out of the input file, we decided that the streaming interfaces between all the IP blocks have an 8-bit width to eliminate any data type conversion derived latency. We do not hide that a larger standard width (64 bit for example) could increase throughput and overall system performance, but an 8-bit width stream throughout the whole design allows for easier and faster integration as data type conversion must not be made at any stage.

We decided to allow for a maximum of $8\ \mathrm{KB}$ to be introduced to a Split function, which limits the worker to a maximum work of $8\ \mathrm{KB}$. This means that the input stream related FIFO that will be inferred for the Split function, has a maximum depth of $8\ \mathrm{KB}$, and although we can send data into this FIFO unlimitedly (if the FIFO is being read to avoid overflow) subsequent IP blocks such as the Reduce IP, must have buffers and buffer depths previously allocated, as they need the data to be stored in memory to operate with it. If this wasn't the case, it would be possible to have an indefinite data stream through the whole MapReduce worker and therefore no data size limit on the proper.

The splitter function itself won't store information, therefore it should not have any memory footprint, since it passes information straight through by splitting the text in between two English words. Seeing that we have up to 8 MR lines, we split the 8 KB into 1 KB lines and merge them back together at the end.

The Map function won't have any memory footprint either, since does not register data, sends whole English words to the Reduce phase and it does not consider any character that is not alphabetic (commas, spaces, stops, etc.…). The map function has therefore an input depth of 1 KB as well as a 1 KB output depth.

The Reduce function requires a relatively large memory footprint. Starting off, by allowing 1 KB of data, the function needs to store at least 500 keys in the worst-case scenario, as well as additional storage for each value, which should have at least the length of a 16-bit integer to cover the worst-case possibilities (up to a value of 65,536). We also must consider that the KV protocol described in section 4.4 introduces additional information on top of the 1 KB data input stream. If we consider that 1 KB of data allows for a maximum of 500 1-Byte keys, and that for every Key + tab character (2 bytes) the

Reduce block introduces a 16-bit value (2 bytes), for every 1 Byte input key the output produces 2 additional bytes of valuable data (Figure 29).

Figure 29. - Key-Value streaming protocol width

Therefore, if 1 KB data produces 500 keys at most, the output data depth is of $2\ \mathrm{KB}$.

Lastly, if the Reduce function outputs a maximum of 2 KB of data, the Merge function must accept up to 8 lines of 2 KB and merge them together, outputting 16 KB of data.

In summary, the presented MapReduce worker accepts 8 KB input data on the Split side (8 streams of $1\ \mathrm{KB}$) and outputs up to 16 KB (8 streams of $2\ \mathrm{KB}$) of valuable data on the Merge side.

4.5. Conclusion

In this chapter, we present the proposal for the MapReduce based hardware accelerator. This includes the design constraints, the underlying IP blocks, interface considerations and final parameters.

The IP block designs represent the main MapReduce stages for a simple application, which are the data Split, Map, Reduce and Merge phases. While the Reduce IP block stores data in memory for its processing, the Split, Map and Merge blocks do not require data to be stored and have therefore a simple behavioural code.

With regards to the architecture, we chose AMBA AXI Stream interfaces and up to 8 Map-Reduce pairs (or lines). The interfaces' width is of 8 bits, as the IP blocks require Byte-by-Byte processing of alphabetic characters.

By interconnecting the different stages of the processing architecture, we create a so-called MapReduce worker, whose capacity has been chosen to be of $8\ \mathrm{KB}$, as the Reduce stage requires limited memory storage. With such an input depth, the worker produces $16\ \mathrm{KB}$ of valuable data.

Chapter 5. HARDWARE PLATFORM DESIGN

In this chapter, we dive into the hardware design process that follows the solution considerations of Chapter 4. As multiple design choices have been exposed in the previous chapter, we describe each designed IP block and its performance with respect to the already described functions. We also comment its main characteristics (I/O width and depth), it's HLS co-simulation latency and derived throughput and lastly its programmable logic footprint.

Since the IP blocks are considered custom logic solutions, we then follow with the hardware platform design where we combine our custom logic with standard Xilinx IP blocks to create a complete hardware design that connects the Processing System with our solution.

After each sub-section (custom IP design and platform design), we combine the derived results to easily understand the solutions product.

5.1. VIVADO HLS CONSIDERATIONS

Vivado HLS requires that an algorithmic description surpasses 4 design phases in order to create a functional IP block [32]:

- **C simulation**, which tests the C/C++ description by using a testbench designed by the user.
- Synthesis, or **High-Level Synthesis**, which translates C/C++ code into RTL and reports estimates of the maximum operational frequency of the IP block and its logic footprint in BRAM's, DSP's, FF's and LUT's. Since algorithmic

descriptions are translated into RTL code, we call this process High Level Synthesis.

In Vivado HLS, synthesis results are not reliable in most cases with respect to working frequency estimation, since the generated RTL has not been converted into a device-specific netlist yet and does not take the SoC's FPGA layout into consideration. Therefore, synthesis results may vary with respect to implementation results (maximum frequency) and should only be used to tune the algorithmic description and to optimize processing latency and logic footprint.

- **Co-simulation**, where an RTL wrapper is set on top of the C/C++ description to execute the testbench and compare both the software + RTL wrapper and the pure RTL description results. This phase reports back the maximum latency of the RTL description.

  As opposed to synthesis, where results may vary with respect to the final implementation phase, we have found that the co-simulation latency (expressed in cycles) agrees with the latency of the final implemented design.
- **Implementation**, which corresponds to the logic synthesis of the design. As opposed to High-Level Synthesis, this phase employs the RTL design and produces logical and physical transformations, transforming a logical netlist and constraints into a placed and routed design, ready for bitstream generation.

To make initial throughput measurements of each block (Vivado HLS does not provide such an estimation) with respect to the processed stream, we rely on co-simulation latency report (in cycles) and a maximum PL working frequency of $250\ \mathrm{MHz}$, which implies a $4\ \mathrm{ns}$ clock cycle.

The IP blocks main behavioural remains unchanged with respect to the proposed solutions, maintaining 8-bit wide AMBA AXI Stream interfaces.

In the following sections, we discuss IP block input and output depths, as we are utilizing data streams between blocks to communicate information. When an IP block is modelled in Vivado HLS with a stream at some of the communicating ends (input or output)

a FIFO is inferred at that end. We must then stress that, the so-called depth of the input or output refers to the depth of the inferred FIFO, which does not stop us from sending an amount of information that surpasses the FIFOs depth to the mentioned IP block to process it, if it is read. What does limit the IP blocks capacity is the size of the registering buffers that we have included to copy the information out of the stream into a register that allows us to process the data; in all the cases, the FIFO depth is dependent on modelled size for this registry.

All the modelled IP blocks have been designed considering that they are separated through external FIFOs. We must stress that, although it may seem redundant, this is an effort to decouple CLK signals and allowing for a dataflow friendly architecture and increase the maximum working frequency as it reduces fan-out.

5.2. SYNTHESIZABLE FUNCTIONS LIBRARY

Vivado HLS does not support some standard C/C++ constructs and functions, such as `tolower()`, `toupper()`, `isalnum()`, pointers to functions or recursive functions, as those constructs are either not understood by the high level synthesis tool or cannot be represented in RTL logic [31]. Since we also employ multiple functions to present accordingly the formatted text results of every IP block, we developed a simple *ad-hoc* synthesizable functions library, or *synthlib*, that contains synthesizable versions of standard functions, as well as additional functions that handle our data buffers.

Some of the contained functions are:

```
bool sl_isalnum(char arg);
bool sl_isdigit(char arg);
bool sl_isalpha(char arg);
char sl_tolower(char arg);
char sl_toupper(char arg);
void sl_strncpy(char arg_dest[DEPTH], char arg_src[DEPTH]);
void sl_axis_strncpy(ap_axis< WIDTH, 1, 1, 1> * arg_dest, char arg_src[W_DEPTH]);
bool sl_strncmp(char arg_a[DEPTH], char arg_b[DEPTH]);
int  sl_length(char arg[DEPTH]);
void sl_print(ap_uint<WIDTH> arg);
void sl_print(kv_pair * arg, int arg_length);
```

An example is shown by the argument of the `sl_isalpha()` function:

```
/**
 * @brief Returns if the presented character is alphabetic
 *
```

```
 *  @param arg the evaluated character
 *  @return 1 if the character is an uppercase or lowercase alphabetic
 *  character, 0 otherwise
 */
bool sl_isalpha(char arg) {
       return ((arg >= 65 && arg <= 90) || (arg >= 97 && arg <= 122));
}
```

By using a custom library for such simple tasks, we are solving two main problems:

- The synthesizability of standard functions.
- Function latency, as we are now in control of the behaviour of the function, and it allows us to optimize runtime latency of any given IP block.

We also simplified code by using pre-processor directives for common functionalities, as presented in the following code.

```
#define shr(x)                      x >>= 8
#define shl(x)                      x <<= 8

#define shr_sub(x,a)                shr(x); x - a
#define shr_add(x,a)                shr(x); x + a
#define shl_sub(x,a)                shl(x); x - a
#define shl_add(x,a)                shl(x); x + a

#define copy_shr(a,b)               a = b; shr(b)
#define copy_shl(a,b)               a = b; shl(b)

#define get_shr(a,b)                a = b >> 8
#define get_shl(a,b)                a = b << 8

#define break_if_list_end(a)        if (a >= list_size) break
#define break_if_depth_end(a)       if (a >= RED_IN_DEPTH) break
#define break_if_last()             if (in_last == 1) break
#define break_if_null(a)            if (a < 0) break
#define break_if_nl(a)              if (a == '\n') break
#definebreak_if_tab(a)              if (a == '\t') break

#define return_if_list_end(a)       if (a >= list_size) return
#define return_if_depth_end(a)      if (a >= RED_IN_DEPTH) return
#define return_if_last()            if (in_last == 1) return
#define return_if_null(a)           if (a < 0) return
#define return_if_nl(a)             if (a == '\n') return
#definereturn_if_tab(a)             if (a == '\t') return

#define read_key_at(i,a)            a = list[i].key
#define read_val_at(i,a)            a = list[i].val

#define assign_key_ptr_at(i,a)      a = &list[i].key
#define assign_val_ptr_at(i,a)      a = &list[i].val
```

Common functionalities include typical 8 bit shifts (`shr()` and `shl()`), breaking out of a section or scope or simple but recurring actions such as retrieving a value out of a struct (`read_key_at(i,a)`).

5.3. TUNEABLE PARAMETERS FILE

It is good practice, especially in developing large C/C++ solutions, to employ a common parameters file, that allows finetuning of the parameters of our designs. These include the width of AMBA AXI Stream signals, the depth of buffers, data type definitions, Key-Value constructs or global variables.

A snippet of our parameters file is presented as follows:

```
#define RED_IN_WIDTH         8      // Maximum Reduce input width
#define RED_IN_DEPTH         1000   // Maximum Reduce input depth

#define RED_OUT_WIDTH        8      // Maximum Reduce output width
#define RED_OUT_DEPTH        2500   // Maximum Reduce output depth

#define W_2                  1
#define W_4                  2
#define W_8                  3
#define W_16                 4
#define W_32                 5
#define W_64                 6
#define W_128                7
#define W_256                8
#define W_512                9
#define W_1K                 10
#define W_2K                 11
#define W_4K                 12

using namespace std;

struct kv_pair {
        ap_uint<AXIS_WIDTH> key;
        ap_uint<W_1K> val;

        kv_pair() {
                this->key = 0;
                this->val = DEFVAL;
        }
        void build(ap_uint<AXIS_WIDTH> key, int val) {
                this->key = key;
                this->val = val;
        }
        void build(ap_uint<AXIS_WIDTH> key) {
                this->key = key;
                this->val = DEFVAL;
        }
        void build(int val) {
                this->key = 0;
                this->val = val;
        }
};

extern int l_size;
extern kv_pair list[LIST_DEPTH];

extern int alph_indexes[26][LIST_DEPTH];
```

In the presented snippet of the high-level model, we defined the Reduce function interface width and depth, as well as the Key-Value construct that we are using throughout the whole design.

Using a common parameters file allows for fast and easy IP block modelling where common parameters must be shared, such as an AXI Stream width.

5.4. AMBA AXI4 STREAM DRIVER

During the development of this and previous theses, we have had multiple experiences using AMBA AXI4 Stream interfaces in C/C++ designs. Using this type of algorithmic description for our IP blocks, requires us to manage AXI Stream fields to communicate accordingly our custom IP's with standard Xilinx IP, as it is up to the designer to drive the signals.

The AXI Stream (or AXIS for short) waveform, follows in Figure 30.

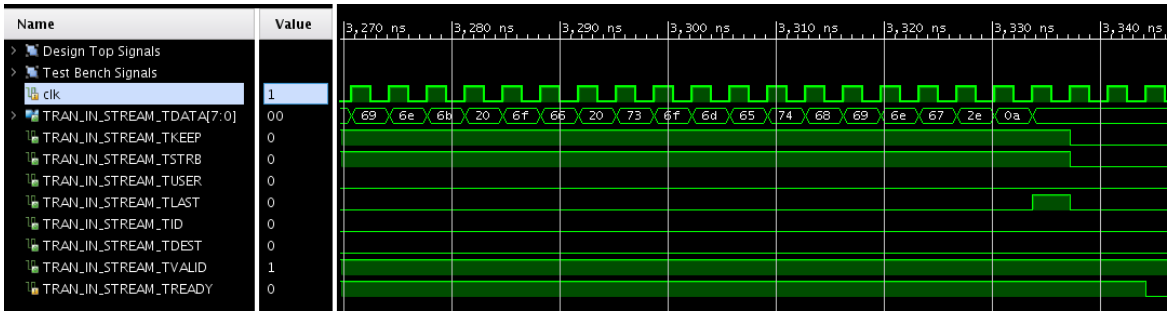

Figure 30. - Example of an AXI Stream waveform

The AXIS interface template is the following.

```
template<int D,int U,int TI,int TD>
  struct ap_axis{
    ap_int<D>        data;
    ap_uint<(D+7)/8> keep;
    ap_uint<(D+7)/8> strb;
    ap_uint<U>       user;
    ap_uint<1>       last;
    ap_uint<TI>      id;
    ap_uint<TD>      dest;
};
```

As defined in [33], by using a TDATA field of an 8-bit width, TKEEP has a width of 1 bit, which has to be asserted to indicate that the associated TDATA must be processed as

part of the data stream. If TKEEP is 0, the associated TDATA is considered null bytes, and can be removed from the data stream.

Lastly, with regards to the TLAST field, all our custom IP blocks employ an output AXI4 Stream. To indicate the streams boundary, the custom IP's send a terminating Byte (NULL) with the TLAST field asserted, indicating that the transmission has ended. We chose the last Byte to not carry information in the data field, as typical character strings are NULL terminated. By agreeing on a common usage of the TLAST field between IP blocks, we facilitate a transparent communication between IP's, making them less susceptible to errors.

Although the designer does not have access to the TREADY and TVALID signals, it is crucial to employ the TDATA, TKEEP and TLAST fields correctly to produce seamless communications between custom and standard IP blocks. To solve that problem, we wrote functions that assert and deassert the axis fields correctly.

First iterations of such a signal driver was written by using standard C/C++ function constructs. Despite it being functional, it produced function call overhead and increased co-simulation latency, which compelled us to re-write those functions as directives, as presented in the following snippet:

```
/** Input registers **/
ap_uint<WIDTH> in_data = 0;
ap_uint<1> in_keep = 0;
ap_uint<1> in_strb = 0;
ap_uint<1> in_user = 0;
ap_uint<1> in_last = 0;
ap_uint<1> in_id = 0;
ap_uint<1> in_dest = 0;

/** Output registers **/
ap_uint<WIDTH> out_data = 0;
ap_uint<1> out_keep = 1;
ap_uint<1> out_strb = 1;
ap_uint<1> out_user = 0;
ap_uint<1> out_last = 0;
ap_uint<1> out_id = 0;
ap_uint<1> out_dest = 0;

/** Stream read function **/
#define read_axis_unit(stream) \
( \
(in_data = stream->data, \
(in_keep = stream->keep, \
(in_strb = stream->strb, \
(in_user = stream->user, \
(in_last = stream->last, \
(in_id = stream->id, \
```

```
(in_dest = stream->dest, \
(stream++, \
(void)0 \
))))))))

/** Stream write function **/
#define write_axis_unit(arg, is_last, stream) \
( \
(out_data = arg, \
(out_keep = 1, \
(out_strb = 1, \
(out_user = 0, \
(out_last = is_last, \
(out_id = 0, \
(out_dest = 0, \
(stream->data = out_data, \
(stream->keep = out_keep, \
(stream->strb = out_strb, \
(stream->user = out_user, \
(stream->last = out_last, \
(stream->id = out_id, \
(stream->dest = out_dest, \
(stream++, \
(void)0 \
))))))))))))))))
```

It is essential to first declare the input and output registers, allowing the stream driver to function properly. By using such an approach, the designer must only provide a pointer to the axis streams on either the reading or writing side. Aside from the stream pointer, the output function requires to specify the TDATA and TLAST value.

We declared both read and write functions as `read_axis_unit()` and `write_axis_unit()`, as the designer might prefer reserving the names `read_axis()` or `write_axis()` for more complex functions, that read or write the whole stream. Such case is presented in the following Reduce IP block, where the output stream is sent without stopping.

5.5. IP BLOCK MODELLING

In this section, we discuss the Vivado HLS implementation for each of the main IP blocks: Split, Map, Reduce, Merge.

5.5.1. Split

The resulting Split IP block has 9 AXI Stream interfaces, 1 of which is the input stream and the other 8 are the output streams, such as presented in Figure 31. The input depth is of 8 KB and the output depth for each line is of 1 KB, with an I/O width of 8 bits on both ends.

Figure 31. - Split IP block

Its behavioural is shown in the following code.

```
ap_uint<2> wc_split(
ap_axis<SPLIT_IN_WIDTH, 1, 1, 1> * instream,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream0,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream1,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream2,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream3,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream4,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream5,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream6,
ap_axis<SPLIT_OUT_WIDTH, 1, 1, 1> * outstream7) {


	/** OMITTED: Variable declaration and initialization **/

	while (in_last != 1) {
		/** Passthrough outstream0 **/
		while (is_open_0 && in_last != 1) {
			over_threshold = i_split0 > cur_threshold;
			i_split0++;

			/** Read the instream and write it to the current outstream **/
			read_axis_unit(instream);
			write_axis_unit(in_data, 0, outstream0);
			out0++;

			/** OMITTED: Check if in_data was alphabetic **/

			if (over_threshold == 1 && in_isalpha == 0) {
				/** Break to allow for interface swapping **/
				i_split0 = 0;
				break;
			}

			if (out0 == SPLIT_OUT_LAST) {
				is_open_0 = 0;
				out0++;
			} else if (out0 >= CRITICAL_RANGE) cur_threshold = 1;
		}

		/** OMITTED: Passthrough oustream1 **/
		/** OMITTED: Passthrough oustream2 **/
		/** OMITTED: Passthrough oustream3 **/
		/** OMITTED: Passthrough oustream4 **/
		/** OMITTED: Passthrough oustream5 **/
		/** OMITTED: Passthrough oustream6 **/
		/** OMITTED: Passthrough oustream7 **/

		/** If all the interfaces are closed, end the operation **/
```

```
            if (!(is_open_0 && is_open_1 && is_open_2 && is_open_3 && is_open_4 &&
is_open_5 && is_open_6 && is_open_7)) break;
        }

        /** OMITTED: Send a NULL with TLAST toggled to all the outstreams **/
        return 0;
}
```

To efficiently divide the incoming stream into 8 MR channels, numbered from 0 to 7, the split IP block reads a certain number of bytes out of the in-stream and copies it to the first interface. Once a 25 bytes' threshold is crossed, the splitter checks whether a word has finished (the finding of a non-alphabetic character at that same position). If this is not the case, the splitter continues copying the stream to the output interface until an end is reached. If the latter is the case, the stream is switched to the next open interface, which is adjacent to the current (interface 0 swaps to interface 1, interface 1 swaps to interface 2, and so forth, while interface 7 jumps back to interface 0). Such a function flow is shown in Figure 32.

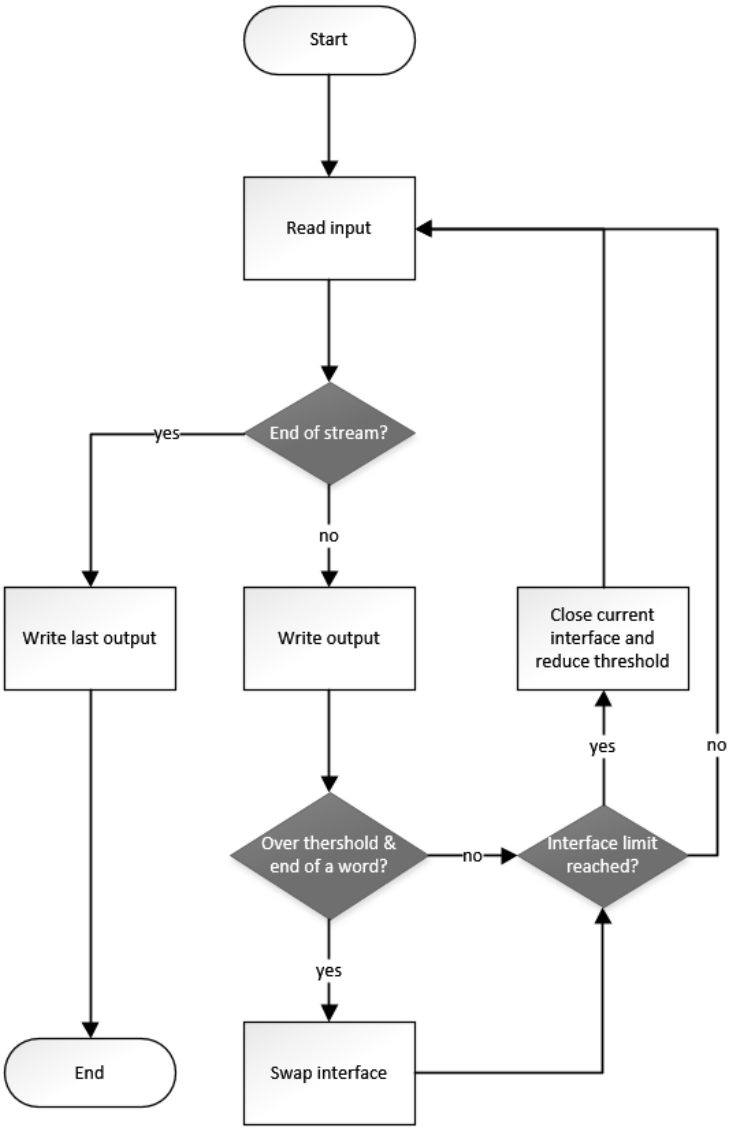

Figure 32. - Split IP block behavioural

Every MR line has a maximum depth of 1 KB. Thus, the split function keeps track of the number of bytes copied and limits its maximum to 999 bytes, such that, in the event of reaching the maximum line depth, the last Byte (1.000) is a NULL with the *last* AMBA AXI field driven high ('1'), to indicate the end of a stream. Regardless of how many data are copied into each stream, once the incoming stream finishes, the output interfaces send a NULL with the *last* field driven high.

The splitters threshold varies throughout the copying process: if an interface is closed, we understand that we are reaching the end of an 8 KB input and that one line is already saturated. As the end of the incoming stream approaches, to fit whole words into

the output interfaces, the splitter reduces the maximum Byte threshold from 25 to 1, which means that the stream is swapped after the ending of each word, and closed if its full.

5.5.2. Map

The Map IP block has 2 AXI Stream interfaces, 1 of which is the instream and the other is the output stream, as presented in Figure 33.

Figure 33. - Map IP block

Its behavioural is shown in the following code.

```
ap_uint<2> wc_map(
ap_axis<MAP_IN_WIDTH, 1, 1, 1> * instream,
ap_axis<MAP_OUT_WIDTH, 1, 1, 1> * outkey) {

	/** OMITTED: Variable declaration and initialization **/
	read_axis_unit(instream);

	in_data &= 0xdf;		// To upper
	in_isalpha = in_data >= 65 && in_data <= 90;

	while (1 < 2) {
		break_if_last();
		switch (state) {
			case IN_WORD:
				/** If in_data is alphabetic, send it. Otherwise, send a TAB
char **/
				if (in_isalpha == true) {
					write_axis_unit(in_data, 0, outkey);
				} else {
					state = NOT_IN_WORD;
					write_axis_unit('\t', 0, outkey);
				}
				break;

			case NOT_IN_WORD:
				/** If in_data is alphabetic, send it and swap state to
IN_WORD**/
				if (in_isalpha == true) {
					state = IN_WORD;
					write_axis_unit(in_data, 0, outkey);
				}
				break;

			default:
				break;
		}
```

```
                read_axis_unit(instream);

                in_data &= 0xdf;                    // To upper
                in_isalpha = in_data >= 65 && in_data <= 90;
        }

        /** Send a NULL with TLAST toggled **/
        write_axis_unit(0, 1, outkey);

        return 0;
}
```

The mapper function remains widely unchanged with respect to its original proposal. The IP block copies the input stream to the output, if it is an alphabetic character. If it is not, it sends a tab character ('\t', ASCII code 9) to denote the end of a key to the mapper, and iterates to the next alphabetic character. Such a workflow is presented in Figure 34.

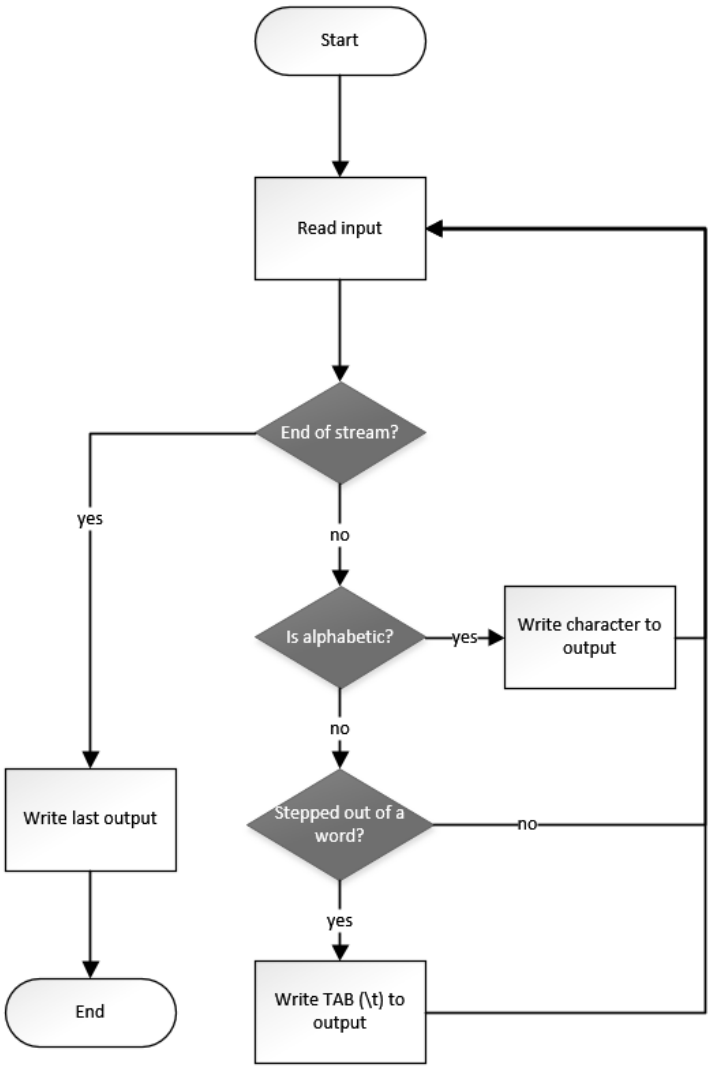

Figure 34. - Map IP block behavioural

Since the Map function does not employ any combiner (an element described in section 2.4 The MapReduce programming model), at this stage we are certain that every emitted key has a value of 1. This allows us to omit its sending to the reducer, since all the keys arrive with a value of 1. By doing this we spare communication latency, as we do not send any values to the next stage. The reduce function itself takes up the task of combining Key-Value pairs.

5.5.3. Reduce

The Reduce IP employs the same interface configuration as the Map block, as presented in Figure 35. The interface width is of 8-bit on both ends, with a 1 KB depth on the receiving end and 2 KB depth on the emitting end.

Figure 35. - Reduce IP block

Its behavioural is shown in the following code.

```
ap_uint<2> wc_reduce(
ap_axis<RED_IN_WIDTH, 1, 1, 1> * instream,
ap_axis<RED_OUT_WIDTH, 1, 1, 1> * outstream) {

	/** Variable declaration and initialization **/

	/** Clear the legal indexes length buffer **/
	for (i = 0; i < 26; i++)
		indl[i] = 0;

	while (1 < 2) {
		/** READ AXIS STREAM - START **/

		/** OMITTED: Reset counters and variables **/

		while (1 < 2) {
			i_arg_lower2 += 8;

			/** Read the incoming stream and break if TLAST is toggled or TDATA
is a TAB
				character (end of a key) **/
			read_axis_unit(instream);
			break_if_last();
			break_if_tab(in_data);

			/** Add the character to the complete word (key_tmp0) **/
			key_tmp0 |= in_data_tmp1;
			in_data_tmp1 = in_data;
```

```
                        in_data_tmp0 |= in_data_tmp1 << i_arg_lower0;
                        key_tmp0 = in_data_tmp0;

                        /** Advance the counter in a PIPELINE friendly fashion **/
                        i_arg_lower0 = i_arg_lower1;
                        i_arg_lower1 = i_arg_lower2;
                }
                key = key_tmp0;
                /** READ AXIS STREAM - END */

                /** I'm substracting 65 to the first and second character (-0x4141). This
way, we bring an 'A' down to the value 0, and 'Z' to 25 */
                break_if_last();

                key_tmp = key - 0x4141;

                char0 = (char) key_tmp;
                shr(key_tmp);
                char1 = (char) key_tmp;

                break_if_null(char0);

                /** If the second character is inexistent, we register the word's legal
index into             * position 25 **/
                if (char1 < 0) char1 = 25;

                index_ptr_pos0 = indexes[char1][0];
                index_ptr_pos1 = indexes[char1][1];
                index_ptr_pos2 = indexes[char1][2];
                ind_size = indl[char1];

                /** With the legal indexes retrieved, we start reading the registry's content
and             * search for the current key **/
                i3_0 = indexes[char1][i + 3];

                read_key_at(index_ptr_pos0, reg_key);
                read_val_at(index_ptr_pos0, reg_val);

                /** Increase the value register, to use it later, in
                /** Increase the value reg, to use it later in case the key is found **/
                reg_val++;

                /** Pre-register keys and values to ease list iteration **/
                assign_val_ptr_at(index_ptr_pos0, reg_new_val);

                assign_key_ptr_at(index_ptr_pos1, reg_key_ptr0);
                assign_val_ptr_at(index_ptr_pos1, reg_val_ptr0);

                assign_key_ptr_at(index_ptr_pos2, reg_key_ptr1);
                assign_val_ptr_at(index_ptr_pos2, reg_val_ptr1);

                assign_key_ptr_at(list_size, reg_new_key);

                for (; i < ind_size; i++) {
                        /** Read the legal section **/

                        i3_1 = indexes[char1][i + 4];

                        if (reg_key == key) {
                        /** If the key is found, assign its new value **/
                                result = 1;
                                (*reg_new_val) = reg_val;
                                break;
                        }
```

```
                        /** Advance all the pointers, counters and registers **/
                        reg_new_val = reg_val_ptr0;

                        reg_key = *reg_key_ptr0;
                        reg_val = *reg_val_ptr0 + 1;

                        reg_key_ptr0 = reg_key_ptr1;
                        reg_val_ptr0 = reg_val_ptr1;

                        assign_key_ptr_at(i3_0, reg_key_ptr1);
                        assign_val_ptr_at(i3_0, reg_val_ptr1);
                        i3_0 = i3_1;
                }

                if (result == 0) {
                        /** If we didn't find the key in the registry, we append it **/
                        *reg_new_key = key;
                        indexes[char1][ind_size] = list_size;
                        ind1[char1]++;
                        assign_key_ptr_at(list_size++, reg_new_key);
                }
        }

        /** WRITE AXIS OUTSTREAM - START **/

        /** OMITTED: Reset counters and variables */

        while (1 < 2) {
                /** Check if we're over the lists maximum size */
                break_if_list_end(list_id0);

                list_id2++;

                cur_key = 0;

                /** Read current key and value into a register **/
                read_key_at(list_id0, cur_key);
                read_val_at(list_id0, cur_val);

                /** Clear current list key and value **/
                clear_key_at(list_id0);
                clear_val_at(list_id0);

                /** Sepparate the value into two bytes, in case it's greater than 255 */
                cur_val_upper = cur_val.range(15, 8);
                cur_val_lower = cur_val.range(7, 0);

                /** Copy 1 Byte from the current word into a register and shift right **/

                copy_shr(arg0, cur_key);
                /** Copy 1 Byte from the current word into a register and shift right **/
                copy_shr(arg1, cur_key);

                while (1 < 2) {
                        if (arg0 == 0) break;
                        /** Copy 1 Byte and shift right **/
                        copy_shr(arg2, cur_key);
                        /** Send Byte **/
                        write_axis_unit(arg0, 0, outstream);
                        /** Advance pre-registered data into current register **/
                        advance_reg(arg0, arg1, arg2);
                }

                /** Write a TAB, the value and a VTAB to the outstream **/
                write_axis_unit('\t', 0, outstream);
                write_axis_unit(cur_val_upper, 0, outstream);
```

```
                write_axis_unit(cur_val_lower, 0, outstream);
                write_axis_unit('\v', 0, outstream);

                advance_reg(list_id0, list_id1, list_id2);
        }

        /** Send a NULL with TLAST toggled **/
        write_axis_unit(0, 1, outstream);

        return 0;
}
```

The Reduce function contains the main Word Count function. The IP block itself copies the incoming bytes into a new key register until it reaches a TAB character. Once a new key is registered, the memory is iterated to find if the IP block already holds the given key; if this turns out to be true, the value of that key is increased by 1; if not, the new key is appended to the KV list and its value is set to 1. Once the input stream sends its last Byte by toggling the *last* AMBA AXI field, the list is emitted through the output stream. Such a dataflow is presented in Figure 36.

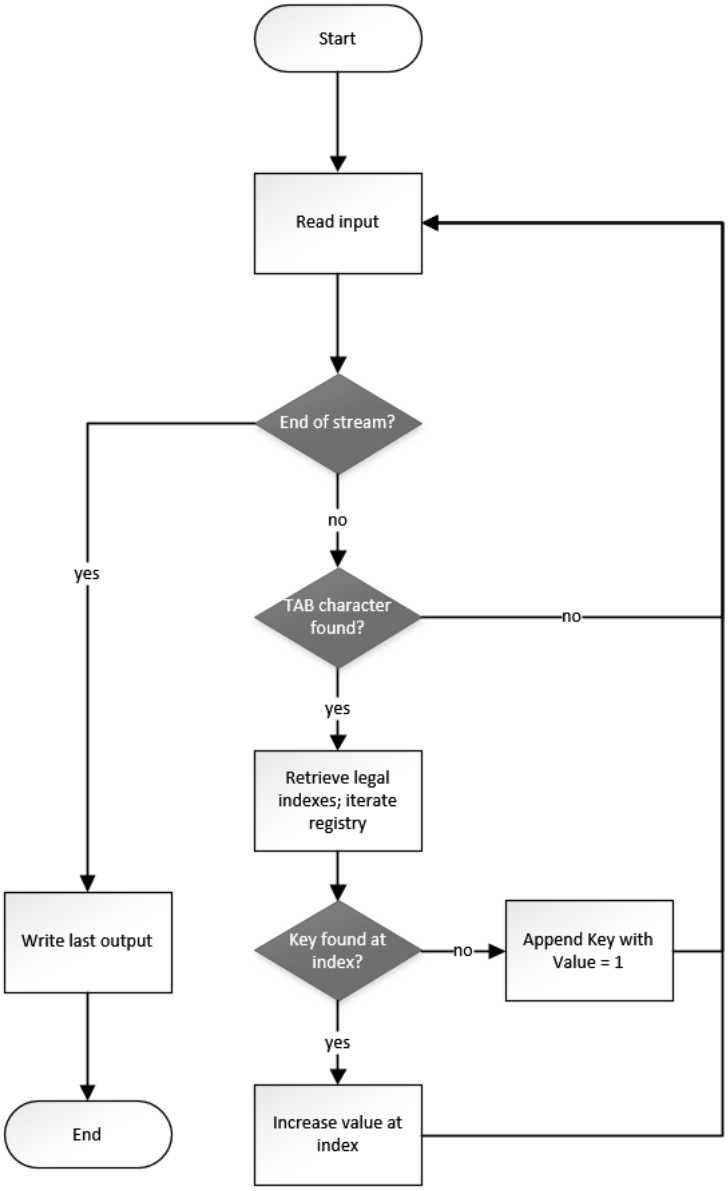

Figure 36. – Reduce IP block behavioural

This IP block is the bottleneck of the entire MapReduce application, since it requires to copy the entirety of the mapped keys into memory to properly increase the values, instead of continuing a dataflow behaviour, such as the Split or Map function.

To decrease latency, the Reduce function was designed such that it does not iterate the entire KV list to find a given Key, but it 'remembers' the position of all the keys that share a common characteristic with the current Key. By doing so, the function only iterates the so called 'legal indexes' that are allowed, which is a fraction of the total list length. The legal indexes are those that share that common characteristic with the input key.

The common characteristic was chosen to be the second letter of any given Key; if a Key is appended to the list, its second character and index is annotated for future lookup. Whenever a new Key arrives, the iterator only handles the indexes where stored Keys share the same second character. The second letter itself is used as an index of the mentioned legal indexes buffer which is used as a Content Addressable Memory (CAM).

The reason we chose the second character to be the defining characteristic, is deduced by considering that future MR implementations may introduce a Shuffle/Sort block in between mapping and reducing phases. By doing so, the Reduce block then receives Keys whose first characters may as well be the same, which would render our effort to reduce latency through common characteristic useless. Therefore, we chose the second character to be the deciding factor.

To fulfil the described task, the Reduce function contains multiple buffers:

- **Key-Value list:** the main KV registry, which has a length of 500 pairs for a 1 KB input, where a Key has a width of 30 bits and a Value has a width of 16 bits ($500\text{x}30\text{x}16$).
- **Legal indexes:** a $26\text{x}500$ bi-dimensional buffer containing the 'legal indexes' for a given key. For example, the word 'Apple' may be appended to the KV list while its current index is added to the 16th array of the 'legal index' buffer, which is position 15 ($\text{A}=0, \text{B}=1, [\ldots]\ \text{P}=15$). 1 letter words, such as the article 'A', are annotated into the 26th array, or index 25, which is reserved for the letter 'Z' but statistically underused.
- **Legal indexes length:** a $26\text{x}1$ buffer that contains the number of annotated legal indexes in the previous buffer, or the maximum iteration length. For example, if the registry contains 5 Keys whose second character is the letter 'P', this buffer reports back a length 5 for the 16th letter ($\text{A}=0, \text{B}=1, [\ldots]\ \text{P}=15$). This allows us to reset the legal indexes by setting all the lengths to 0, without the need to set the legal indexes buffer contents back to NULL. This spares a great amount of latency with each Reduce function call.

One of the greatest weaknesses of the Reduce block, is its need to copy the whole set of mapped Keys into memory to merge its values accordingly. This means that, as opposed to the other IP blocks, the Reduce block is the only one that does not send information in a dataflow friendly fashion; it starts emitting information once the last Key is registered into memory.

Lastly, this IP block is the most memory and logic consuming design, as it contains multiple KV buffers to work at high throughputs.

5.5.4. Merge

Conversely to the Split IP block, the Merge block takes 8 AXI Stream input interfaces and merges them into 1 AXI output stream interface, as shown in Figure 37. The stream width is of 8-bit on both ends, with an input depth of 2 KB and an output depth of 16 KB.

Figure 37. - Merge IP block

Its behavioural model is shown in the following code.

```
ap_uint<2> wc_merge(
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream0,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream1,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream2,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream3,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream4,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream5,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream6,
ap_axis<MERGE_IN_WIDTH, 1, 1, 1> * instream7,
ap_axis<MERGE_OUT_WIDTH, 1, 1, 1> * outstream) {

	/** OMITTED: variable declaration and initialization **/

	/** Passthrough instream0 **/
	/** Read first element of current instream**/
	read_axis_unit(instream0);
	tmp_data = in_data;
	while (in_last != 1) {
		/** Write current data to output**/
		write_axis_unit(tmp_data, 0, outstream);
		/** Read next element of current instream **/
		read_axis_unit(instream0);
		tmp_data = in_data;
```

```
        }

        /** OMITTED: Passthrough instream1 **/
        /** OMITTED: Passthrough instream2 **/
        /** OMITTED: Passthrough instream3 **/
        /** OMITTED: Passthrough instream4 **/
        /** OMITTED: Passthrough instream5 **/
        /** OMITTED: Passthrough instream6 **/
        /** OMITTED: Passthrough instream7 **/

        /** Send a NULL with TLAST toggled **/
        write_axis_unit(0, 1, outstream);

        return 0;
}
```

The Merge IP block is the last of the MapReduce worker. In view of the Reduce blocks not being able to send information straight away, but retaining it until the last Key is registered, the Merge block does not need to swap between streams like the splitter function: finding that a MR line is ready for transmitting its whole KV registry, means that the Reduce block has finished and we can send the complete line's result before swapping to the next. The described functionality can be found in Figure 38.

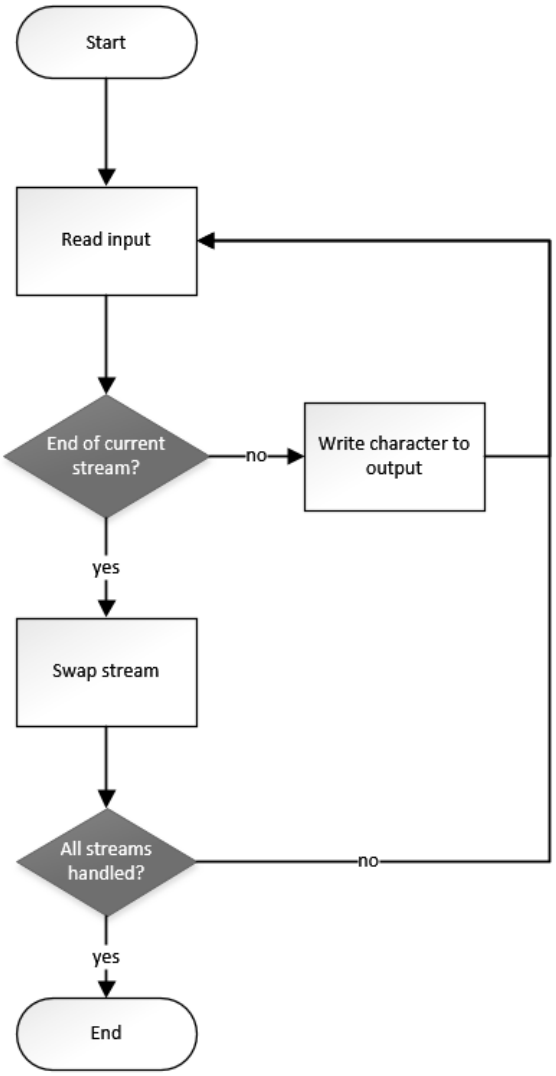

Figure 38. – Merge IP block behavioural

Since we expect the usage of FIFOs with sufficient depth connecting the Reduce and Merge IP blocks, we do not need to worry about throttling the data transfer when handling one complete line before swapping to the next; data is streamed from the Merge IP block to the DMA without interruption.

5.6. CO-SIMULATION AND IMPLEMENTATION RESULTS

Considering co-simulation results being reliable, in regards of the latency expressed as cycles, we summarise in Table 1 the co-simulation latency and the maximum working frequency obtained in the implementation process to make throughput estimations. Although the frequency estimated through logic synthesis may vary in the final implementation, the co-simulation latency is not going to vary.

We stress that, since the Zynq's PL works at a maximum frequency of $250\ \mathrm{MHz}$, the throughput calculations have been made with such a frequency, which corresponds to a $4\ \mathrm{ns}$ CLK cycle. It follows then, that all our custom IP blocks have a requirement of working at a maximum of $\sim 4\ \mathrm{ns}$ frequency to properly function.

Throughput has been calculated *per* equation (1), with D being the data processed or transferred and T the time it took to do such a task.

$$\mathrm{Throughput[bits\ per\ second]} = \frac{\mathrm{D[bits]}}{\mathrm{T[s]}} \qquad (1)$$

The parameter report has been synthesized in the following Table 1.

Table 1. – Implementation report – Frequency, latency and throughput

IP	Max. lat. (Cycles)	Max. Freq. (ns)	Max. Freq. (MHz)	Depth (bytes)	Throughput (Mbps)
Split	16,620	2.931	341.180	7,941	955.596
Map	947	2.140	467.290	911	1,923.970
Reduce	7,982	3.986	250.878	882	220.997
Merge	15,507	2.542	393.391	7,726	996.453

As we can see, the highest throughput is obtained for the Map function. The Split and Merge lower throughput as they are splitting and combining multiple MR lines, and the Reduce function has a low throughput response, as it contains high computation and comparatively complex algorithms (compared to its neighbouring IP blocks).

With these results, we conclude that one MR line (composed of 1 Map and 1 Reduce IP block) has a maximum throughput of $220.997\ \mathrm{Mbps}$, as defined by its lowest element (the Reduce block). If we consider that the final MR worker solution employs 8 MR lines, we can deduce a maximum throughput of $1.768\ \mathrm{Gbps}$, although the Merge stage is going to diminish that throughput down to $\sim 996\ \mathrm{Mbps}$.

The utilization of each block is summarised in the following Table 2.

Table 2. – Implementation report – Programmable logic utilization

IP Block	SLICE (350.0K)	SLICE (%)	DSP (900)	DSP (%)	LUT (218.6K)	LUT (%)	FF (437.2K)	FF (%)	BRAM (1,090)	BRAM (%)
Split	511	0.146	0	0.000	1,406	0.643	1,300	0.297	0	0.000
Map	47	0.016	0	0.000	119	0.054	144	0.032	0	0.000
Reduce	1,644	0.469	0	0.000	4,837	2.212	2,363	0.540	26	2.385
Merge	135	0.038	0	0.000	349	0.160	350	0.080	0	0.000

It is important to note that the Vivado HLS reports BRAM utilization as 18 Kb blocks, whereas Vivado IDE reports BRAM utilization as 36 Kb blocks.

In terms of logic utilization, it is evident that the Reduce block has the highest utilization rate over all the resources, employing 26 18 Kb BRAM's (468 Kb) per Reduce block, which amounts to 3.744 Mb for 8 MR lines. The limiting resource factor is therefore the BRAM consumption of the Reduce block, with a 2.385 % utilization, which allows only 41 Reduce IP blocks (~97.785 %) on the ZC706 device, without considering additional IP blocks and standard Xilinx IP's.

5.7. PLATFORM MODELLING

Once the custom logic has been defined, we construct the hardware platform that is implemented on the FPGA of our given SoC. The platform modelling is done via the Vivado IP Integrator, which allows the integration of custom IP solutions as well as standard Xilinx IP.

Our solution employs a DMA to directly communicate in-memory allocated data with our custom platform. This does not mean, however, that a future implementation may interconnect the hardware accelerator core with a peripheral solution, such as a GigE interface or PCIe. The described architecture can be summarized in the following block diagram (Figure 39). We omitted the AXI Interconnect block, that connects the PS derived AXI Master with the custom logic and the DMA block.

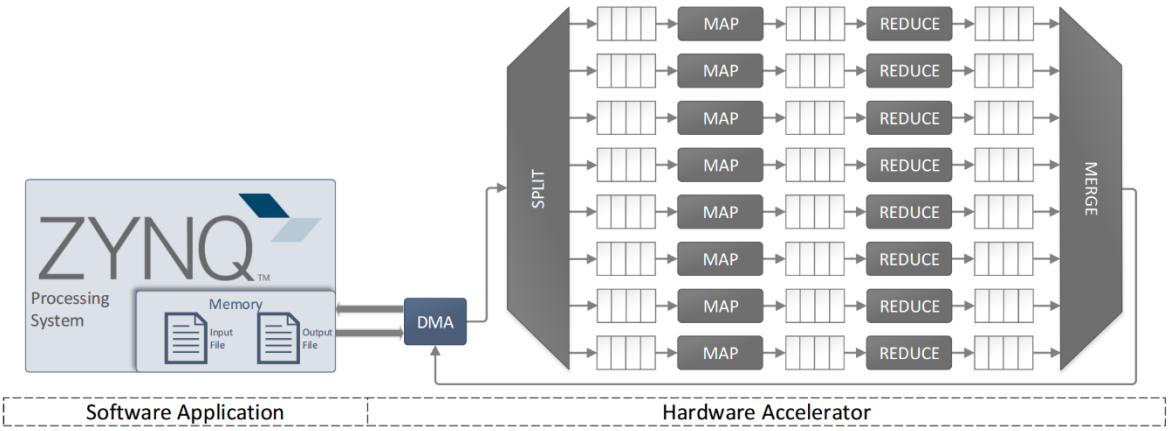

Figure 39. - MapReduce based Word Count worker architecture

By instantiating the necessary IP blocks and elements required for our design, the resulting platform is presented in Figure 40.

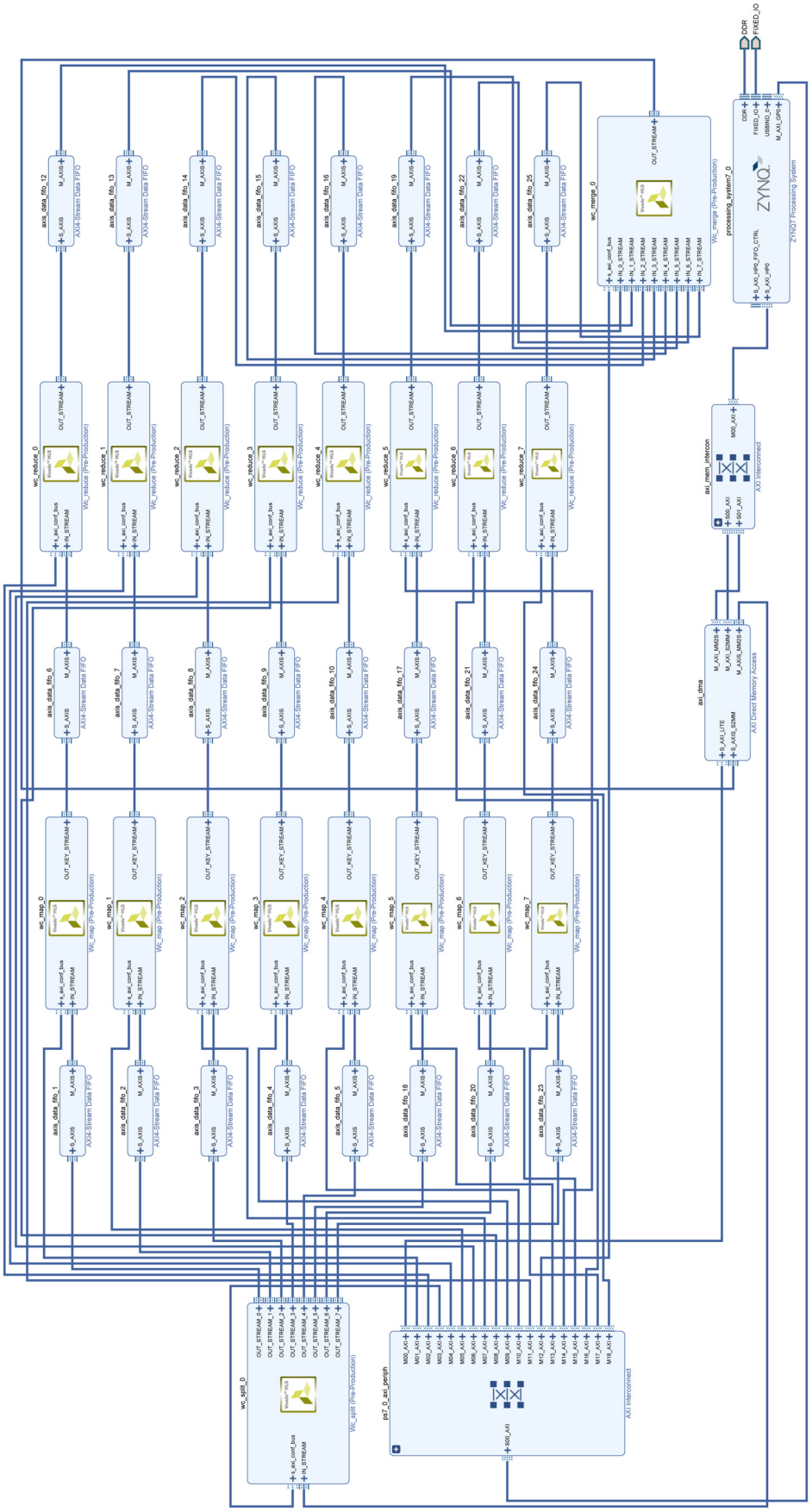

Figure 40. – Zynq ZC706 Platform – Complete MapReduce worker

The shown platform has been simplified, showing interface connections only and omitting clock and reset signals, as well as the Processor System Reset IP block. Custom IP blocks stand out by using a Vivado HLS logo, whereas standard Xilinx IP do not.

The standard Xilinx IP employed in the design are:

- **AXI Direct Memory Access (DMA)**: interfacing custom logic IP with an AXI Interconnect that accesses the PS memory.
- **AXI Interconnect**: interfacing the PS memory with the DMA.
- **AXI Interconnect**: interfacing the PS controlled AXI Master interface with the custom logic AXI Lite Slave interfaces for IP block configuration and control (start, stop, reset).

  It is a requirement to design the IP block including an AXI Lite interface so that future driver functions may be inferred by the Vivado IDE. Otherwise, no interfacing between the PS located software and the custom IP block can occur, which makes us unable to drive the IP blocks signals to start, stop, restart or configure it.
- **Processor System Reset**: required standard Xilinx IP that governs the global reset signal**.**
- **ZYNQ7 Processing System**: required standard Xilinx IP. Allows for PS configurations, such as establishing the PL fabric clock frequency ($250\ \mathrm{MHz}$ max), the processor/memory clock frequency $800\ \mathrm{MHz}$ max), enabling/disabling fabric interrupts, enabling/disabling AXI or peripheral interfaces, etc.
- **AXI4-Stream Data FIFO**: employed between split, map, reduce and merge stages to allow for a dataflow friendly architecture. We used 24 FIFOs: 16 with a depth of 1 KB and 8 with a 4 KB depth.

In light of achieving better implementation results, we opted for a Performance retiming strategy, accessible via the Vivado Implementation Options [32].

To comply with our stream depth and width, we used 1 KB FIFOs connecting the Split and Map blocks, as well as in between the Map and the Reduce stages. Since the Reduce block sends at most 2 times the data input (as described in Chapter 4, section 4.2.3)

and the FIFO depths come in multiples of 16, we employ 2 KB FIFOs at its output, connecting them to the Merge IP. We did not require additional FIFOs between the DMA and our custom IP's. We illustrate the FIFO configuration in Figure 41.

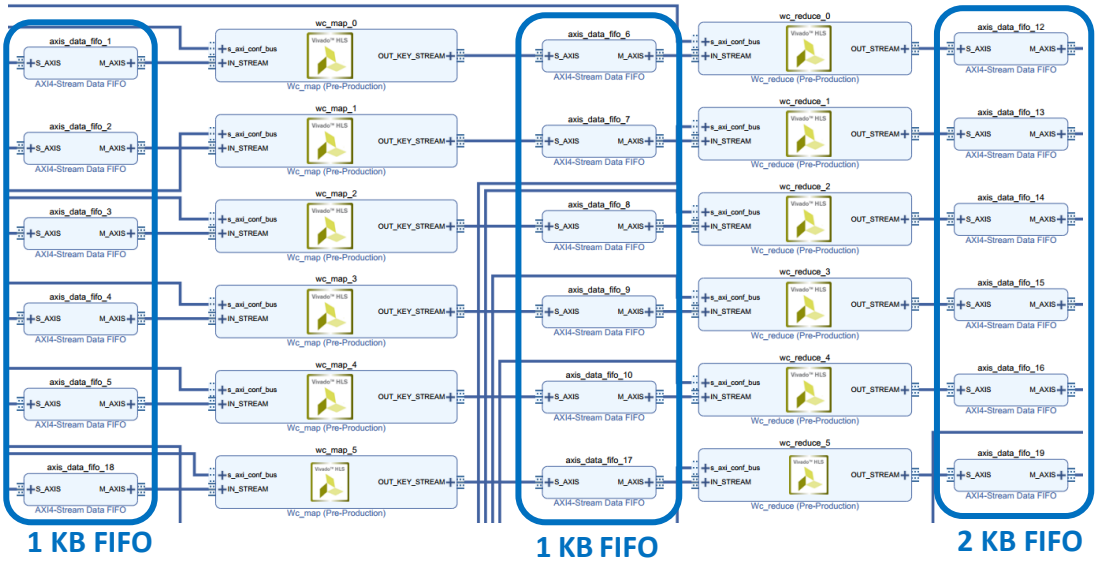

Figure 41. - FIFO position and configuration

We decided to use our DMA in interrupt mode for both MM2S and S2MM (memory-to-stream and stream-to-memory) channels, as we reduce data transfer latency by interrupting whatever the PS was doing and serving the interrupting agent. In order to allow both, MM2S and S2MM interrupts to be served, we employ a Concat IP block that allows a logic OR between both interrupt signals and connects the result to the PL-PS fabric interrupt of the Processing System (Figure 42).

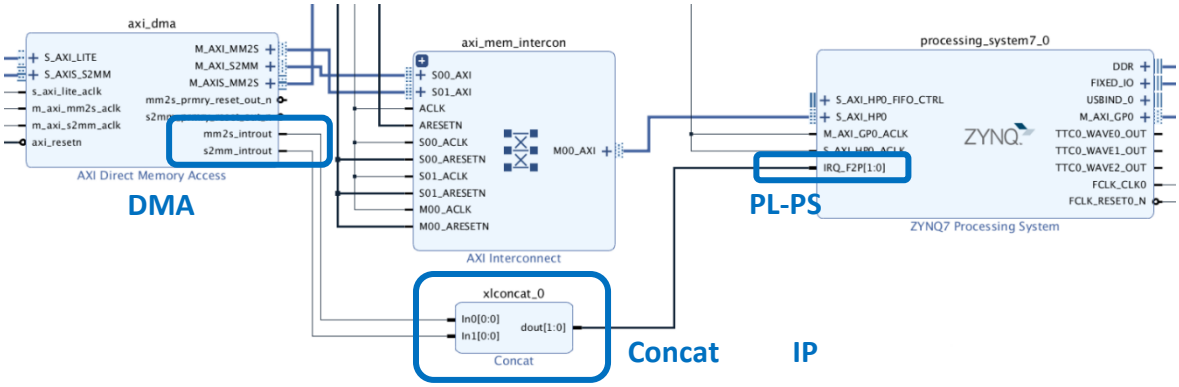

Figure 42. - DMA PL-PS interrupt setup

The reset of the DMA configurations include an 8-bit data width on both the read channel and the write channel, with a memory map data width of 32 bits.

5.8. Platform Implementation results

Once the platform is modelled accordingly, the design is put through a logic synthesis and implementation phase to produce the required output products that are ultimately implemented on the SoC's FPGA as a netlist. The results of this implementation phase are presented in this section.

5.8.1. Layout

To identify all the custom PL elements that compose our platform, Figure 43 presents the hardware layout, highlighting the most relevant cells of the design.

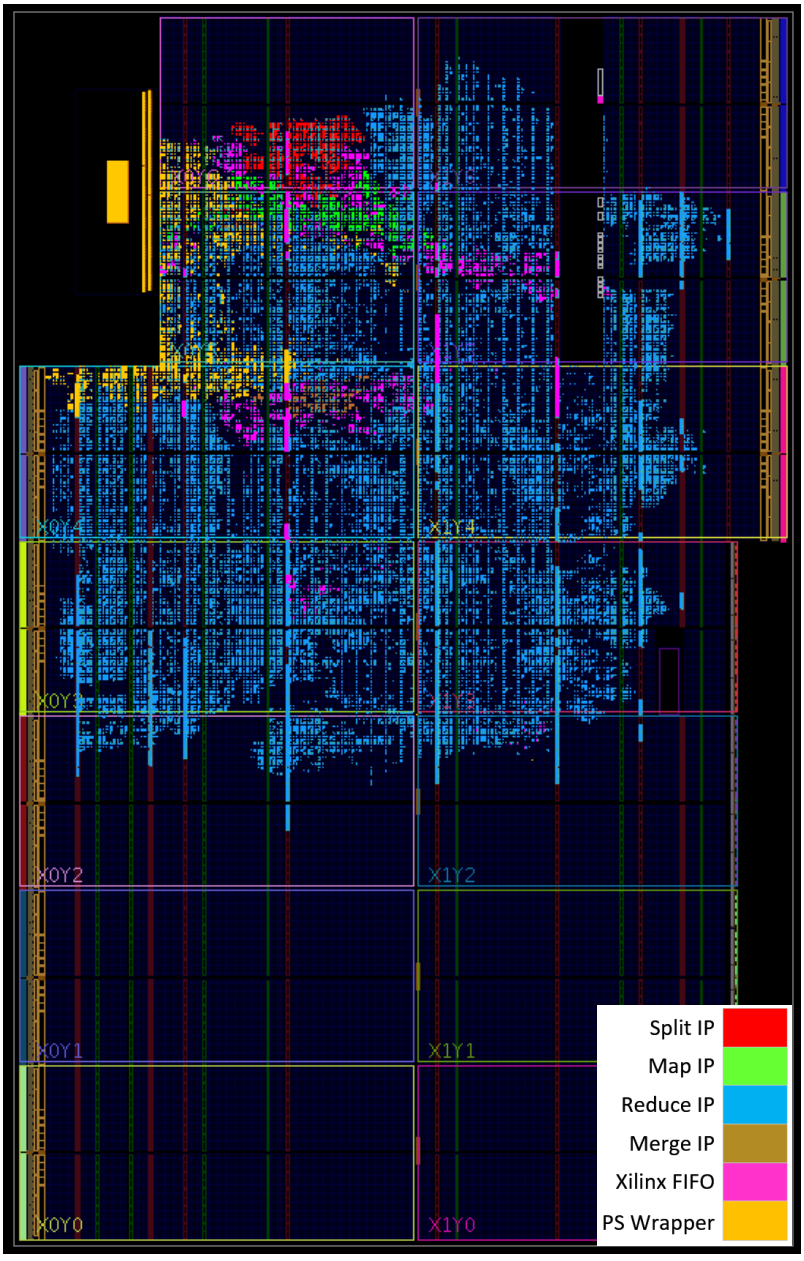

Figure 43. - MapReduce worker platform layout

We observe that the programmable logic starts off by gathering around the system's PS (upper left, orange) and then constructs itself around it to increase the maximum operating frequency by reducing datapath lengths and clock signal routing. As expected, the Reduce IP block has the greatest programmable logic footprint, since it consumes the most BRAM's.

5.8.2. Utilization

The complete utilization report is shown in Table 3 and Table 4. We illustrated as 'Platform' the report associated to the complete hardware accelerator, including custom and standard IP. We want to remind that Vivado reports the BRAM usage by considering 36 Kb blocks, instead of 18 Kb such as Vivado HLS does.

Table 3. – Absolute hardware platform utilization report [1]

IP	Slices (54,650)	LUT (218,600)	LUTRAM (70,400)	FF (437.2K)	BRAM (454)
Platform	17,049	45,564	21,564	30,113	145.5
Split	457	1,296	0	1,298	0.0
Map	427	1,039	0	1,152	0.0
Reduce	14,143	37,901	21,376	20,768	104.0
Merge	145	321	0	350	0.0

Table 4. – Relative hardware platform utilization report

IP	Slices (%)	LUT (%)	LUTRAM (%)	FF (%)	BRAM (%)
Platform	31.20	20.84	30.63	6.88	32.05
Split	0.84	0.59	0.00	0.29	0.00
Map	0.78	0.47	0.00	0.26	0.00
Reduce	25.88	17.34	30.36	4.75	19.08
Merge	0.26	0.15	0.00	0.08	0.00

We conclude that the limiting factor for the usage of more MR-WordCount workers is the BRAM usage at 32.05 %, and the LUTRAM usage coming second, as one worker consumes up to 21,564 LUTRAM's (30.63 %), whose greatest contributor are the Reduce IP's. The utilization report is illustrated in Figure 44.

[1] Standard IP derived programmable logic has been omitted, such as FIFOs or AXI Interconnects

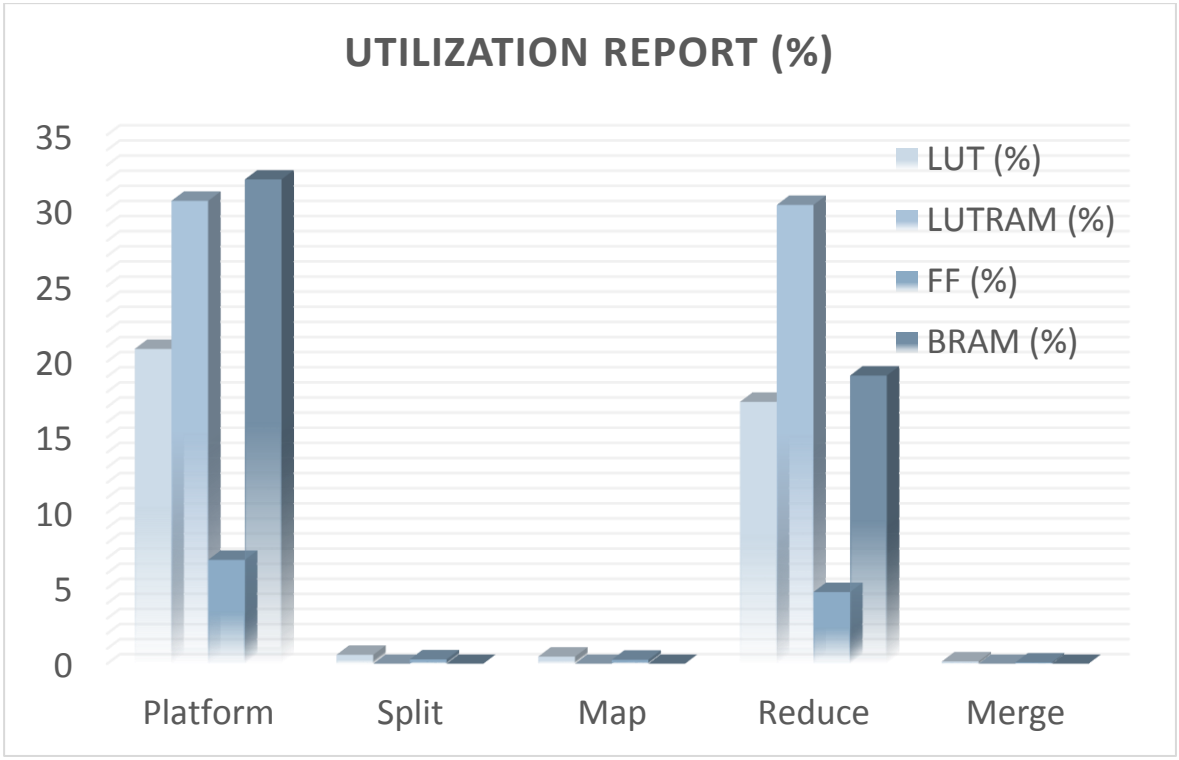

Figure 44. - Hardware platform utilization report (%)

It easily follows, that the Reduce IP block is the most consuming in the entire platform, which occurs because it integrates buffers of large dimensions, something that could be resolved or optimized in future iterations.

We conclude that we can fit a maximum of 3 MR workers on the custom platform, consuming then a maximum of $96.15\ \%$ of the maximum available LUTRAM's.

5.8.3. Utilization factor

As explained in [26], 4 LUT's and 8 FF's are contained in one single Slice of a Zynq PL. As a matter of measuring PL utilization efficiency, we define a Utilization Factor (UF) as presented in equation (2) and (3).

$$\mathrm{UF}_{\mathrm{LUT}} = \frac{\text{N. of LUT's}}{\text{N. of Slices}} \qquad (2) \qquad \mathrm{UF}_{\mathrm{FF}} = \frac{\text{N. of FF's}}{\text{N. of Slices}} \qquad (3)$$

Considering the slices, LUT and FF utilization of our hardware accelerator, we can synthesize its Utilization Factors in

Table 5.

Table 5. – Hardware accelerator utilization Factor

IP	UF_{LUT}	UF_{FF}
Platform	2.67	1.77
Split	2.84	2.84
Map	2.44	2.70
Reduce	2.68	1.47
Merge	2.21	2.41

The mean $\mathrm{UF}_{\mathrm{LUT}}$ is 2.67, while its $\mathrm{UF}_{\mathrm{FF}}$ is 1.77. As previously stated, 1 slice contains up to 4 LUT's and 8 FF's. Regarding our results, we can conclude that our design has a good $\mathrm{UF}_{\mathrm{LUT}}$ factor is it approaches 4, whereas the $\mathrm{UF}_{\mathrm{FF}}$ is poor. This occurs as our design relies more on BRAM and LUT programmable logic and not on FF's, since the Reduce IP is the most PL consuming block.

If our design were less memory oriented and dataflow centred, our $\mathrm{UF}_{\mathrm{FF}}$ would rise to be closer to a factor of 8. The same would happen if we'd use retiming directives during the Vivado HLS design phase as these techniques requires a high FF usage. However, the used area would increase since BRAM's are more compact in terms of storage than FF's.

By comparing this work with other in the field, such as presented by Carballo in [34], we can compare our results with other implementations that share similar architectures on other Zynq devices, as presented in Table 6.

Table 6. – Slices utilization factor comparison with [34]

Design	Design architecture	HLS Tool	FPGA Device	Utilization factor	
				$\mathrm{UF}_{\mathrm{LUT}}$	$\mathrm{UF}_{\mathrm{FF}}$
0	Platform + IP	Vivado HLS	Xilinx Zynq 7z020	2.91	2.90
1	IP	CtoS	Xilinx Zynq 7z045	3.32	1.64
2	Platform + IP	Vivado HLS	Xilinx Zynq 7z045	2.29	3.05
3	IP	Vivado HLS	Xilinx Zynq 7z045	2.12	2.79
4	Platform + IP	CtoS	Xilinx Virtex-5 FX 130t	2.85	2.09
MR Worker	Platform + IP	Vivado HLS	Xilinx Zynq 7z045	2.67	1.77

Our LUT utilization factor is on par with the overall $\mathrm{UF}_{\mathrm{LUT}}$ for similar solutions on Zynq devices, as well as a Xilinx Virtex-5. None of the designs achieve a high $\mathrm{UF}_{\mathrm{FF}}$ usage, something we can attribute to the HLS design flow, which underuses the available slices, despite having up to 8 FF's per slice on a Zynq device.

5.8.4. Timing summary

The designed hardware platform, or accelerating core, is implemented with a maximum operating frequency of $800\ \mathrm{MHz}$ on the PS side, and $250\ \mathrm{MHz}$ on the PL side. The achieved Worst Negative Slack (WNS) on the PL side is $97\ \mathrm{ps}$, allowing an operating frequency of $250\ \mathrm{MHz}\ (4\ \mathrm{ns})$.

5.8.5. Power consumption

Lastly, the power consumption is summarized in Figure 45, as provided by the Vivado implementation report tool.

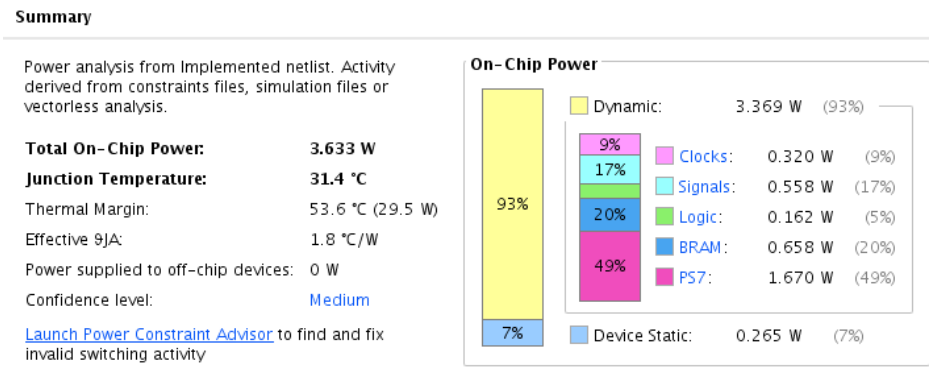

Figure 45. - Hardware platform power consumption summary

The power consumption is divided into dynamic and static power. Whereas the first belongs to the power consumption derived from the implemented logic on the FPGA, the static power belongs to the amount of leakage current of the employed technology's transistor and materials.

The MapReduce platform has a dynamic power consumption of $3.369\ \mathrm{W}$ while its static power consumption is of $0.265\ \mathrm{W}$. As we can see in the previous figure, the most consuming element of the dynamic power is the PS IP block, that communicates over all the platform using AXI-Lite interfaces to all the custom IP's, as well as managing the communications between the DMA and memory through AXI Stream.

5.9. Conclusion

This chapter concerns itself with the hardware platform design, that is ultimately implemented on our ZC706 device. The design process starts by stating the Vivado HLS considerations, or design dos and don'ts, as HLS does not support some C/C++ constructs. To solve these problems, we present a synthesizable functions library, and to allow for a flexible design flow we present the usage of a tuneable parameters file and an *ad-hoc* created AMBA AXI4 Stream driver, to properly drive the interface's signals.

Following up, we describe the modelling processes for our individual hardware IP's, design issues and obtained results by presenting their co-simulation and implementation products. The highest obtained throughput belongs to the Map IP block with $1{,}923.970$ Mbps, while the lowest belongs to the Reduce IP, with 220.997 Mbps, through which we can infer a maximum Reduce stage throughput of $1{,}767.976$ Mbps.

We cover the hardware platform design by using our custom hardware IP's and the standard Xilinx IP's. We implement the MapReduce worker, as proposed in Chapter 4, resulting in maximum a programmable logic utilization of 32.05 %, as a consequence of the Reduce IP block's high BRAM utilization. The maximum working frequency is the highest we can set for the PL, which is 250 MHz.

Chapter 6. HARDWARE-SOFTWARE INTEGRATION

Since Chapter 5 concerned itself with the hardware platform design of our accelerator core, in this chapter, we present the integration of the hardware platform and the software application that runs it. This includes the hardware (HW) platform instantiation, the Board Support Package (BSP) creation, the software creation itself and the presentation of its behaviour.

The Hardware-Software integration is accomplished using the Xilinx SDK environment, which allows for easy FPGA and SoC memory programming as well as debugging through JTAG.

6.1. HARDWARE PLATFORM INSTANTIATION AND BSP CREATION

In the interest of integrating the hardware with a software application via the Xilinx SDK environment, we start instantiating the generated Hardware Platform (HWP). The HWP specification captures all the information and files from our generated hardware accelerator core in Vivado IDE, that is required for the designer to be able to write, debug and deploy software applications for that hardware [31].

The HWP contains first and foremost, the hardware design wrapper generated in Vivado, the PS configuration for the targeted device, and the drivers associated to our custom PL solutions, as presented in Figure 46.

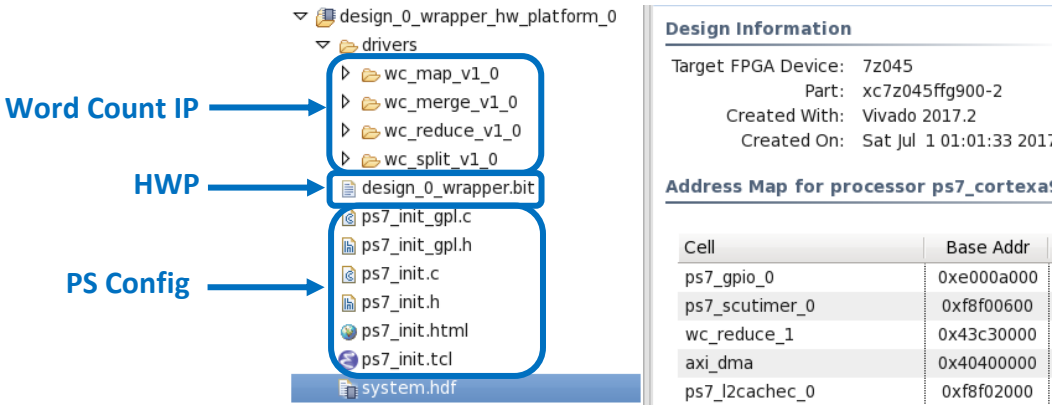

Figure 46. - Word Count MR Worker Hardware Platform files

Having the HWP at hand, we must develop two more elements to complete the Hardware-Software integration:

- The Board Support Package (BSP).
- The bare-metal software application.

A BSP is a collection of libraries and drivers that provides the designer with the means of accessing the lowest layer of our Hardware-Software solution [31]. Since our HWP carries the drivers for our custom IP designs, during the BSP creation process, Xilinx SDK generates the remaining drivers and libraries needed for a designer to access the lowest layer of the custom design, and merges them with the custom IP drivers. If a software design is integrated without a BSP, it is not possible to link the software layer with the generated hardware. To evidence this, we present the HWP, BSP and application integration flow through Figure 47.

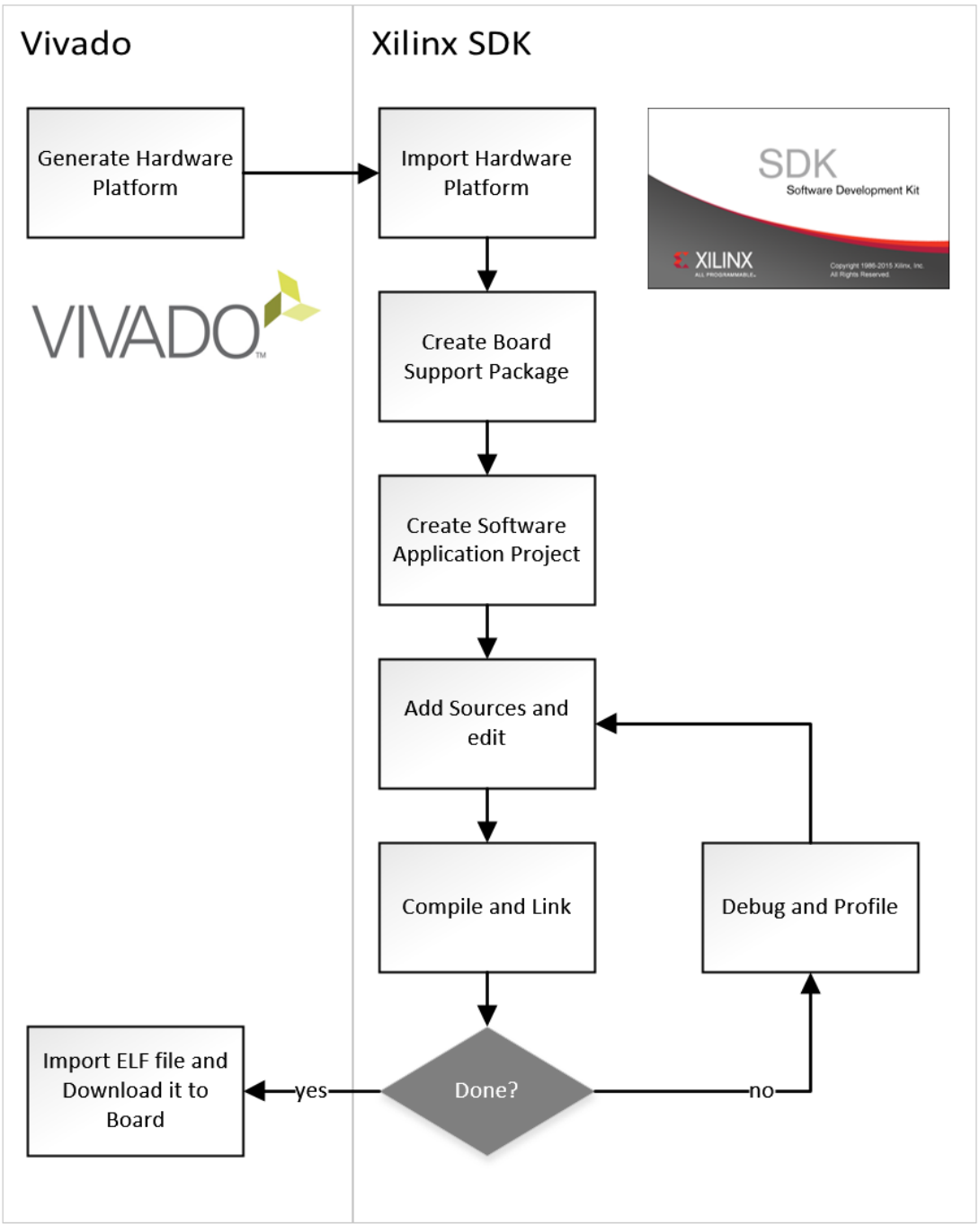

Figure 47. - Xilinx SDK integration flow and Hardware-Software integration

Once the BSP is generated, it includes our custom drivers, standard C/C++ libraries (mostly adapted by Xilinx) and custom Xilinx libraries, that allow for easier communication between designer and design. As an example, in Figure 48 we illustrate the mentioned drivers and a few of additional peripherals.

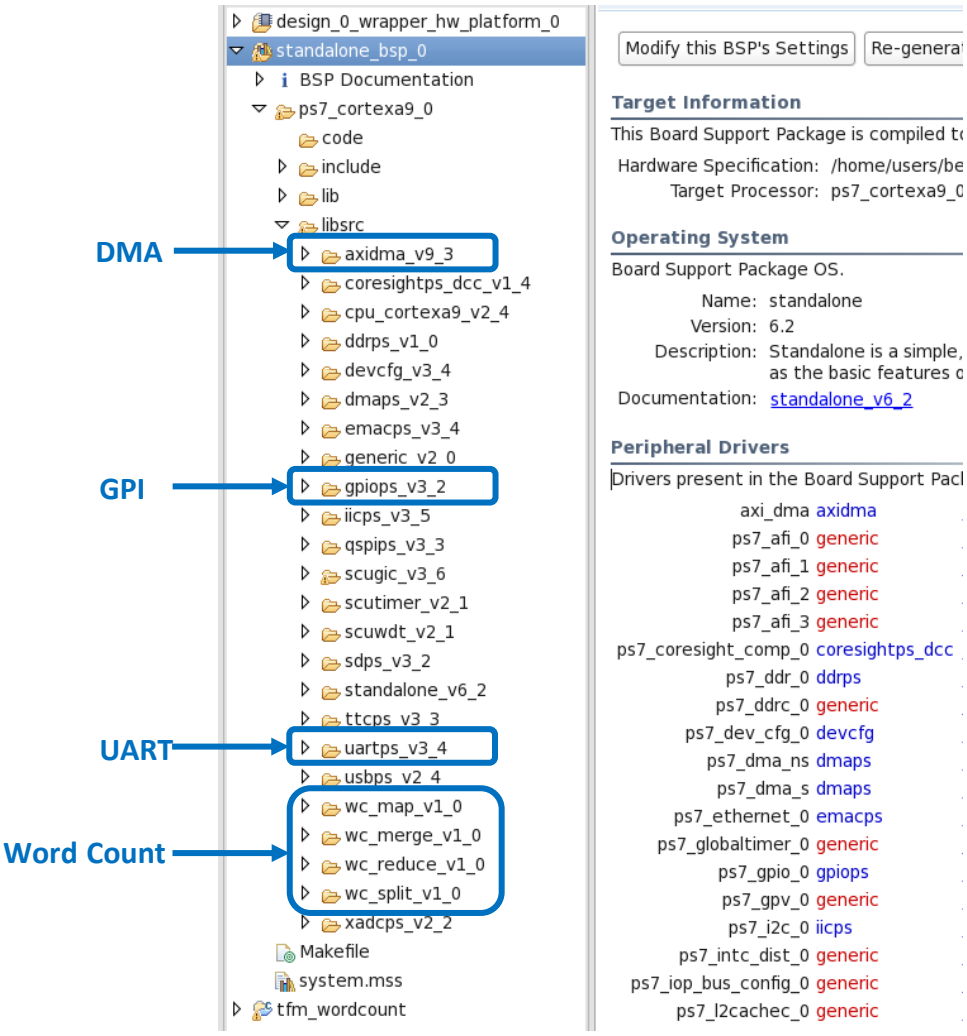

Figure 48. - ZC706 BSP and its drivers

Once the Hardware Platform and the BSP are created, it is possible to design a Bare-Metal Application that uses the BSP to ultimately access the HWP on our Zynq AP SoC.

6.2. Bare-metal application

Equivalent to platform modelling in hardware, in this section we concern ourselves with the application design in software. The software application manages the hardware platform and memory initialization and configuration, as well as signal handling such as interrupts and resets of the system.

The BSP OS chosen for our design was Standalone 6.2, which is a simple, low-level software layer that provides access to the basic features of the processor such as caches, interrupts and exceptions, as explained by the Xilinx SDK environment [35].

6.2.1. Used libraries and drivers

We have used different libraries and drivers throughout the application development. Before we dive into the behaviour of our bare-metal-application, in Table 7 we summarize the main utilized drivers and libraries, including those that allow us to manage our custom PL IP blocks. Standard C/C++ function libraries have been omitted. We marked the libraries and drivers according to its author: Xilinx, if it is designed by Xilinx in its entirety; HLS, if it is a driver issued by the Xilinx Vivado HLS environment; User, if it is a user defined library or driver.

Table 7. - Used libraries and drivers

Lib./Driver	Creator	Functionality
xtime_l.h	Xilinx	Access to the 64-bit Global Counter in the PMU. Allows to get/set time in the global timer. Used for timestamps.
xscugic.h	Xilinx	Generic interrupt controller driver component.
xaxidma.h	Xilinx	Driver API for the AXI DMA engine.
xil_printf.h	Xilinx	Function library to output data the same as the standard printf() function, without the overhead most run-time libraries involve.
xparameters.h	Xilinx	Contains definitions related to the current hardware platform, such as the device ID of the AXI DMA or the PS IP block. Definitions for the peripheral memory addresses, as mapped
xparameters_ps.h	Xilinx	Contains address definitions for the hard peripherals attached to the PS core.
xil_exception.h	Xilinx	Contains ARM Cortex A9 specific exception related API's.
xwc_map.h	HLS	Contains the driver API that manages the custom IP for initialization, configuration and execution. Created automatically by Vivado HLS.
xwc_reduce.h	HLS	
xwc_split.h	HLS	
xwc_merge.h	HLS	
platform.h	User	Contains the MapReduce worker API for initialization, configuration and execution. Basically, employs the xwc_* driver API's to set up the entire MapReduce worker.
utils.h	User	Contains user specific functions for buffer and time stamps handling.
wc_parameters.h	User	Contains user specific parameters to fine-tune the design with ease.
wc_axidma.h	User	Contains the AXI DMA API for initialization, configuration and execution. Basically, extends the xaxidma driver API to set up the DMA, its interrupt system and its IRQ services.

We remind the reader that the programmed Zynq AP SoC put under test and validation with our custom hardware platform, has no OS installed in its memory, as it is a Standalone OS. Although we are still using a DMA, we can copy the input files onto a Secure Digital (SD) card and read the information back to send it to our accelerator platform. In the same fashion, we can recover the data from the core and store the results in an output

file.. Thus, the input stream data is allocated previously in memory and then sent to the hardware accelerator, while the output stream is recovered and presented to the designer to validate the behaviour of the hardware accelerator core.

6.2.2. System Setup

To send data to our MapReduce worker, we must first initialize and configure our hardware platform. For short, we call this System Setup, and it can be summarized in Figure 49.

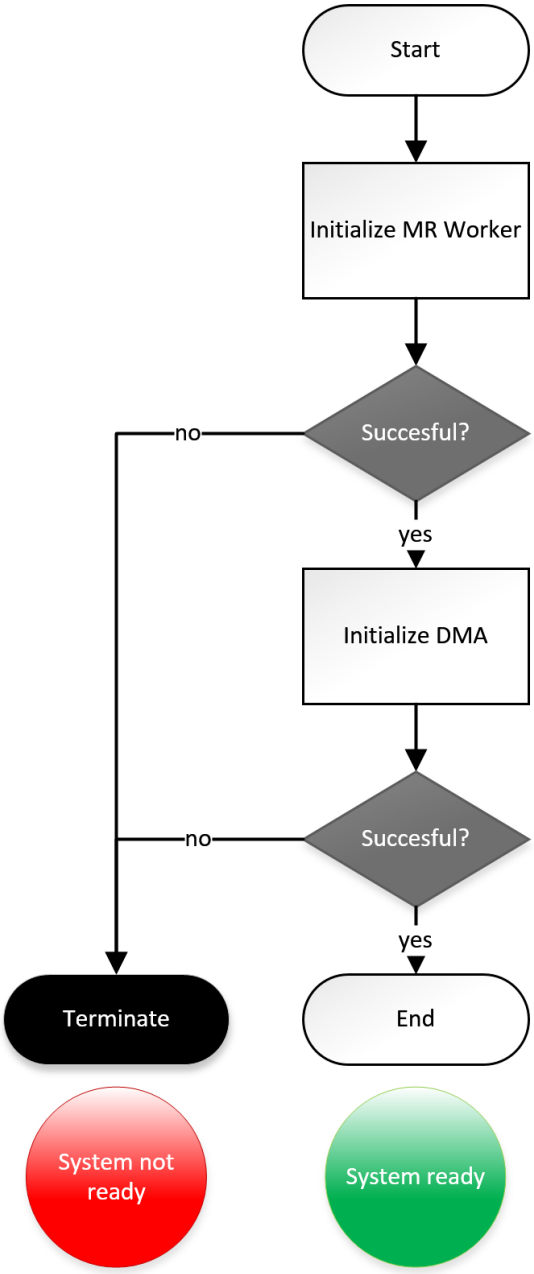

Figure 49. - System setup flow

The System Setup processes must include:

- DMA initialization.
  - Initialize the DMA driver.
  - Enable interrupts on both TX and RX channels of the DMA
  - Set up the interrupt system, which includes setting the interrupt triggers type and priority and linking the interrupt controller driver with the device driver handler that responds when an interrupt for the device occurs.
  - Enable the IRQ routine that services the DMA interrupts and resumes the main application.
- MapReduce worker initialization.
  - Initialize the Split IP driver.
  - Initialize the 8 Map IP's drivers.
  - Initialize the 8 Reduce IP's drivers.
  - Initialize the Merge IP driver.

An initialization of any of the underlying IP blocks uses the HLS generated drivers and may look like this:

```
int map_init(XWc_map * device, int device_id) {
status = XWc_map_Initialize(device, device_id);

if (status != 0) {
       xil_printf("\tMap initialization failed %d. Incidental with device
              number \r\n", status, device_id);
       return 1;
}

XWc_map_Start(device);
XWc_map_EnableAutoRestart(device);
XWc_map_InterruptGlobalDisable(device);

xil_printf("[Map] Map %d initialized.\n\r", device_id);
return 0;
}
```

First, we link the software construct pointer to the devices hardware ID, enabling therefore access to the hardware device by calling its software construct. Following up, we start up the device, enable the auto restart functionality of our IP block (the device restarts its I/O signals if a transaction

is completed) and we disable its interruptions, as they are not used for any of the blocks that reside inside our MR worker.

6.2.3. Application flow

Once the system is set up successfully, the hardware platform is ready to receive data and produce a formatted output at its end. A normal application function flow then is summarized in Figure 50.

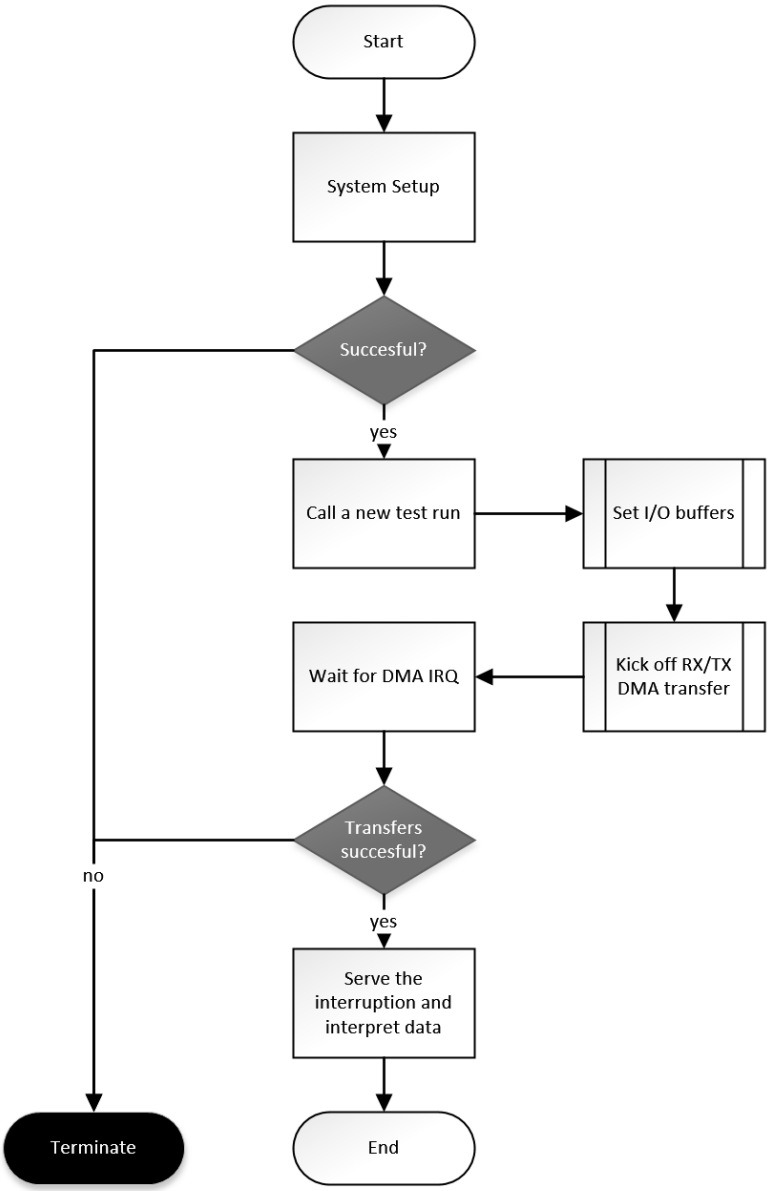

Figure 50. - Main application behavioural flow

Such a behavioural is presented in the following snippet of code.

```
int main(int argc, char *argv[]) {
```

```
	xil_printf("\r\n--- Entering main() --- \r\n");

	/** Initialize the XTime variables with trivial values **/
	XTime_GetTime(&tStart);
	XTime_GetTime(&tEnd);

	/** Set up the platform **/
	platform_setup();

	/* Make a test run to start the IP blocks.
	 * The IP blocks that store information in memory (WC_Reduce), clear themselves
	 * when emitting a response, which allows new transfers. */

	/* Execute runs */
	for (int i = 0; i < 5; i++) {
		for (int j = 0; j < 80; j++) {
			partition = 100 * j;
			test_run(key_ptr[i], partition);
			if (Status != XST_SUCCESS) {
				xil_printf("\tERROR! Run %u failed! Status value: %d\n\r", i,
Status);
				return XST_FAILURE;
			}
		}
	}

	xil_printf("\r\n--- Exiting main() --- \r\n");
	return Status;
}
```

After calling the main System Setup function with satisfactory results, the user can transfer data to and from the MR worker. We previously allocated ~ 8 KB test keys to explore our MapReduce application.

The `test_run()` function is given the size of the data and a pointer to it. It then kicks off the DMA transfers and waits for the DMA interruption to respond. Once the IRQ is served, the `test_run()` function presents the data and clears the necessary variables to allow new transfers to happen.

By defining the tests runs as we did in the previous snippet of code, we use a pointer to iterate through the given keys. Every one of the keys is then sent to the MapReduce platform by gradually increasing the transfer length from 100 bytes to a maximum of 8 KB in steps of 100 bytes. Doing so allows us to record all the latencies and throughputs by using one test application with different keys.

6.3. CONCLUSION

Having described the complete design flow from the Hardware Platform creation, Board Support Packet generation and the design of the Bare-Metal Application, we can summarize the applications main characteristics as follows:

- The MapReduce platform is wrapped in such a manner that the designs behavioural code is abstracted to the user, and only requires a System Setup and subsequent data transfers to happen.
- The MapReduce platform allows a maximum input data size of 8 KB of data.
- The output has a maximum size of 16 KB, as the data is received using the Key-Value stream protocol described in section 4.3, Streaming interfaces considerations.
- Clearing of the interrupt signals and the reset of the buffer contents and buffer lengths happen automatically after each transfer.
- The employed data is previously declared in memory, as we are using a Bare-Metal, DMA based, architecture.
- The DMA is set up using interrupt signals for both TX and RX channels, in an effort to reduce latency.
- The presented application recollects the test runs' data length, computation latency and its derived throughput for further analysis.

Chapter 7. VALIDATION PHASE

In this chapter, we use the MapReduce platform and revise its main behavioural description, validating whether it produces the expected output for a given input and in a respectable latency. By doing so, we explore the output parameters of the whole system, such as overall throughput. Once we have presented the platform's main parameters, we model its latency response with respect to a given input data to make latency predictions, and we make improvement proposals, before arriving at the conclusions for the validation phase.

7.1. VALIDATION ENVIRONMENT CHARACTERISTICS

In Figure 51, we observe the evaluation board's setup for the final validation phase, being directly connected to the workstation using JTAG and UART connectors. In this manner, we are able to program the evaluation board with our custom hardware platform and our software solution and debug it in runtime. The Ethernet shown Ethernet connectors are not being used.

For the final validation, we use 5 test keys that are used on the Word Count platform, plus a 1-Byte token key. Whereas the first key includes only a single character, the other entail a close to $8\ \mathrm{KB}$ extract from multiple different sources. The test keys are described in Table 8.

By using multiple different keys, we can calculate the platforms parameters employing samples whose word distribution and recurrence varies. We use a Byte1-Byte token key in an effort to find the lowest latency of the platform.

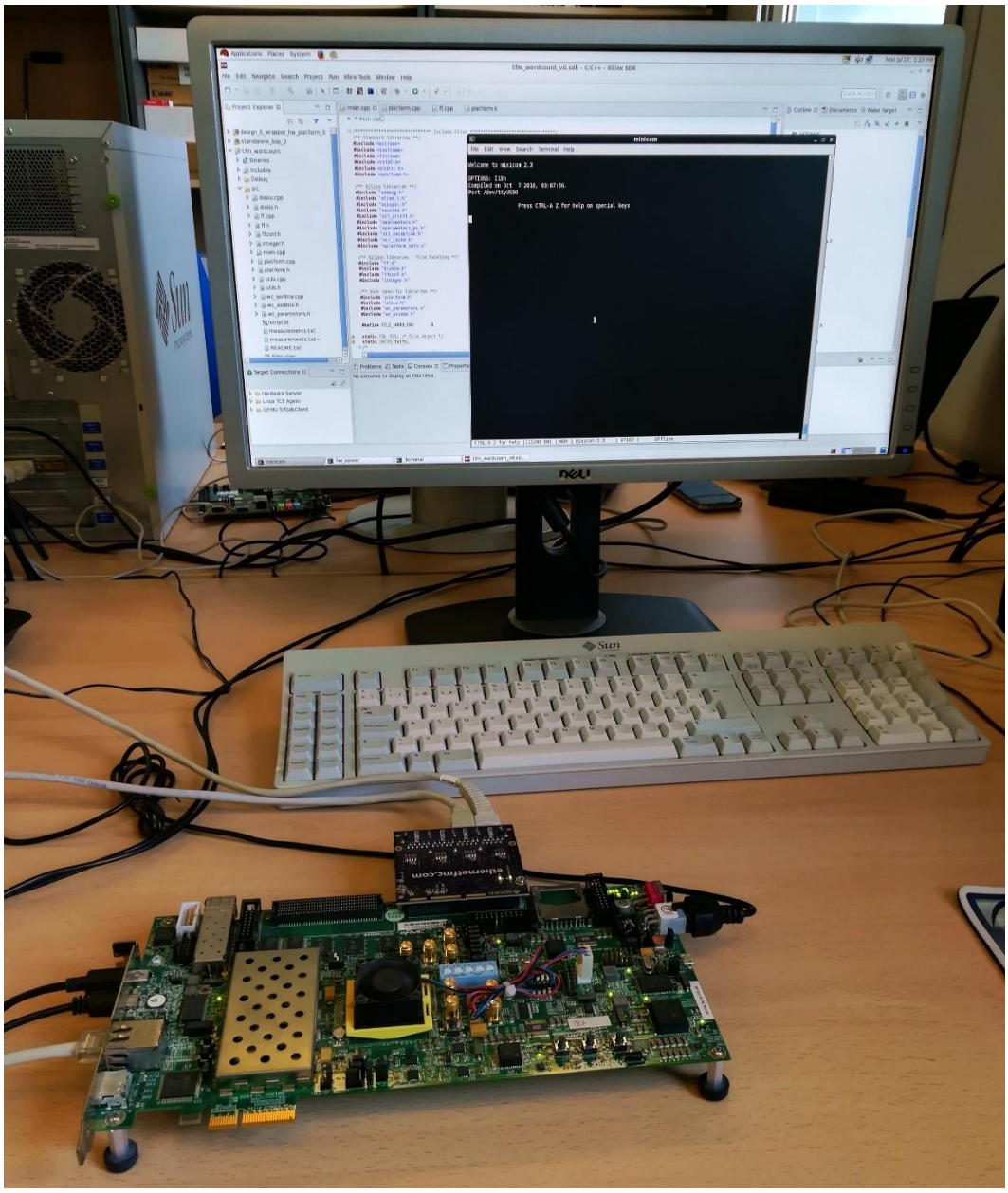

Figure 51. – Zynq ZC706 setup

Table 8. – Validation test keys

Key name	Size (bytes)	Content/Observation
token	2	Arbitrarily chosen Byte + '\0' character
space_od	7,845	An extract from the 1968 novel, "2001: A Space Odyssey" by Arthur C. Clarke
kafka	7,996	An extract from the 1915 novel, "The Metamorphosis" by Franz Kafka
li_europan	8,000	An extract from Edgar von Whal's assertion "Li Europan lingues", 1922
goethe	7,998	An extract from the 1774 epistolary, "The Sorrows of Young Werther", by Johan Wolfgang Goethe
harryp	8,000	An extract from the 1997 novel "Harry Potter and the Philosopher's Stone", by J. K. Rowling

7.2. LATENCY ANALYSIS

The latency itself is measured by using two methods:

1. the worker latency from the Split to the Merge block (or the delay between DMA transfer to device and its response), and
2. the line latency, which is found by measuring the MapReduce line's latency (the delay between the first Map block's start processing till the last Reduce block ends sending data).

These two methods are included, as a MapReduce architecture on a SoC solution may include the Splitting and Merging functions as a software implementation, as opposed to custom hardware IP's in the PL's logic.

Out of simplicity, the worker latency is also identified as a DMA per interruptions setup (DPIS), as the DMA is set up using system interrupts to service incoming data transfer, while the line latency is also called Reduce per polling (RPP), as in this case the application measures the systems delay until all the Reduce IP's have their information transmitted and uses polling to learn about it.

We use the XTime provided C/C++ constructs to place time stamps (for more information we refer the reader back to 6.2.1 Used libraries and drivers) with a precision in μs. Although we understand that latency measurements have a higher degree of accuracy by employing Integrated Logic Analyzer's (ILA), which measure latency in cycles, an overall latency measurement of this type requires a common interface through which we can observe both the transmission and reception of our data to and from the MapReduce worker. Our architecture does not have such an interface, as we transmit data over one AMBA AXI interface and receive the formatted data over another. Thus, we are required to use in-software placed timestamps to estimate latency behaviour at a higher level of abstraction.

The usage of software timestamps presents two problems:

a. It includes function call overhead and produces fluctuations in the measured time, as it depends on the PS' workload at the time of the execution. We

solved this by eliminating all unnecessary function calls, data prints and variable handling during the worker's data transfer.

b. The second problem is solved by executing a test key multiple times and taking the mean of the measured latency.

7.2.1. Latency measurements

Latency measurements are done by feeding the MapReduce worker fragments of the original test keys with an increasing size till reaching $8\ \mathrm{KB}$ of data. We repeat this process for all the given keys, with regards to obtaining enough data to model the systems latency. We arbitrarily chose sending data fragments to the MapReduce worker with a data size step of 100 bytes. The results for both DPIS and the RPP cases, are presented in Figure 52 and Figure 53.

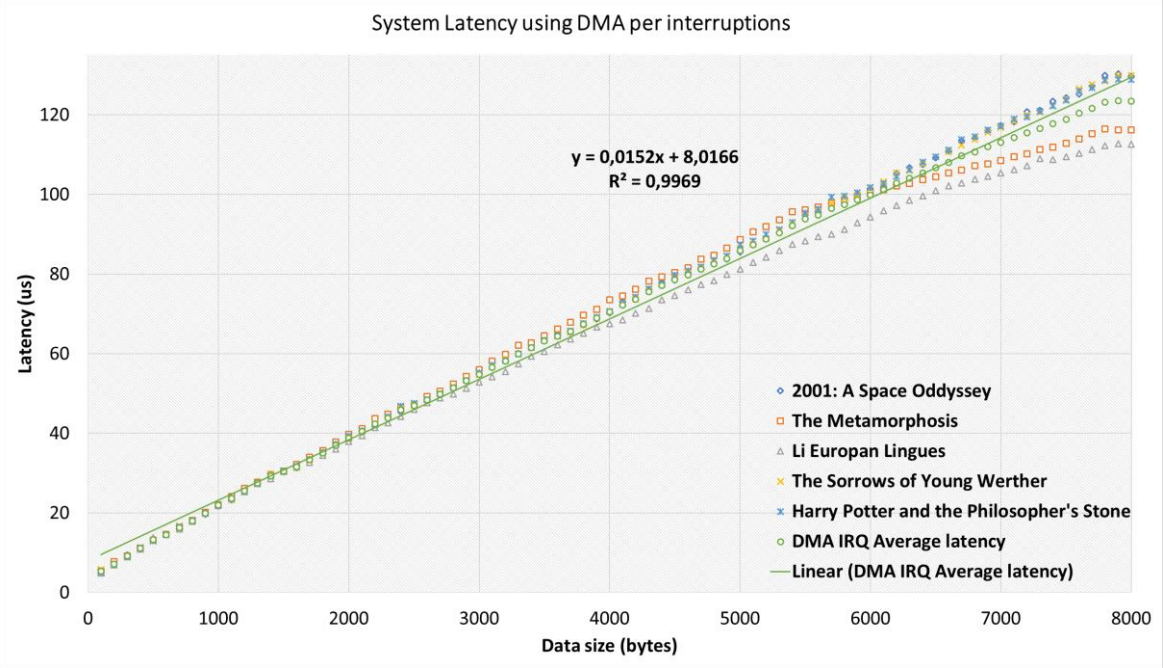

Figure 52. - System Latency using DMA per interruptions

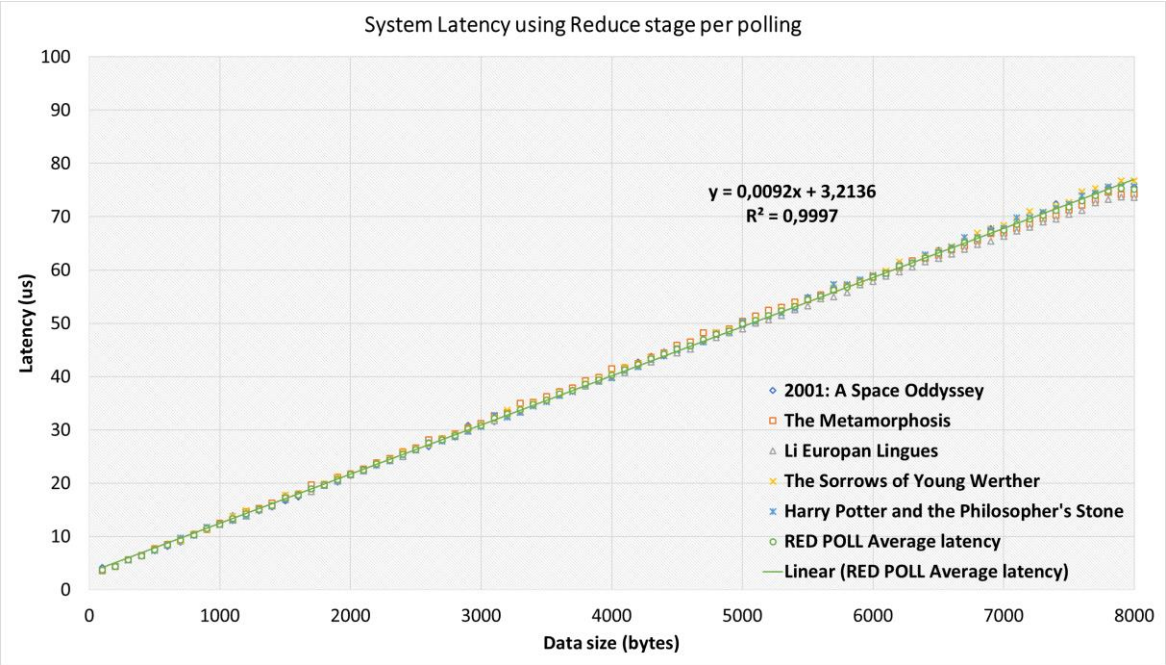

Figure 53. – System Latency using Reduce stage per polling

The mean MapReduce worker latency is higher than the line latency, as one would expect considering that it includes two IP blocks on both ends, splitting and combining the stream, adding latency to the datapath. It stands out that the line delay for the token key is higher than the delay of the worker, as the latency is measured for the first case through polling and for the second through interruptions. Also, the mean delay is higher for the entire worker. We synthesize the mean latencies in Figure 54.

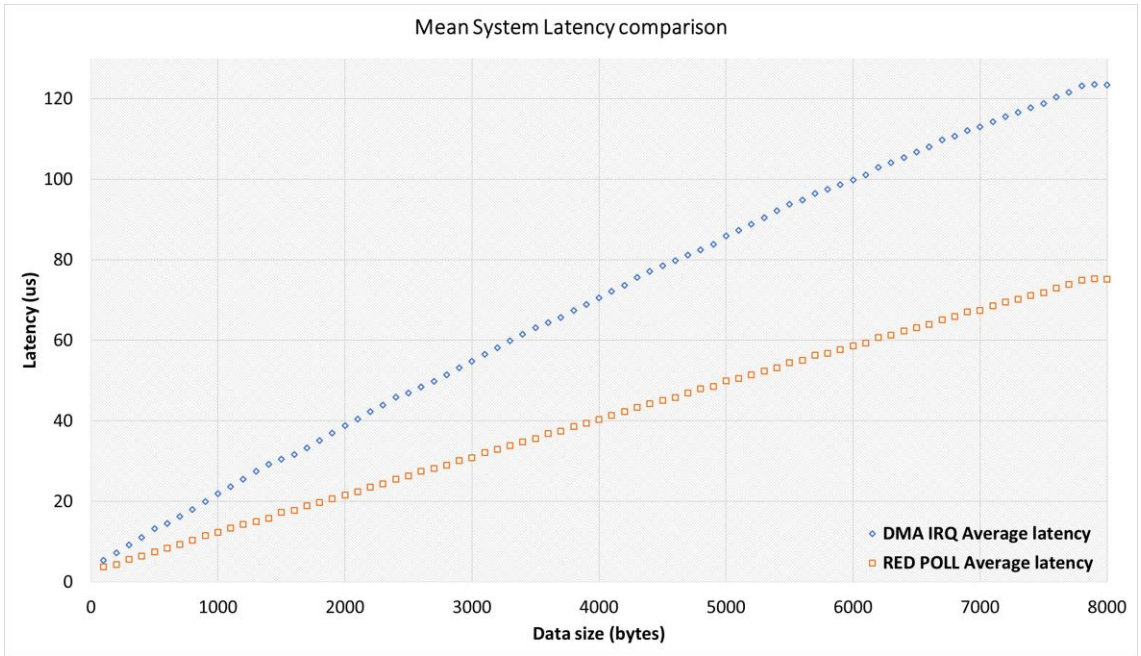

Figure 54. – Mean system latency comparison

As Figure 52 and Figure 53 shows, the platform requires $8\ \mu s$ for the DMA per interrupt case and $3.2\ \mu s$ for the Reduce stage per polling case, for $1\ \text{Byte}$ of data to traverse the entire Word Count platform, a delay we call the node delay, or d_{node}, as it is the minimum delay introduced by the system.

Whereas the line latency increases at a rate of $9.2\ \text{ns per Byte}$, the worker latency increases at a rate of $15.2\ \text{ns per Byte}$. Both latencies have a linear behaviour, with an initial offset or minimum d_{node} latency.

It is evident that the line latency is significantly lower, as the data is not passed through a bottleneck in order to gain access to the PL device and back to the memory through the DMA. By comparing these latencies, we conclude that the Split and Merge IP blocks introduce significant throughput penalty to the whole operation.

7.2.2. Throughput measurements

The throughputs have been calculated using the aforementioned latencies, and using equation (1) (as defined in section 5.6). These results are presented in Figure 55.

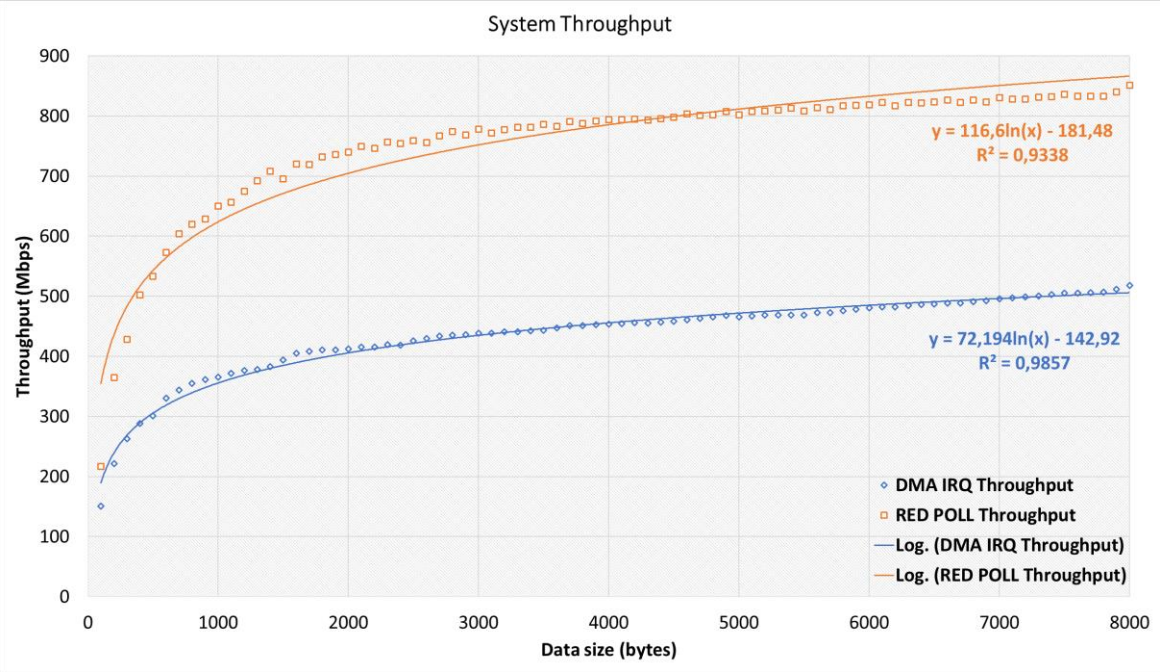

Figure 55. - System throughput estimation

Evidently, the calculated throughputs are higher for each of the lines, as we are not including the splitting and merging phases. Both throughput estimations have logarithmic behaviour, since the MapReduce accelerator requires a minimum time delay d_{node}, and

therefore does not behave linearly when $d_{node} \cong Latency$. Once $Latency \gg d_{node}$, the throughput has a linear behaviour, which occurs at ~ 1500 bytes, as shown by Figure 55.

The complete MapReduce worker throughput can be estimated at 500 Mbps with the line throughput being estimated at 850 Mbps. Lastly, we can argue that transmissions of poor amounts of data have a delay time such that it is close to d_{node}, and therefore are not efficient.

7.3. Conclusions and system validation

An extract of the obtained data of one of the test keys is presented in Figure 56.

```
--- Entering main() ---
[Plat] Starting platform setup...
[Map] Map 0 initialized.
[Map] Map 1 initialized.
[Map] Map 2 initialized.
[Map] Map 3 initialized.
[Map] Map 4 initialized.
[Map] Map 5 initialized.
[Map] Map 6 initialized.
[Map] Map 7 initialized.
[Red] Reduce 0 initialized.
[Red] Reduce 1 initialized.
[Red] Reduce 2 initialized.
[Red] Reduce 3 initialized.
[Red] Reduce 4 initialized.
[Red] Reduce 5 initialized.
[Red] Reduce 6 initialized.
[Red] Reduce 7 initialized.
[Spl] Split 0 initialized.
[Mer] Merge 0 initialized.
[DMA] DMA initialization.
[DMA] DMA initialized.
[DMA] DMA interrupt system is up.
[DMA] DMA interrupts enabled in both directions.
[Plat] Platform setup complete.
[Main] ~~~~ New run:
[Util] Buffer contents:
   This is a test phrase used to estimate latency and throughput parameters
[Util] Number of printed bytes: 73
[Main] Transfer complete.
THIS    1
IS      1
A       1
TEST    1
PHRASE  1
USED    1
TO      1
ESTIMATE        1
LATENCY 1
AND     1
THROUGHPUT      1
PARAMETERS      1

[Util] 5.507500  us.
[Main] 73        bytes.
[Main] 106.037 Mbps.

--- Exiting main() ---
```

Figure 56. - Word Count execution sample

The formatted data is uppercased as to avoid non-merging of two keys that are expressed in either upper- or lowercase. The data itself is formatted such that a key is

separated from its value by a tab character, making it easy to read both by the user and text parsing machines.

We must stress that, since we did not use a Shuffle/Sort function or IP block between the Map and the Reduce stage, it may happen that two identical keys are not sent to the same IP block to be merged together. As an example, consider a KV pair “THE 15” mapped by Mapper_0 and another pair “THE 4” mapped by a neighbouring Map IP block. If the mapped KV pairs are shuffled and sorted before being sent to the Reduce block, the same block oversees merging those two KV pairs into “THE 19”. As we do not employ such a solution, two Reduce blocks in our design process those KV pairs and send them to the Merge IP, thus presenting the KV pair “THE 15” and “THE 4” separately.

Despite not using a shuffle/sort IP, we confirm after careful consideration, that the system produces the expected output in an acceptable amount of time that allows it to be placed into high performance scenario’s.

Regarding the computational delay introduced by the MapReduce Hardware Accelerator, we conclude that the measured latencies produce a data throughput acceptable for High-Throughput environments.

For a maximum sample size of $8\ \mathrm{KB}$, the worker latency rises to $127\ \mu\mathrm{s}$ while the Map-Reduce pair lines have a $75\ \mu\mathrm{s}$ latency. These result in throughputs of $500\ \mathrm{Mbps}$ and $850\ \mathrm{Mbps}$, respectively.

The created application that oversees the hardware accelerator core, despite it using previously in-memory allocated keys, has been built such that it is easy to handle, rewrite and use to produce a Word Count formatted output.

The entire MapReduce framework was designed keeping in mind that the ultimate application is interchangeable, laying down the groundwork for MapReduce based applications in other fields. For a new application to be inserted upon the MapReduce worker, one must only revise the Map and Reduce main function bodies as well as the Split IP separation criterion.

Chapter 8. RESULT ANALYSIS, CONCLUSIONS AND FUTURE WORK

This final chapter encompasses the final conclusions obtained during the development of this thesis. First, we cover the overall performance results regarding the functionality, the latency and the throughput of the designed MapReduce worker. On a second note, we use this information to make improvement proposals that can be applied upon our work. As a matter of context, we then compare the results of our work with similar solutions in the field. Lastly, we close this chapter by stating the overall conclusions and making suggestions for future developments that may use this work as a mean of developing solutions in the field of FPGA based AP SoCs using HLS methodologies.

8.1. OVERALL RESULTS AND CONCLUSIONS

This work proposed the implementation of a MapReduce worker platform that responds to a Big Data application on a Xilinx Zynq AP SoC. Explicitly, the objective of this thesis is the development of the underlying acceleration Kernels required in both, hardware and software, to implement the final solution. The solution itself was defined as a MapReduce based Word Count application, and allows for the design of a basic MapReduce framework on a heterogeneous system, as it is our Zynq AP SoC. The final solution ought to comply with low-latency and high-throughput standards, as it works in the HPC field.

Having reached the end of this thesis, we can conclude the design of such a MapReduce based worker, with the Word Count accelerator core embedded as a hardware solution on the FPGA, and the data handling constructs in software, governed by the PS. The system itself employs previously in-memory allocated data, as we linked the worker's I/O streams to a local DMA.

The worker itself can be summarized by having the following characteristics:

I. The designed hardware **accelerating core** uses a **MapReduce based framework**, employing a data split, map, reduce and data merging phase for high throughput data processing. Neither a combiner nor a shuffle/sort stage has been included.

II. The designed worker employs a **Key-Value based data protocol** for both memory storage and streaming purposes between IP blocks.

III. The **worker** itself **is adapted for character parsing solutions, specifically for Word Count purposes**. Changing the worker's application (i.e. Linear Regression, Kmeans, Histogram) requires a re-design of both the Map and the Reduce IP blocks. The Split IP block requires a minor change, which is the splitting criterion.

IV. **The maximum amount of data a worker can support is** $8\ \mathrm{KB}$. Although it is true that the worker could accept indefinite data as long as its FIFOs are being read, the Reduce stage has limited memory, as it needs to compare the incoming Key-Value pairs to the ones stored in memory.

V. **The accepted characters to compose keys** (words), **are lowercase or uppercase alphabetic letters**. In ASCII, accepted keys are composed of the codes 65 to 90 ('A' to 'Z') and 97 to 122 ('a' to 'z'). Alphanumeric words are therefore not accepted as a complete key; the non-alphabetic component is ignored and considered a separating character.

VI. **Letter casing is not relevant**, as incoming words are uppercased by the Word Count mechanism.

VII. Considering that a line-latency regards the processing delay for a Map-Reduce IP pair, and that the entire worker's latency includes all the line latencies as well as the splitting and merging action, we conclude:

- **A maximum line latency of** $75\ \mu\mathrm{s}$ **and a throughput of** $850\ \mathrm{Mbps}$.
- **A maximum worker latency of** $127\ \mu\mathrm{s}$ **and a throughput of** $500\ \mathrm{Mbps}$.

VIII. Data traversing the MapReduce worker suffer an overall minimum d_{node} latency of 8 ns, suggesting that the worker is at its least efficiency when the data processing delay is $\sim d_{node}$.

8.2. IMPROVEMENT SUGGESTIONS AND FUTURE DEVELOPMENTS

In the spirit of concluding this document and closing the work done in this Master's Thesis, we make improvement suggestions of the designed system, and take the opportunity to make recommendations for future endeavours that may encompass this work or elements of it.

8.2.1. Throughput issues

The first issue we want to address is the throughput throttling elements throughout the whole design.

The first of these elements are the Split IP's and the Merge IP's. It is obvious that these comprise crucial nodes, with regards to the latency and throughput measurements of the worker. As the split and merge elements handle the streams of our worker to parallelize the computation, their latencies ought to be the lowest of the whole design, something we were not able to achieve during this thesis.

The second of these elements, is the Reduce IP. This stage is the most crucial of the entire MapReduce worker, as a result of the reduce phase requiring to store the entire Key-Value registry into memory, iterating it, and finding and merging KV pairs. This IP block is the only one with such a behaviour, as the others employ a dataflow type behaviour. This meant that we had to use drastic optimization solutions for the Reduce IP, with regards to lowering computation latency, at the cost of high fan-out's and a high logic memory footprint.

In the spirit of reducing the processing delay, we suggest using a lower level of abstraction with regards to the algorithmic description of these IP's. We suggest using a SystemC based description to achieve low-latency designs that increase the worker's overall performance, tuning the description to be closer to a RTL level.

Regarding the Reduce IP, on top of the SystemC design suggestion, we propose a hash table storage structure for Key-Value pairs in between the Map and the Reduce phase, to reduce lookup and transfer related latency [21], [36], [37].

Lastly, we do also suggest a study of the AMBA AXI Stream interface width, as we employed a Byte1-Byte wide stream interface. We used such a width, as most of the designed IP blocks use a Byte-by-Byte processing method to analyse the characters that compose the stream. By doing so, we eliminated data type conversion latency, since the data must be registered into a buffer and consequently processed *per* Byte. We still encourage a study of a MapReduce worker with increased width, i.e. 64-bit's width for all the interfaces, expecting to increase throughput.

8.2.2. Memory limitations

The Reduce IP block is the main limiting factor with respect to the memory allocation of the KV pairs, and is the most programmable logic consuming IP of the whole design. Since we are using a great number of instances of these IP's, we suggest the usage of a memory storage solution, that allows storing KV pairs in the least memory available. Such a solution may include hash based storage buffers, such as Bloom Filters (BF) [38].

A second approach to reduce memory footprint of the Reduce IP, suggests not using buffers where KV pairs are stored adjacently but concurrently, therefore employing the available buffer in a more efficient manner. This means, instead of storing an example key 'Key_0' into a memory address A, and 'Key_1' into address B, we suggest storing both Keys concurrently into the same address and copy the remainder into the next. By applying this, less memory is left unused and therefore more keys and values can be stored into the registry.

8.2.3. Parallelism

Out of simplicity, disregarding the already mentioned throughput issues with the Split and Merge IP's, we propose the usage of multiple MapReduce workers in a concurrent manner, therefore increasing the throughput of the whole system.

For our system, the total programmable logic footprint depends on the total LUTRAM usage of a $32.05\ \%$, therefore allowing up to 3 MapReduce workers on the Zynq ZC706 device. Ideally, since our system has a maximum throughput of $850\ \text{Mbps}$

considering line latency, and a throughput of 500 Mbps for the entire worker, one could achieve throughputs of $2{,}550$ Mbps and 1500 Mbps, respectively.

8.2.4. Software application improvement

The developed standalone application is designed for a simple MapReduce and Word Count based hardware accelerator. This hardware core is not fault-tolerant, as the underlying IP's have no means of contacting the PS about a misbehaviour of the architecture, neither is the solution work balancing as the individual IP's have no specified protocol nor interface to report statistical inferences.

For these remarks to be included in a new and more complex solution, it is required to alter the software application such that it interprets data received from the underlying IP's, although it is true that for this to happen the hardware IP's must include a mean of communicating with the PS side of the SoC.

8.2.5. Architecture modifications

One of the main shortcomings of this thesis, is the lack of a sort/shuffle IP or function that sorts the recently mapped Key-Value pairs such that the Reduce stage may process the KV pairs accordingly and without any errors and repetitions. We therefore suggest the creation of either a hard IP sort/shuffle block, that allows to interface between the Map and the Reduce stage for the proper sorting of the KV pairs. This solution is not limited to be integrated into a hardware IP, but can also be designed as part of a software solution, always considering the expenses to be made when sending data from the PL section to the PS to shuffle and sort, and send it back.

A second modification may build upon the MapReduce worker itself, as we designed this system to employ a simple Word Count application. This does not limit the MapReduce worker, as we recommend remodelling the relevant IP's to integrate another application, such as Kmeans, Linear Regression, PCA, or else.

A third idea arises when considering the original MapReduce implementations, where a combiner function is used after the Map stage to relieve computation off the Reduce stage. This could also be a future modification, including a hardware or software implementation of a combiner function, although we suggest a hardware solution to avoid a bottleneck when sending data back from the MapReduce line to the PS side of the SoC.

Lastly, and we find it to be one of the most relevant suggestions, is the modification of the data input and output interfaces. As for now the implementation connects the MapReduce worker with a DMA that switches the data from and to memory. As the DMA only works at a maximum speed of 240 MHz for AXI Stream and 200 MHz for AXI-Lite interfaces [39], we suggest changing the architecture integrating an interface whose throughput is higher than a DMA, e.g., PCIe or GigE interfaces, keeping in mind that such an implementation calls for the extension of the implementation to an architecture that allows an external host to communicate with the Zynq AP SoC through either of those interfaces.

8.3. Closing statements

Having arrived at this point of the current thesis, we can safely conclude the design and implementation of a MapReduce worker on a Xilinx Zynq FPGA based SoC, implemented for a Word Count application and adapted for high-throughput environments.

Despite having found numerous issues and irregularities along the development of this work, such as inconsistent Vivado HLS' logic synthesis result estimations or RTL behavioural's descriptions that do not adhere to their C/C++ algorithmic description, none of the results remain unexpected. We can state that the designed hardware accelerator core works exactly as we have predicted during the design phase, although much improvement has been made during the modelling of the solutions.

Regarding the performance estimations of our MapReduce worker, we can reiterate that FPGA based SoCs make up an optimal solution for high-throughput environments, as it is the case in the Big Data field, with a minimal cost in power consumption and memory footprint. However, it is crucial to understand the underlying architecture of the employed devices to be able to create optimal hardware based solutions, let it be designed through HLS methodologies or classic RTL design.

REFERENCES

[1] K. Adam, I. Hammad, M. Adam, I. Fakharaldien, and M. A. Majid, “Big Data Analysis and Storage,” *Proceedings of the 2015 International Conference on Operations Excellence and Service Engineering*, pp. 648–659, 2015 [Online]. Available: http://umpir.ump.edu.my/7341/

[2] T. J. Barnett, A. Sumits, S. Jain, U. Andra, and T. Khurana, “Cisco Visual Networking Index (VNI) and VNI Service Adoption - Global Forecast Update, 2015-2020,” 2016 [Online]. Available: https://goo.gl/lYgCtd

[3] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. Gódor, G. Szabó, and T. Westholm, “Deep packet inspection tools and techniques in commodity platforms: Challenges and trends,” *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1863–1878, 2012 [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804512001622

[4] C. Kachris and G. C. Sirakoulis, “A Reconfigurable MapReduce Accelerator for multi-core all-programmable SoCs,” *2014 International Symposium on System-on-Chip (SoC)*, 2014 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6972430/

[5] Y. Shan, J. Yan, Y. Wang, and N. Xu, “FPMR : MapReduce Framework on FPGA A Case Study of RankBoost Acceleration,” *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 93–102, 2010 [Online]. Available: https://dl.acm.org/citation.cfm?id=1723129

[6] D. Diamantopoulos and C. Kachris, “High-level synthesizable dataflow MapReduce accelerator for FPGA-coupled data centers,” *Proceedings - 2015 International Conference on Embedded Computer Systems: Architectures, Modeling and*

Simulation, SAMOS 2015, no. Samos Xv, pp. 26–33, 2015 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7363656/

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004 [Online]. Available: http://www.engr.scu.edu/~mwang2/projects/MapReduce_cluster_13s.pdf

[8] J. Shuai Che and Jie Li and Sheaffer, J.W. and Skadron, K. and Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101–107, 2008 [Online]. Available: ieeexplore.ieee.org/abstract/document/4570793/

[9] A. Dollas, "Big data processing with FPGA supercomputers: Opportunities and challenges," *Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI*, pp. 474–479, 2014 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6903409/

[10] B. Vega, P. P. Carballo, and P. Hernández-fernández, "TCP / IP Packets Analyzer based on Configurable Zynq Platform," pp. 2–5, 2015.

[11] Mike Santarini, "How Xilinx Halved Power Draw in 7 Series FPGAs," *Xilinx Xcell Journal*, no. 76, pp. 9–15, 2011 [Online]. Available: www.xilinx.com/publications/archives/xcell/Xcell76.pdf

[12] Xilinx Inc., "A Generation Ahead for Smarter Systems: 9 Reasons why the Xilinx Zynq-7000 all Programmable SoC Platfom is the Smartest Solution," 2014 [Online]. Available: http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf

[13] Xilinx Inc., "Zynq-7000 All Programmable SoC Overview," 2016 [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[14] Daniele Bagni, A. Di Fresco, J. Noguera, and F. M. Vallina, "A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS," 2016 [Online]. Available:

http://china.zylinks.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf

[15] Xilinx, "Zynq-7000 All Programmable SoC ZC706 Evaluation Kit Getting Started Guide," 2015 [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwjHjsX-n4TVAhVBPBoKHUsxBWwQFgg0MAA&url=https%3A%2F%2Fwww.xilinx.com%2Fsupport%2Fdocumentation%2Fboards_and_kits%2Fzc706%2F2014_4%2Fug961-zc706-GSG.pdf&usg=AFQjCNHQ6xy

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 13–24, 2007 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/4147644/

[17] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for Shared-Memory Systems," *MapReduce '11 Proceedings of the second international workshop on MapReduce and its applications*, pp. 9–16, 2011 [Online]. Available: http://csl.stanford.edu/~christos/publications/2011.phoenixplus.mapreduce.pdf

[18] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the MapReduce framework on Intel Xeon Phi coprocessor," *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pp. 125–130, 2013 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6691563/

[19] J. Cao, H. Cui, H. Shi, and L. Jiao, "Big data: A parallel particle swarm optimization-back-propagation neural network algorithm based on MapReduce," *PLoS ONE*, vol. 11, no. 6, pp. 1–17, 2016 [Online]. Available: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0157551

[20] M. Aly, É. P. De Montréal, Y. Shaban, and D. Ph, "Analysis of Massive Industrial Data using MapReduce Framework for Parallel Processing," *Reliability and Maintainability Symposium (RAMS), 2017 Annual*, 2017 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7889681/

[21] Z. Wang, S. Zhang, B. He, and W. Zhang, “Melia: A MapReduce Framework on FPGAs, OpenCL-based FPGAs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9219, no. c, pp. 1–14, 2016 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7425227/

[22] E. Ghasemi and P. Chow, “Accelerating Apache Spark Big Data Analysis with FPGAs,” *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016 [Online]. Available: http://ieeexplore.ieee.org/document/7816915/

[23] K. Neshatpour, A. Sasan, and H. Homayoun, “Big data analytics on heterogeneous accelerator architectures,” *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. pp. 1–3, 2016 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7750986/

[24] N. Elgendy and A. Elragal, “Big Data Analytics: A Literature Review Paper,” *Advances in Data Mining. Applications and Theoretical Aspects*, vol. 8557, pp. 214–227, 2014 [Online]. Available: http://link.springer.com/10.1007/978-3-319-08976-8_16

[25] X. Wu, X. Zhu, and S. Member, “Data Mining with Big Data,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 1, pp. 97–107, 2014 [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6547630/

[26] Xilinx, “Zynq-7000 All Programmable SoC Data Sheet: Overview,” 2017 [Online]. Available: https://goo.gl/Gq4L5E

[27] Xilinx, “ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC: User Guide,” 2016 [Online]. Available: https://goo.gl/QjUiQo

[28] Xilinx, “UltraFast High-Level Productivity Design Methodology Guide,” 2017 [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf

[29] Xilinx, “UltraFast Design Methodology Guide for the Vivado Design Suite,” 2016 [Online]. Available: https://goo.gl/VcoECN

[30] Xilinx Inc., “Vivado Design Suite User Guide - Design Flows Overview,” *Ug892*, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug903-vivado-using-constraints.pdf

[31] Xilinx, “Vivado Design Suite User Guide: High-Level Synthesis,” 2017 [Online]. Available: https://goo.gl/csGHxv

[32] Xilinx, “Vivado Design Suite User Guide: Implementation,” *UG904*, 2017 [Online]. Available: https://goo.gl/g8juRh

[33] ARM, “AMBA 4 AXI4-Stream,” 2010 [Online]. Available: https://silver.arm.com/download/download.tm?pv=1074010

[34] P. P. Carballo, “Aportaciones a la metodología de diseño basada en síntesis de alto nivel. Aplicaciones al diseño de IPs para procesado de eventos complejos y codificación de vídeo,” 2016 [Online]. Available: https://acceda.ulpgc.es/handle/10553/18431

[35] Xilinx Inc., “Xilinx Software Development Kit (SDK),” *UG1145*, 2016. [Online]. Available: http://www.xilinx.com/tools/sdk.htm

[36] S. J. Matthews, “Using Phoenix++ MapReduce to Introduce Undergraduate Students to Parallel Computing,” *J. Comput. Sci. Coll.*, vol. 32, no. 6, pp. 165–174, 2017 [Online]. Available: http://dl.acm.org/citation.cfm?id=3069658.3069682

[37] E. Ghasemi, “A Scalable Heterogeneous Dataflow Architecture For Big Data Analytics Using FPGAs,” University of Toronto, 2015 [Online]. Available: https://dl.acm.org/citation.cfm?id=2847294

[38] A. Cuzzocrea, M. Cosulschi, and R. de Virgilio, “An Effective and Efficient MapReduce Algorithm for Computing BFS-Based Traversals of Large-Scale RDF Graphs,” *Algorithms*, vol. 9, no. 1, 2016 [Online]. Available: http://www.mdpi.com/1999-4893/9/1/7

[39] Xilinx, “AXI DMA v7.1,” *LogiCORE IP Product Guide*, 2016 [Online]. Available: https://goo.gl/MJQTyw