SYSTEM-ON-CHIP MEMORY DESIGN FOR A DOMAIN-SPECIFIC RISC-V PROCESSOR

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Utku Gülgeç
May 2023

SYSTEM-ON-CHIP MEMORY DESIGN FOR A DOMAIN-SPECIFIC RISC-V PROCESSOR
By Utku Gülgeç
May 2023

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Süleyman Tosun

Uğur Güdükbay

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

SYSTEM-ON-CHIP MEMORY DESIGN FOR A DOMAIN-SPECIFIC RISC-V PROCESSOR

Utku Gülgeç
M.S. in Computer Engineering
Advisor: Özcan Öztürk
May 2023

The use of graph applications is common in many areas; however, irregular and data-driven memory access patterns combined with the large sizes of graph data results in performance loss in general-purpose computing systems. Existing studies proposed hardware accelerators often implemented on FPGAs to alleviate performance problems and improve energy efficiency while having less emphasis on programmability and flexibility. This thesis presents a hardware implementation of a domain-specific processor design for graph applications on a system-on-chip (SoC) platform accompanied by a design of a memory framework. The proposed system architecture improves the micro-architecture of a baseline design, integrates the baseline with an efficient system-on-chip bus communication protocol, and compares alternative memory framework implementations. The hardware is implemented on a state-of-the-art evaluation board. Popular graph benchmarks are used for performance evaluations of the implemented system, and various sensitivity analysis is done on newly added system parameters to determine the optimal system configuration. An analysis of power consumption and resource utilization is also provided. Overall, average speed-ups vary between 15% and 25% depending on the benchmark and graph data, while on-chip power consumption varies between 3.8 to 4.2 Watts depending on the system clock frequency.

Keywords: System-on-Chip, Zynq, Graph Applications, Processor.

ÖZET

ALANA ÖZGÜ RISC-V İŞLEMCİSİ İÇİN ÇİP-ÜSTÜ-SİSTEM'DE BELLEK TASARIMI

Utku Gülgeç
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Mayıs 2023

Çizge uygulamalarının kullanımı bir çok alanda yaygındır ancak düzensiz ve veriye dayalı bellek erişim kalıpları, büyük boyutlu çizge verisi ile birleşince, genel amaçlı bilgi işlem sistemleri çizge işlemede düşük performans göstermektedir. Varolan çalışmalarda, FPGA temelli donanım hızlandırıcıları, performans problemlerini azaltmak ve enerji verimliliğini artırmak için önerilirken programlanabilirlik ve esneklik konuları arka planda kalmıştır. Bu tez, alana özgü bir işlemci tasarımının, bir Çip-Üstü-Sistem platformunda donanımsal olarak gerçeklenmesini ve buna eşlik eden bir bellek çatısı tasarımını sunmaktadır. Önerilen mimari, taban alınan tasarımın mikro-mimarisini iyileştirmekte, taban tasarımı Sistem-Üstü-Çip'e bir çip-üzeri veri iletişim yolu protokolü kullanarak entegre etmekte ve alternatif bellek çatılarının tasarımlarını karşılaştırmaktadır. Donanım gerçeklenmesi, son teknoloji bir değerlendirme kartında yapılmıştır. Popüler çizge denektaşları, tasarlanan sistemin performans ölçümlerinde kullanılmış ve en iyi performans çıktısı için çeşitli sistem parametreleri üzerinde hassasiyet analizi yapılmıştır. Güç tüketimi ve kaynak kullanımı üzerine bir analiz de çalışmaya dahil edilmiştir. Genel itibariyle, denektaşına ve çizge verisine göre değişmekle beraber, averaj hızlanma değerleri %15 ile %25 arasında ölçülürken, çip-üzeri güç tüketim değeri 3.8 ile 4.2 Watt olarak ölçülmüştür.

Anahtar sözcükler: Çip-Üstü-Sistem, Zynq, Çizge Uygulamaları, İşlemci.

Acknowledgement

This work has been supported in part by the Scientific and Technological Research Council of Turkey (TUBITAK) 1001 program through the EEEAG 119E559 project.

I wish to express my appreciation to my academic advisor, Prof. Özcan Öztürk, for giving me the opportunity to work on this project and for invaluable guidance. Also, I would like to thank jury members Prof. Süleyman Tosun and Prof. Uğur Güdükbay for their time and patience in reading this thesis and for providing me with valuable feedback. Finally, I would like to thank my beloved family and my friends from Bilkent University and Middle East Technical University for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph processing applications can be found in many fields [1], such as data science, medical applications, social network analysis, and databases. Although the use of graph structures comes in handy in various problems, by their nature, graph applications' execution time and power usage increase considerably as data size increases and applications become more complex.

There are several challenges present for the efficient processing of graph structures. For example, graph applications usually have irregular memory access patterns, making it difficult to utilize cache systems [2, 3]. Moreover, real-life graph data can be immense, and memory footprints of graph computations are large [2]. Such complications make graph processing in general-purpose processors (GPP) cumbersome and pave the way for considering Domain-Specific Architecture. It is also worth mentioning that industry workloads are often small sets of repetitive tasks which can benefit from custom processing units that execute them [2].

In literature, several studies touch on the subject. For example, Gui et al. examined graph processing accelerators in their survey [3]. They highlighted that, even though many software solutions are available, graph processing performance potential and energy consumption are still limited by the hardware. Thus, dedicated hardware accelerators come in handy. Field Programmable Gate Array

(FPGA)-based or Application Specific Integrated Circuit (ASIC)-based designs are usually the primary choice for hardware accelerators.

ASIC-based designs have high performance and small power consumption but have poor flexibility. Another design approach called Application-Specific Instruction Processor (ASIP) compromises two extremes [4], ASIC and GPP, which emanates as an alternative methodology. An ASIP is designed by adding custom instructions to the processor. In our work, we leveraged the ASIP design methodology. In previous efforts, a RISC-V instruction set architecture-based custom architecture [5] and software support with a novel micro-architectural design [6] were proposed for iterative graph applications.

1.1 Objective of the Thesis

The main objective of the thesis is to implement a custom graph processor design on a real hardware platform. A System-on-Chip (SoC) platform is used to realize the graph processor on the hardware. Furthermore, a memory system utilizing embedded resources of the hardware platform is added to the system design. The final design of the custom processor is tested on large real-life graph data, leading to meaningful performance evaluations.

The main contributions of this thesis to the previous work on the custom graph processor are:

- implementation of the graph processor on hardware by utilizing dedicated SoC resources,
- proposing a memory framework that can be implemented and tested in real execution scenarios,
- performance evaluations with large-scale graph structures,
- providing an energy consumption analysis.

1.2 Organization of the Thesis

The necessary background information on the work conducted in the thesis is given in Chapter 2. This chapter contains a summary of graph applications and an overview of the hardware platform used in the implementation process.

Chapter 3 provides a literature review on hardware accelerators and custom processors to present related previous work.

Chapter 4 describes the details of the SoC implementation of the custom processor and the proposed memory system, which are the main technical contributions of the thesis.

Chapter 5 presents the experimental evaluation of the final system design. Experimental setup, benchmarks, graph data, and design parameters are included in this chapter. Also, the effects of selected design parameters on overall performance are evaluated. Moreover, the power analysis of the platform and the main factors affecting energy consumption are explored.

Finally, the thesis is concluded in Chapter 6 with possible future work and cost-benefit analysis.

Chapter 2

Background

2.1 Graph Applications

A graph consists of a set of vertices and a set of edges. Graph data structures are mainly used for modeling some relation between objects. Graph G can be defined as $G = (V, E)$ where V represents vertices and E represents edges. Algorithms running on a graph can be expressed in two main programming models [3]. In Vertex-centric Programming Model, graph computations are described for each vertex, and since vertices can be processed independently, parallelism can be leveraged. In Edge-centric Programming Model, the graph program is designed to collect information from connected vertices for each edge.

Several graph representations can be utilized for graph algorithms. Edge-List, Adjacency-Matrix, and Adjacency-List are common data structures used for graph representation. An edge list is a basic representation that lists off pairs of vertices. A form of edge-list representation called Edge-array [3] is used in edge-centric graph processors. In an edge array, each element of the array contains an edge. Although this representation allows sequential access to the edges, vertex access is random. Adjacency-List and Adjacency-Matrix are similar graph representations. Adjacency-Matrix stores a graph of n nodes in $n \times n$ matrix,

which makes this approach efficient for storing dense graphs. On the other hand, Adjacency-List keeps vertices in an array, where each vertex in the array stores a pointer to a linked list of edges. Adjacency-List is memory efficient since only existing edges are stored, and it is mainly used for representing sparse graphs [7]. In fact, most real-world graphs, such as social and web network graphs, are globally sparse but may exhibit locally dense regions. [8, 9]

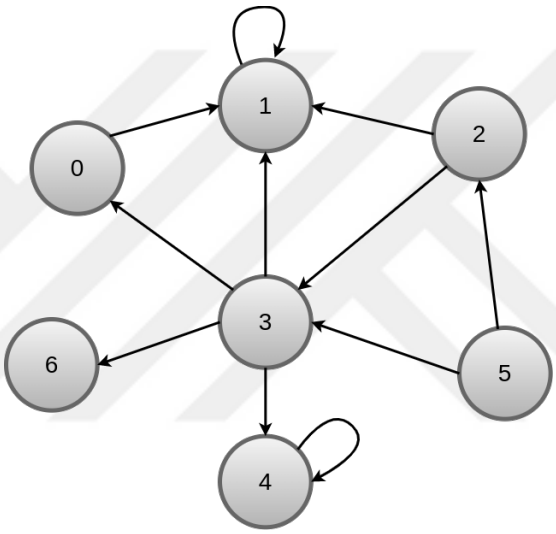
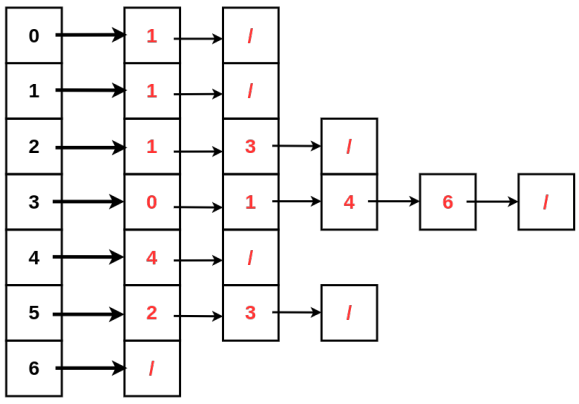


Figure 2.1: A basic directed graph with 7 vertices and 11 edges.

	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	0	0	0	0	0
2	0	1	0	1	0	0	0
3	1	1	0	0	1	0	1
4	0	0	0	0	1	0	0
5	0	0	1	1	0	0	0
6	0	0	0	0	0	0	0

(a) Adjacency-Matrix



(b) Adjacency-List

Figure 2.2: Different representations of the same graph given in Figure 2.1.

Figure 2.2 shows the different representations of the graph given in Figure 2.1. Note that we used more storage in Adjacency-Matrix due to representing

non-existent edges.

Compressed Sparse Row (CSR) is another popular representation format to store sparse graphs. Two arrays store a sparse graph in CSR representation: vertex (rows) array and edge (columns) array. Also, two additional arrays can be used to store vertex and edge attributes; data and weight array. Each cell in the vertex array stores the starting offset in the edge array, where the list of the outgoing neighbors of the vertex is stored. CSR format stores a graph G with V vertices and E edges in size $O(|V| + |E|)$. CSR representation has a spatial locality benefit for the iterative applications since elements in CSR arrays are stored densely and suitable for consecutive access[10, 11], however; CSR is inefficient in terms of graph mutations since it needs to be rebuilt after updates [7, 12]. Figure 2.3 shows the color-coded CSR representation of the sample graph in Figure 2.1.

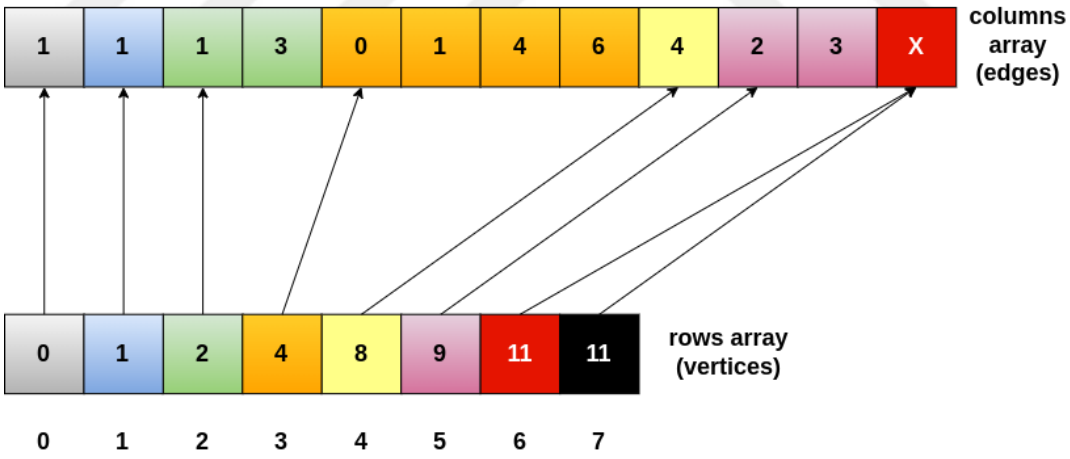


Figure 2.3: CSR representation of the graph given in Figure 2.1.

Algorithm 1 General description of vertex-centric graph algorithms, using arbitrary work-list and update function f on a graph G represented by CSR.

1: **Input** $G = (rows, columns, data, weights)$
2: **for all** vertex index $i \in work - list$ **do**
3: **for all** $(j \leftarrow columns[rows[i]]$ **to** $columns[rows[i+1]]$ **do**
4: $f(G, i, j)$
5: **end for**
6: **end for**

As CSR can be cache friendly, it is suitable to use with graph algorithms described in a vertex-centric manner. However, poor cache locality is often the case due to graph algorithms' irregular and data-driven nature. Algorithm 1 approximately depicts vertex processing in graphs represented by CSR. A work-list is the list of vertices that needs to be processed. For iterative algorithms, work-list can be all vertices, but it can also be a queue for a breadth-first search (BFS) algorithm or a stack for a depth-first search (DFS) algorithm.

Consider the access patterns in Algorithm 1. Since vertices in the work list are not necessarily in sequential order, vertex access can be random. On the other hand, the edge list for each vertex index in the work list is accessed sequentially. Another random access pattern will likely emerge if we want to access neighboring vertices and their data. It is also worth noting that the topological structure of a graph also affects access patterns.

For example, consider the CSR representation of the graph in Figure 2.3 with the given work-list used with Algorithm 1. In the following figures, each vertex and its neighbors are coded with the same color, and rectangular shapes represent the access footprint of the same colored vertex. The random accesses to vertices can be seen in Figure 2.4 as we process vertices and data in the work-list. On the other hand, access to edges and weights is sequential and can be seen in Figure 2.5. Another set of random accesses occurs as we want to process neighbors of the vertices and their data in the work-list, depending on the graph's topological structure. In Figure 2.6, irregular memory footprints can be seen by rectangles of the same color. Note that temporal locality can be leveraged when we access the data of vertices neighboring themselves.

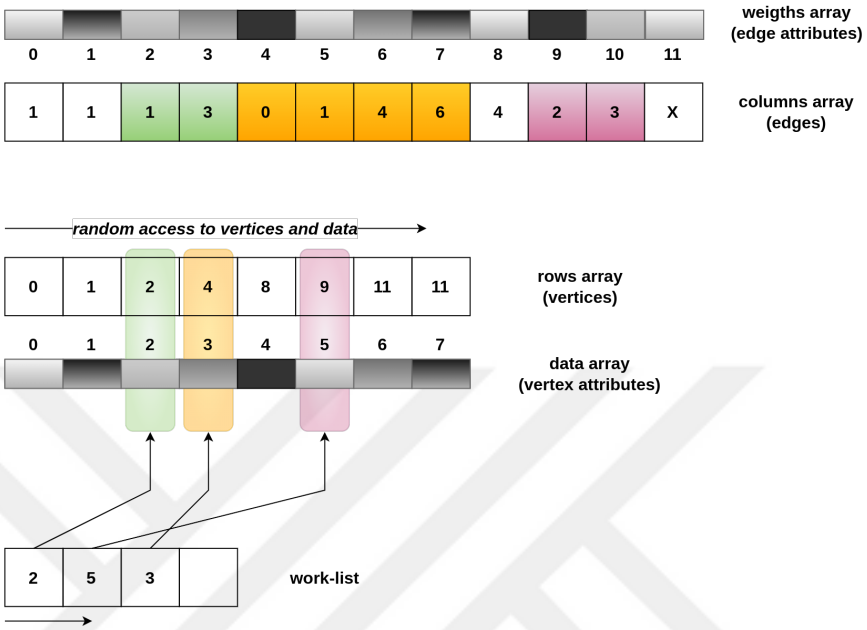


Figure 2.4: Accesses to vertices and data with the given work-list. Since work-list can be in any order, random access to vertices is likely to occur.

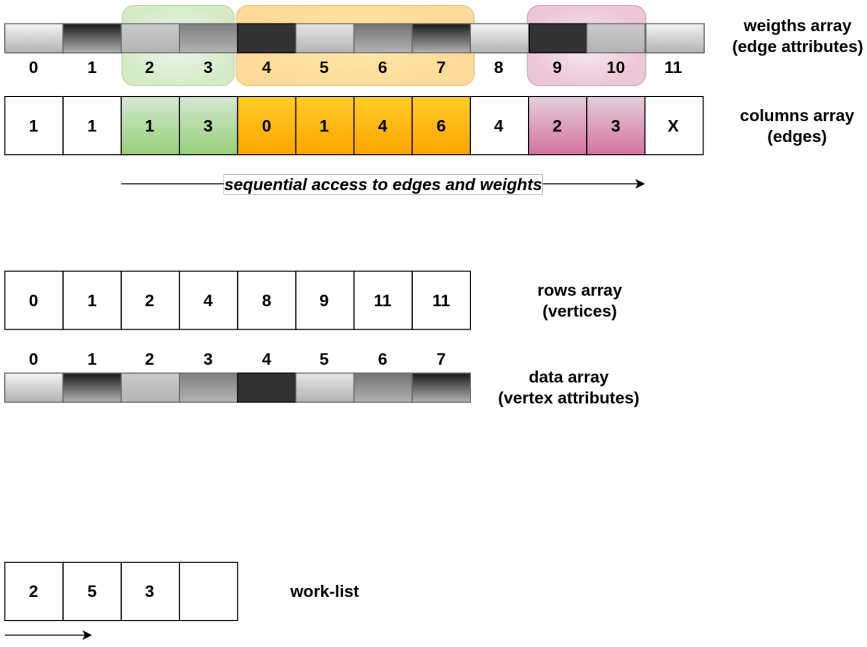


Figure 2.5: Accesses to edges and weights are sequential by the nature of the CSR arrays.

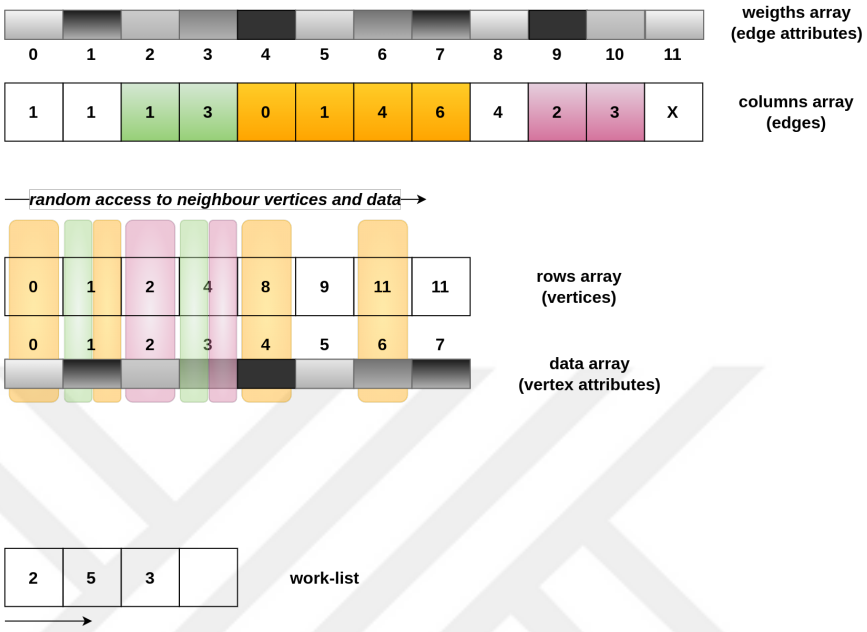


Figure 2.6: Access patterns to neighbor vertices and their data depends on the graph's topological structure.

2.2 Hardware Platform

2.2.1 ZCU102 Evaluation Board

Zynq Ultrascale+ MPSoC[1] (ZynqUs+ MPSoC) is an all-programmable multi-processor-system-on-chip (MPSoC) platform by Xilinx that combines dual or quad ARM Cortex-A53 processors with FPGA fabric[13]. ZynqUs+ MPSoC architecture is divided into two parts: *Programmable Logic (PL)* and *Processing System (PS)* [14]. Hardware configuration is done in the PL part, where primitive design resources are utilized. PS part is used for software configuration. Advanced eXtensible Interface (AXI) is used for interactions between PL and PS. The Zynq device family is frequently used in accelerator designs due to its flexibility and compactness.

[1]https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

ZCU102[2] board contains an XCZU9EG-2FFVB1156E ZynqUs+ MPSoC die, a 4 GB SODIMM DDR4 external RAM, a 512 MB PL DDR RAM, an SD card reader, and various other components [15].

Processing System (PS)	
APU	ARM Quad-core Cortex-A53
DRAM	4 GB DDR4 SODIMM
L1 Cache	32 KB
L2 Cache	1 MB

Table 2.1: Processing System (PS) specifications of ZCU102.

Programmable Logic (PL)	
Flip-Flops	548160
Logic Cells	599550
LUTs	274080
DSP Slices	2520
BRAMs	912
DRAM	512 MB DDR4 Component

Table 2.2: Programmable Logic (PL) specifications of ZCU102.

Table 2.1 and 2.2 describes the technical specifications of the PS and PL parts of the ZynqUs+ chip used in the ZCU102 board. PS side also contains a real-time processing unit and a graphics processing unit; however, they are not used in our work.

- **Look-up Table (LUT)** is a specialized resource element holding a truth table. Logic functions in the design are implemented using LUTs in the PL. LUTs can also be used for storing data in LUTRAM configuration.
- **Flip-flops (FF)** are used for creating registers and storing data.
- **DSP slices** are arithmetic logic units embedded into the FPGA fabric.
- **Block RAMs (BRAMs)** are a special type of memory embedded in the FPGA fabric. BRAMs support independent synchronous read and write operations with a one clock cycle operation latency. In the UltraScale+

[2]https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

Architecture, a BRAM stores up to 36 Kb of data and can be configured as either two independent 18 Kb RAMs or one 36 Kb RAM based on the design choice.

Full-power Domain (FPD), Low-power Domain (LPD), PL-power Domain (PLPD), and Battery-power Domain (BPD) are isolated power islands in the chip[16]. Power domains can also be isolated individually, allowing functional isolation for safety-critical applications. In our work, we used FPD and PLPD components. Power domains could be exploited to investigate more complex designs further. However, we do not explore this direction in the thesis.

PL-PS communication in ZCU102 is achieved via multiple programmable AXI interfaces[16]. PS-PL interfaces can be two-way coherent or one-way (I/O) coherent, or their coherency needs to be maintained by software drivers on the PS side. High-Performance (HP) ports enable low-latency transactions between PL and PS; however, software should maintain data coherency between PS and PL. High-Performance Master (HPM) interfaces allow components such as CPUs and DMAs to be configured from PS, making the PS an AXI master. Accelerator Coherency Port (ACP) is one-way coherent (I/O coherent) with Cache Coherent Interconnect (CCI) with L2 cache allocation. AXI coherency port offers a two-way coherent path between memory in PL and CCI [17].

Abbreviation	Data Width	Addr. Width	Master	Slave
ACP	128	40	PL	PS FPD
ACE	128	40	PL	PS FPD
HPC{0,1}	128	49	PL	PS FPD
HP{0:3}	32/64/128	49	PL	PS FPD
PL_LPD	32/64/128	49	PL	PS LPD
HPM[{0,1}	32/64/128	40	PS FPD	PL
LPD_PL	32/64/128	32	PS LPD	PL

Table 2.3: AXI Peripherals of Zynq Ultrascale+.

Table 2.3 lists AXI peripherals available in ZCU102. In our work, we used a single HP port and maintained data coherency manually by using a software driver provided by Xilinx on the PS side.

2.2.2 Advanced eXtensible Interface (AXI)

AXI is one of the many protocols of the Advanced Microcontroller Bus Architecture (AMBA)[3], a set of on-chip communication specifications introduced by ARM in the late 1990s [18]. AMBA protocols define how functional blocks in an SoC communicate. AMBA's third generation introduced AXI, which is still maintained today. Some of the other publicly available bus architectures can be listed as Avalon (Intel)[19], CoreConnect (IBM)[20], and Wishbone (open-source)[21].

AXI Channels and Signals				
Read Channels		Write Channels		
Read Addr (AR)	Read Data (R)	Write Addr (AW)	Write Data (W)	Write Resp (B)
ARADDR	RDATA	AWADDR	WDATA	BVALID
ARVALID	RVALID	AWVALID	WVALID	BREADY
ARREADY	RREADY	AWREADY	WREADY	BRESP
ARLEN	RRESP	AWLEN	WLAST	
ARSIZE	RLAST	AWSIZE		
ARBURST		AWBURST		
ARCACHE		AWCACHE		

Table 2.4: AXI channels and signals, including burst function-related signals.

The AXI protocol adopts the master-slave paradigm, defining five independent transaction channels between an AXI master and an AXI slave [18]. For write operations, the AXI master initiates the transaction by sending the write address and write data to the AXI slave. After the write operation, the slave responds via Write Response Channel. Read operations are similar to the writes. The AXI master initiates the transaction by sending Read Address to the slave. The slave returns the data via Read Data Channel. The Read Response Channel does not exist since the read response is sent via Read Data Channel. In Table 2.4, read channels with signals are labeled green, and write channels with signals are labeled blue. Moreover, burst configuration signals are labeled red, and cache mode signals are labeled purple. In our work, burst and cache functions are not used.

[3]https://developer.arm.com/Architectures/AMBA

Figure 2.7 shows a typical single-beat AXI master-slave pair and transaction channels.

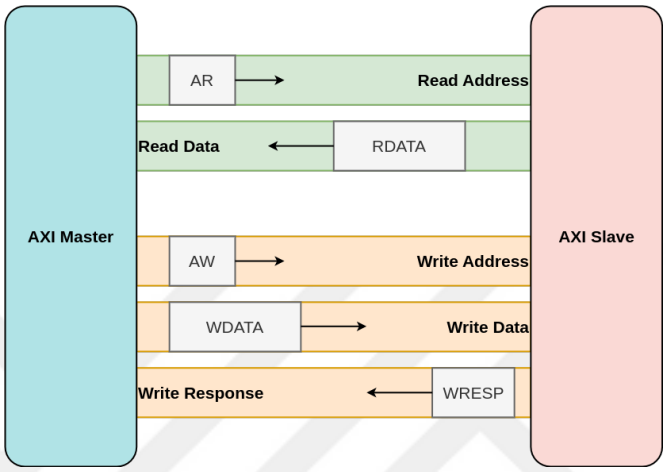


Figure 2.7: Channels between an AXI master and an AXI slave.

AXI transfers are coordinated by ready-valid handshake. The source asserts the *xVALID* signal when valid information is available. The *xREADY* signal is asserted when the destination can accept the information. Figure 2.8 shows the basic behavior of the handshake protocol. An example can be given using the Read Address and Read Data Channels. The master using Read Address Channel is the source since it is the master's responsibility to ensure the read address is valid. On the other hand, the same master will become the destination for the Read Data Channel. [18]

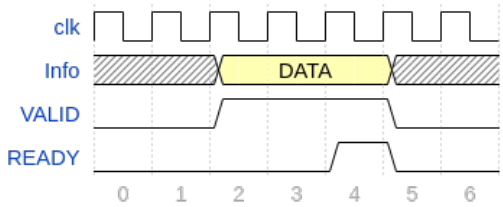


Figure 2.8: Ready/Valid handshake protocol.

In AXI transfers, target information exchange only occurs at the rising edge of the clock and when both *READY* and *VALID* signals are asserted. A source cannot wait for *READY* to be asserted before asserting *VALID*; however, a destination can wait for *VALID* to be asserted before asserting *READY*.

While an AXI transfer is a single exchange of information with a single handshake, an AXI transaction is a burst of transfers involving multiple handshakes. Multiple outstanding transfers are supported in AXI, making parallel transfer processing possible. AXI protocol is also capable of burst transfers, which enables the AXI slave to calculate the following transfer address based on the burst start address provided by the AXI master for faster transfers.

The granularity of the address mapping in AXI is 4KB [18]. In other words, the smallest address space that can be assigned to an AXI slave is 4KB. The 4KB boundary specification restricts AXI burst transfers. That is, a burst transfer should not cross addresses multiples of 4KB. This implies that the last twelve bits of the burst start address should be equal to the last twelve bits of the burst end address. Crossing the boundary may lead to unknown results.

A bus management core called AXI interconnect [22] is usually inserted between a memory-mapped AXI master and a memory-mapped AXI slave. AXI interconnects ensure transactions between master and slave are running properly even when AXI configurations are different. Also, AXI interconnects manage the transactions in case of multiple AXI masters and multiple AXI slaves are present on the same bus. Figure 2.9 shows a typical AXI interconnect core, connecting multiple masters to multiple slaves.

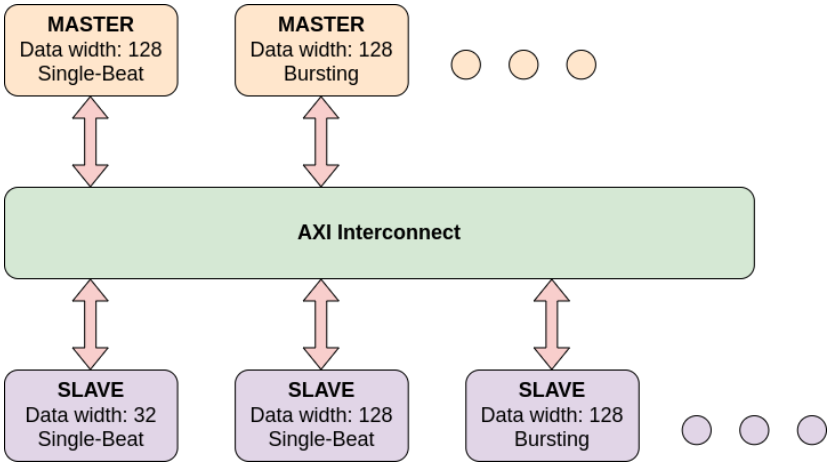


Figure 2.9: Multiple AXI masters and slaves connected via AXI interconnect.

Fourth-generation AMBA introduced AXI4 [23], which divides into three interface options:

1. **AXI4-Full**: AXI4 is a memory-mapped, high-performance interface that supports burst transactions.
2. **AXI4-Lite**: Lightweight and memory-mapped interface mainly used for register accesses. Does not support burst transactions.
3. **AXI4-Stream**: High-performance stream-based interface designed for unidirectional transfers.

AXI protocol is highly customizable, and Xilinx design tools[4] and intellectual properties (IPs) adopted the AXI protocol [24]. Therefore, the graph processors' design blocks are connected with these state-of-the-art AXI interfaces.

[4]https://www.xilinx.com/developer/products/vivado.html

Chapter 3

Related Work

The history of computing started with the need for specialized computing units to perform particular tasks efficiently. General-purpose computers later emerged as a jack of all trades, master of none units that are easy to manufacture, programmable, and accessible to ordinary households. We can list modern specialized hardware as accelerators, Application-Specific and domain-specific processors[25, 26].

3.1 Accelerators

A hardware accelerator is a specialized hardware whose purpose is to implement various functions to perform a set of instructions with higher performance than a general-purpose microprocessor by exploiting four main techniques; data specialization, parallelism, local and optimized memory, and reduced overhead [27].

3.1.1 ASIC-based Accelerators

ASIC is an integrated circuit highly specialized in a requirement. Although ASICs have high performance and low power consumption, they lack programmability [27]. Traditionally, ASIC accelerators are often used for a single function [28]. For example, neurons are directly mapped to the hardware in an ASIC neural network accelerator. Even though Intel's analog circuit neural network chip ETANN [29] is outdated, the scalability problem was obvious. The number of neurons that could be integrated into the chip was 64 due to the manufacturing process.

Another example of an ASIC-based accelerator is ASIC Clouds, proposed by Magaki et al. [30]. Their work examines large arrays of ASIC accelerators built for data centers to optimize chronic workloads as Cloud services are becoming more popular daily.

3.1.2 FPGA-based Accelerators

FPGA is a configurable integrated circuit with high flexibility. FPGAs can be programmed using hardware description languages, making them suitable for rapid prototyping. FPGAs usually operate with low frequencies of $\approx$ 50-200 MHz and have limited memory resources.

FPGA-based accelerator designs can be widely found in deep learning acceleration. For example, the Deep Learning Accelerator Unit (DLAU), running only at the frequency of 200 MHz, demonstrated $\approx$ 36 times speed up at a network of size 256×256 using data sets MNIST and CIFAR-10, compared to a CPU running at 2.3 GHz [31]. Another example using MNIST and CIFAR-10 benchmarks is proposed by Wang et al. [32], which can reach 152 times speed up in throughput by exploitation of pipeline in all operations compared to the baseline, IBM TrueNorth [33] processor. Database systems are another field where FPGA-based accelerators are used. Casper and Olukotun proposed a prototype to accelerate primitive database operations such as selection, merge-join, and sort [34].

3.1.3 GPU-based Accelerators

Graphic Processing Unit (GPU) is a microprocessor specialized in image and graphics-related tasks. Although it is mainly used in personal computers, gaming consoles, and mobile devices, thanks to their parallel processing capability, GPUs are used as a hardware accelerator in fields such as deep learning applications, medical imaging, and informatics [35].

Fast Fourier Transform (FFT) is widely used in Magnetic Resonance Imaging (MRI) reconstruction. Many studies use GPUs for an efficient FFT calculation [36, 37]. GPUs are also suitable for implementing convolutional neural networks (CNN). A 2010 study by Strigl et al. [38] showed GPU could perform 2 to 24 times faster than CPU in tasks related to pattern recognition, such as optical pattern recognition (OCR). Another study by Buber and Diri [39] compared GPU and CPU performance in classifying web pages with a recurrent neural network (RNN). They observed GPU is faster than the CPU in all test settings, and in some settings, GPU is 4 to 5 times faster than the CPU.

3.2 Application-Specific Instruction Processors

Application-Specific Instruction Processors (ASIPs) can perform particular operations efficiently while not compromising flexibility as much as an ASIC [40]. Recent ASIP studies usually involve using a baseline soft microprocessor and extending its instruction set to the needs. For example, an ASIP for advanced encryption standard (AES) is proposed and implemented on an FPGA in the study conducted by Good and Benaissa [41]. Another example is presented for accelerating multimedia applications [42]. This design relies on a 32-bit RISC processor and uses instruction set extension (ISE) techniques to alleviate computationally extensive parts of a multimedia application with custom instructions. A similar approach is used in TamaRISC-CS [43], an ASIP designed for compressed sensing, a technique for compressing sparse signals. The design uses a RISC baseline microprocessor and accomplishes 62 times speed-up in addition

to the ≈12 times power savings compared to the baseline. Lastly, Muller et al. [44] proposed a custom hardware platform consisting of reconfigurable ASIPs, a dedicated memory interface, and an interconnect for turbo decoding algorithms. The design is an ASIP-based multiprocessor platform, and its performance was evaluated for 4, 8, 16, and 32 ASIP cores. The proposed platform demonstrated a 100 Mbit/s throughput in DVB-RCS turbo decoding with a 16-core configuration.

3.3 Domain-specific Processors

Domain-specific processors support a broader spectrum of applications closer to the GPP than an ASIP. GPU and Tensor Processing Unit (TPU) [45] are in this category of processors. TPU is designed for deep learning acceleration by Google. TPUs are deployed in data centers and used to accelerate neural network inference. TPU has 65,536 8-bit multiply-and-add units for matrix operations and a software-managed on-chip memory. TPU is stated as an ASIC, but it has a CISC instruction set and is highly flexible since it can be programmed using Google's TensorFlow framework. TPU's performance evaluation shows it is 15 to 30 times faster than GPU and CPU in its specific domain.

Tensor Streaming Processor (TSP) is another example of a domain-specific processor introduced by Abts et al. [46]. TSP is specialized in the machine learning domain, and its ISA exposes temporal information about each instruction, letting the compiler precisely control each instruction's dispatch time. Performance evaluation suggests TSP can outperform both TPU and GPU in experimented machine learning benchmarks.

3.4 Graph Accelerators

Graph applications can largely benefit from the potential of hardware acceleration. Graph accelerators using recent technologies are giving promising results.

For example, Gui et al. [3] emphasize that general-purpose CPUs are not ideal for graph operations due to the irregular nature of graphs. In GraphS [47], Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM) is transformed into massively parallel processing units, which leverages the Processing-in-Memory (PIM) paradigm to eliminate off-chip memory accesses. Another example of PIM architecture was proposed by Ahn et al. called Tesseract [48]. In addition to the 3D logic stacking, Tesseract uses hardware prefetchers specialized for graph processing and has a programming interface.

A different approach to graph processing can be seen in Graphicionado [49], a fully pipelined hardware framework for graph analytics. Graphicionado uses an optimized memory subsystem; it utilizes an on-chip scratchpad memory. Moreover, many micro-architectural optimizations are made around GraphMat [50], a vertex-centric software graph framework. GraphReduce [51], on the other hand, uses a combination of both edge and vertex-centric implementations of the Gather-Apply-Scater (GAS) programming model and operates on GPUs. GraphReduce also uses CSR and CSC layout formats to store the graph data. Another proposal was given by Rahman et al. by introducing a hardware framework, GraphPulse [52], that focuses on asynchronous graph processing with an event-driven schedule. GraphPulse uses delta-based accumulative processing, which differs from the edge and vertex-centric approaches. Delta-based accumulative processing propagates a change (delta) in vertex value towards outgoing neighbors of the vertex. Finally, a soft processor-based approach for efficiently processing sparse graphs is proposed by Kapre [53]. In his work, lightweight processors with custom instruction sets are interconnected via a 2D array.

It is clear that many different approaches to graph processing focus on similar challenges, such as the irregular nature of the graphs, inefficient memory access, the size of the graphs, and novel computation models.

3.5 Implementation Choices

Real platform implementation of the graph accelerator prototypes is another factor that affects experimental results. Examining the related work on graph processing, it is evident that the dominant choice is FPGA implementation. For example, GraphStep [54] is one of the pioneers in using a generic FPGA architecture that supports more than one graph algorithm. GraphGen [55] allowed storing graph data in DRAM and partitioning the graph data to fit into much smaller BRAM. GraphSoC is another framework that was implemented on FPGA. GraphSoC stores the entire graph in CSR format in BRAMs. For the graph data that is larger than the total BRAM capacity, multiple FPGA devices are used. Another proposal focusing on efficient scaling is ForeGraph [56], which uses multiple FPGA devices, each with an off-chip DRAM module. All of the FPGA implementations point to two fundamental problems: Insufficient DRAM-FPGA bandwidth and FPGA internal memory.

GPU implementation is another design of choice for graph accelerators. GraphReduce can be given as an example in this category. It is evaluated on an HPC node equipped with GPUs. GraphiDe [57] is also implemented on a platform that uses a single GPU and a hybrid-memory cube (HMC).

Chapter 4

System-on-Chip Design and Architecture

Since the hardware implementation phase of the design restricts many ideas and prototypes used in simulations, a direct import from simulation to a real platform is highly unlikely. Therefore, hardware-dependent modifications to the design are inevitable. The base design's memory subsystem is redesigned due to the requirements of complex on-chip communication protocols and used external hardware components. This chapter describes our software and hardware components alongside integration details.

4.1 Previous Work

The previous study [6] used an open-source soft processor called IBEX [58] as the baseline design. Custom instructions that manage a scratch-pad memory (SPM) are defined for the baseline. SPM is used in the Edge Scratch-Pad (ESP) design, which stores read-only edge data, i.e., columns array in CSR representation. ESP is managed by the ESP controller, a soft controller that executes custom instructions. Below are the custom instructions used in our processor:

- **spmcon**: *"Scratch-pad Memory Configure"* configures ESP Memory by marking the start address of vertex and edge data.
- **memspm**: *"Memory-to-SPM"* moves edge data array from external memory to ESP Memory.
- **spmreg**: *"SPM-to-Register"* loads edge data array from ESP Memory to CPU registers.
- **delspm**: *"Delete SPM"* removes edge data array from ESP Memory.

Figure 4.1 describes the micro-architecture of the previous design. Both IBEX and ESP are interfaced to memory by Request Arbiter. Request Arbiter keeps CPU and ESP requests in separate queues implemented in a FIFO manner and prioritizes one request to another where both CPU and ESP controller issue a request simultaneously. Non-blocking Cache Controller responds to the memory requests, and depending on the situation, the Miss Status Holding Registers (MSHR) controller makes a memory request to the RAM. RAM is implemented as a 32-bit wide array.

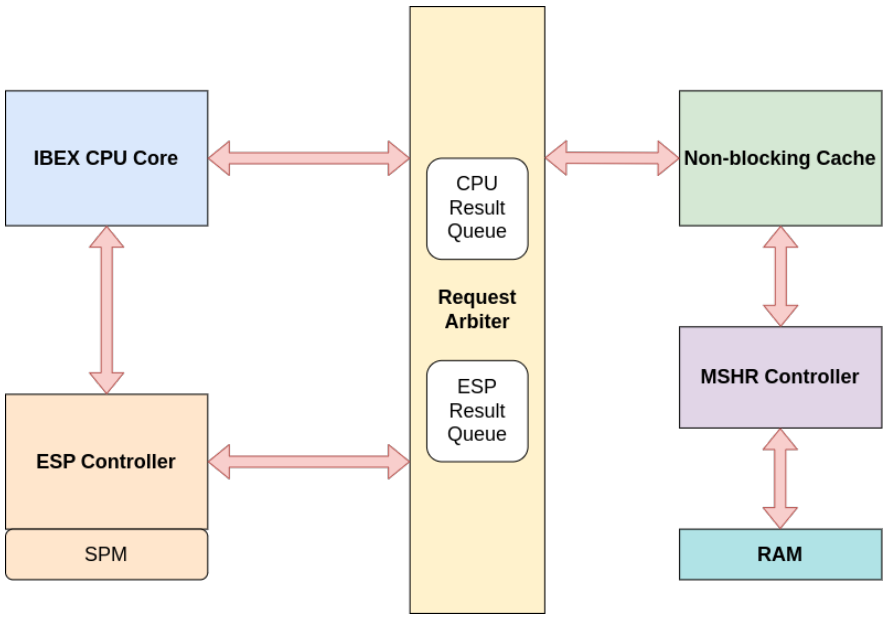


Figure 4.1: Baseline system architecture of the processor.

Non-blocking cache is direct-mapped and has a 128-bit (4 words) cache line. A prefetch buffer is also embedded in the non-blocking cache, storing the results of predictable future memory requests. The non-blocking functionality of the cache comes from MSHR. MSHR, also known as *miss buffer* and is implemented as FIFO, can handle multiple outstanding misses in the cache. Cache misses are held in MSHR FIFO, and memory requests are made by the controller accordingly.

4.2 Overall System Design

Changes and additions to the micro-architectural design have been made to implement the base design on an actual hardware platform. Major changes are made in the memory subsystem and the on-chip communication network. Utilization of the AXI protocol is more efficient when the data widths are large enough. In other words, a single memory request for a large data block is more efficient than multiple requests for a data block of the same size. As a result, additional cache memory and a custom AXI interface are integrated into the base design, allowing fewer memory requests with large data widths and making the processor chip have two caches, L1 and L2. ZCU102 development board is used for the implementation.

Figure 4.2 describes the improved design with external components used in the implementation. Overall system uses both the PL (highlighted in red) and PS (highlighted in blue) parts of the Zynq device. The host computer stores graph data and instructions on the SD card. The PS part of the Zynq is responsible for reading the CSR graph data and program instructions from the SD card and writing them to the external DDR RAM. When the data load is completed, the PS side prompts the host computer. The user turns dedicated dip switches ON on the development board, enabling the graph processor to reach the DDR RAM via AXI peripherals. Program instructions are read via a custom AXI module and stored in the distributed RAM. The graph processor uses AXI modules and an AXI interface through the program execution time. When the program ends, an interrupt is sent to the PS side from the PL side. Statistical data is written

to the DDR RAM. Finally, the PS side verifies the memory dump and prompts the host computer.

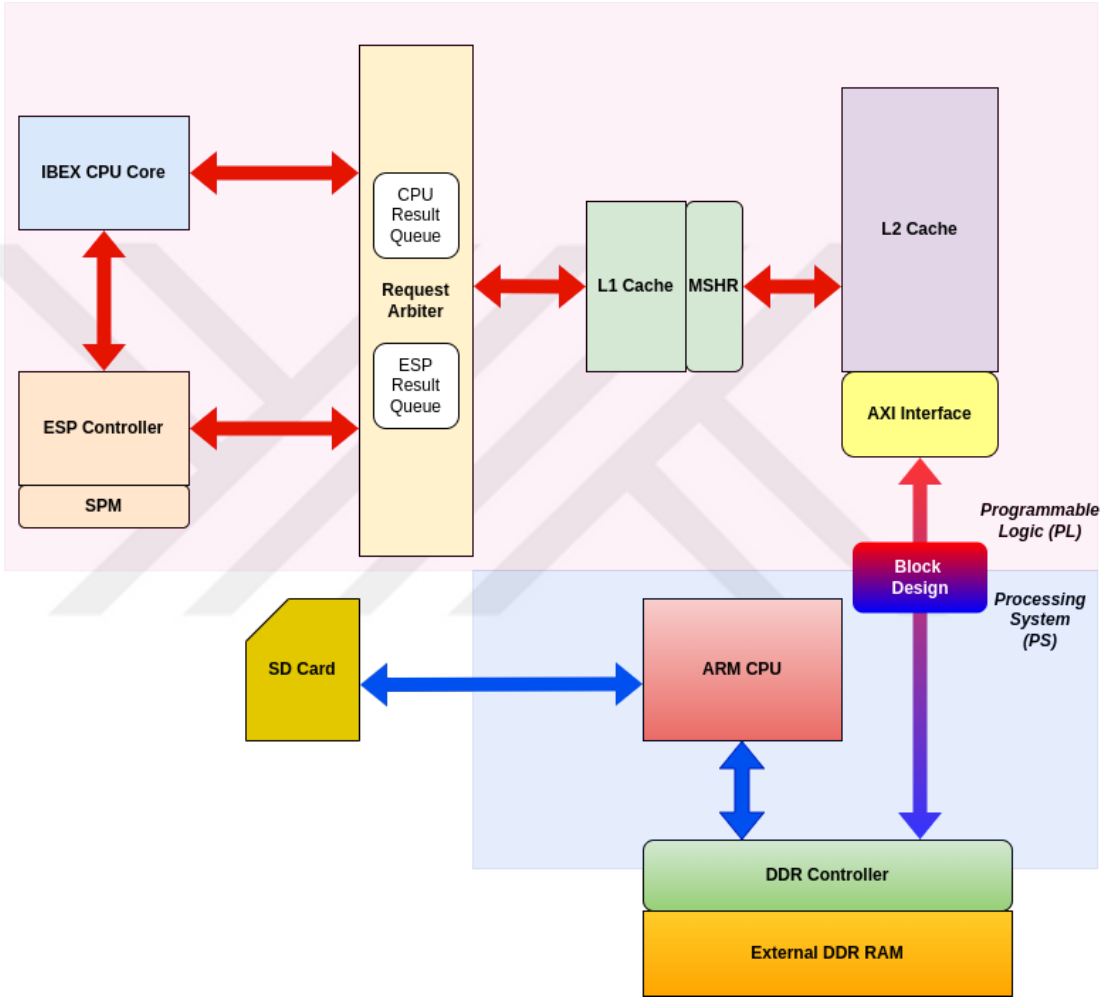


Figure 4.2: Improved system design, with external components used in the implementation.

4.3 File I/O Operations

Graph data, program instructions, and reference memory dump are stored in the SD card in text file format. SD card is placed into the card reader of the development board. The ARM processor reads the SD card on the PS side using a simple bare-metal application. The application writes the graph data and the instructions to the specified addresses of the DDR RAM. The application also uses software-controlled cache flush operation to maintain cache coherency between the PS and PL sides. Figure 4.3 describes the process happening in PS for file operations.

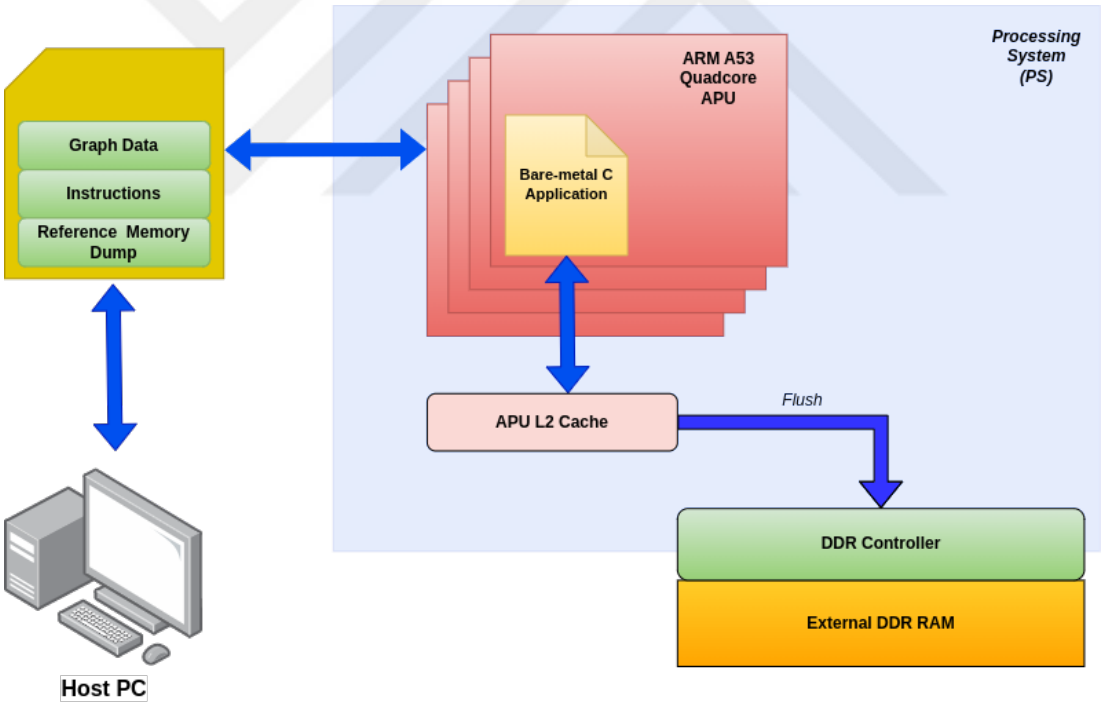


Figure 4.3: File read and write operations on PS.

4.4 Memory Interface

This section describes the internal logic of a custom AXI master module, an AXI interface, and the L2 cache in the design. Since the baseline design is implemented in the PL part of the Zynq device and the DDR controller is placed on the PS side, AXI peripherals should be used to establish communication between the PS and PL. Xilinx offers *Zynq UltraScale+ MPSoC Processing System Intellectual Property* (Zynq IP) to ease the configuration between PS and PL [59]. Zynq IP allows the designer to program features such as AXI peripherals, PS clock and reset generators, and various hardware controllers. As Zynq IP's recommended design flow adds it to the system as an IP block, a block design is added to the baseline.

4.4.1 Custom AXI Module

Memory requests are issued on the PL side, and they need to be transferred to the DDR RAM on the PS side. Since the PL side is the initiator, an AXI master is needed to control the transfers as efficiently as possible. Therefore, a custom AXI master module is designed. The custom module uses the AXI4-Full protocol and operates as a single-beat AXI master. The data width is set to 128-bit to achieve maximum throughput. Multiple instances of the module can be added to the block design, and it can be re-used. It is coded in Verilog HDL by using Xilinx's design template.

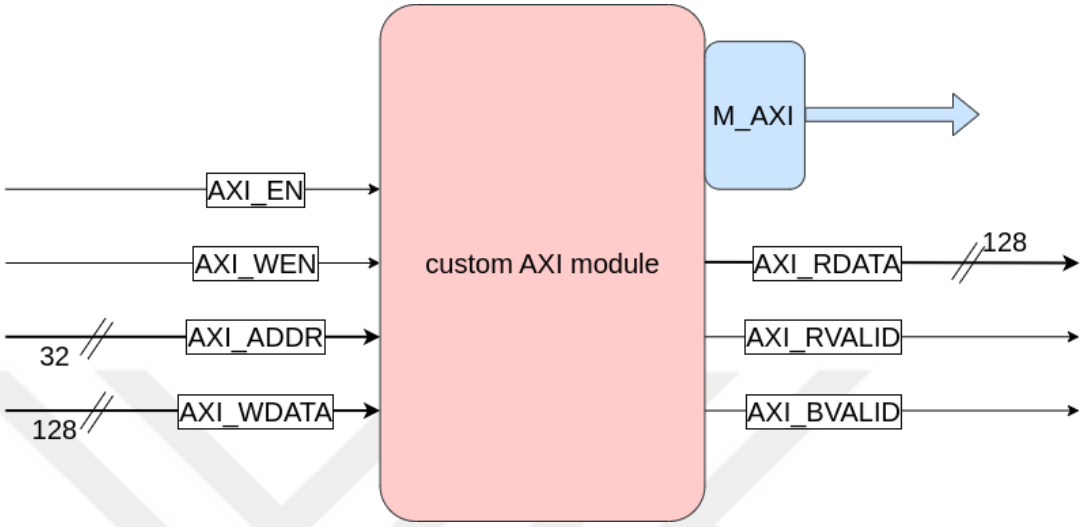


Figure 4.4: Custom AXI module block diagram.

Figure 4.4 shows the block diagram of the custom AXI module. M_AXI sub-module is the collection of standard AXI4 signals present in the template, allowing the custom module to be connected to other Xilinx IPs. The rest of the input and output signals are added to the template, and their functionalities are given below:

- **AXI_ADDR**: 32-bit input originates from the L2 cache.
- **AXI_RDATA**: 128-bit output originates from the module's internal registers.
- **AXI_WDATA**: 128-bit input originates from the L2 cache.
- **AXI_EN**: Single-bit module activation input originates from the L2 cache.
- **AXI_WEN**: Single-bit write-enable input originates from the L2 cache.
- **AXI_RVALID**: Single-bit read-valid output originates from the module's internal registers.
- **AXI_BVALID**: Single-bit write-valid output originates from the module's internal registers.

The custom module always keeps RREADY, and BREADY signals high to lower latency. This means the module is always ready to accept read data or acknowledge write operations.

The write operation starts with a single clock cycle pulse input to AXI_EN and AXI_WEN, simultaneously loading the address and data value to the AXI_ADDR and AXI_WDATA input, respectively. AWVALID and WVALID are set after the pulse and wait for the AXI slave to assert AWREADY and WREADY. As the slave asserts AWREADY and WREADY, registration of the write address and data values to the slave is completed. After some delay, the slave returns BVALID, completing the transfer. In Figure 4.5, a digital waveform graph of the expected behavior in the write operation of the custom AXI module is given.

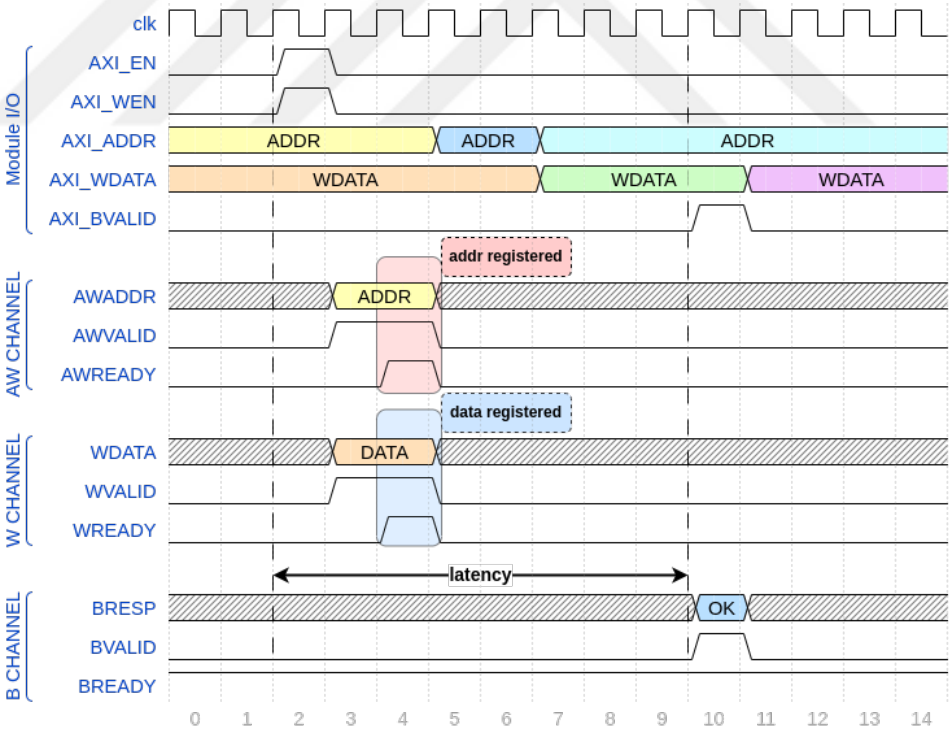


Figure 4.5: Standard write operation of the custom AXI module.

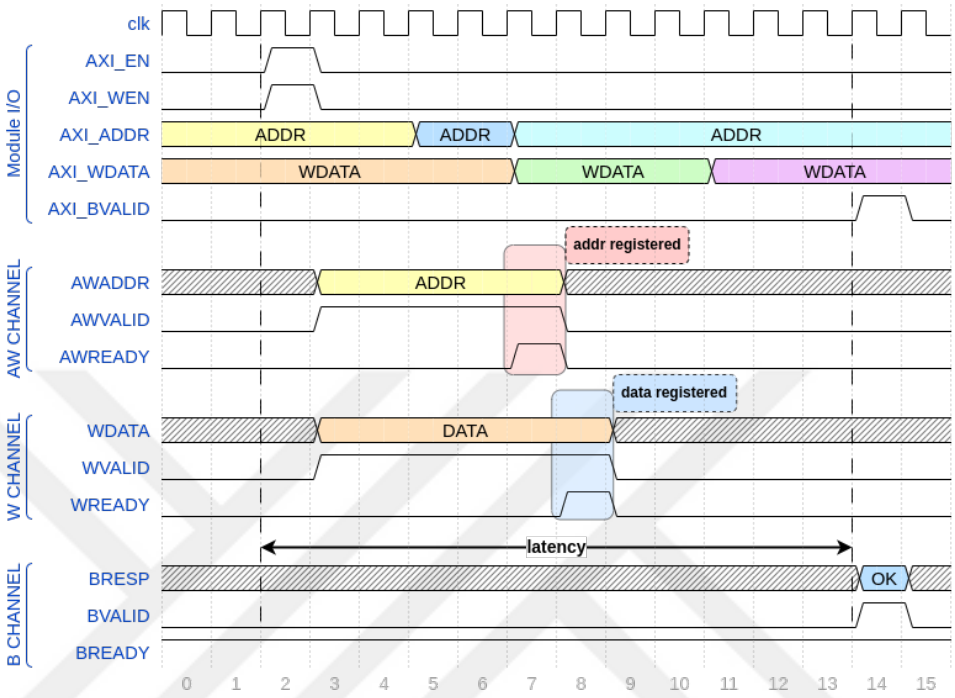


Figure 4.6: Write operation waveforms of the AXI module in the case of delayed xREADY signals by the AXI slave.

In Figure 4.6, write operation in the case of delayed READY signals are shown. Custom AXI module asserts VALID signals until READY signals are returned from the slave.

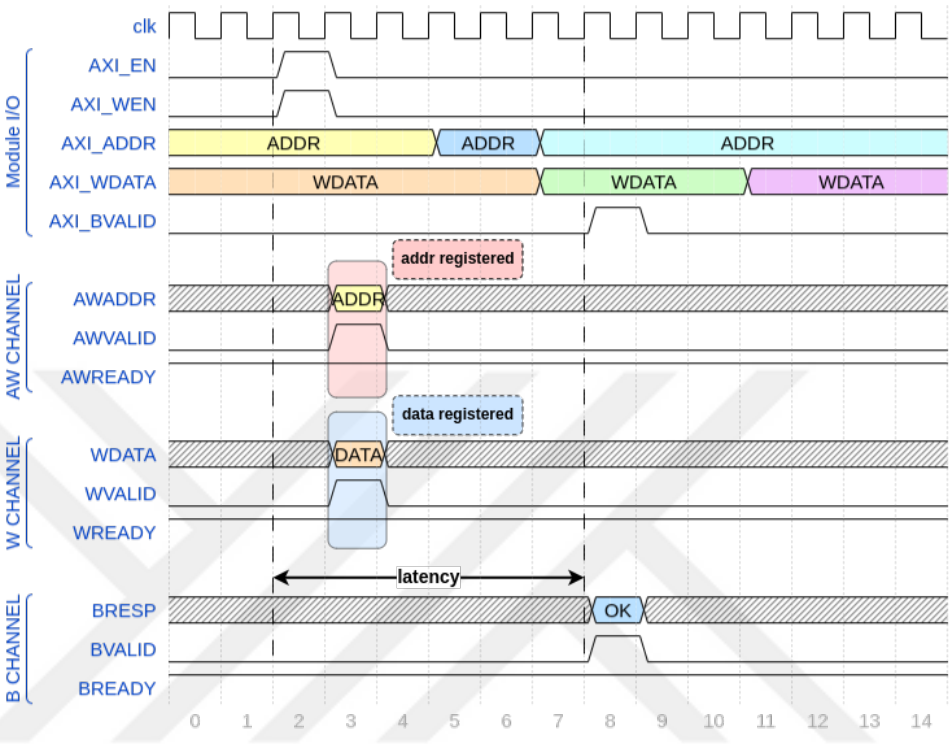


Figure 4.7: Write operation waveforms of the AXI module where the xREADY signals are always held high by the AXI slave.

Figure 4.7 shows the scenario when READY signals are always held high by the slave.

Read operation starts with a single clock cycle pulse input to the AXI_EN and loading the address value to the AXI_ADDR input. ARVALID sets after the pulse input and waits for the AXI slave to assert ARREADY. As the slave asserts ARREADY, registration of the read address value to the slave is completed. After some delay, the slave returns RVALID with RDATA, completing the transfer. Figure 4.8 shows the digital waveform graph of a read operation.

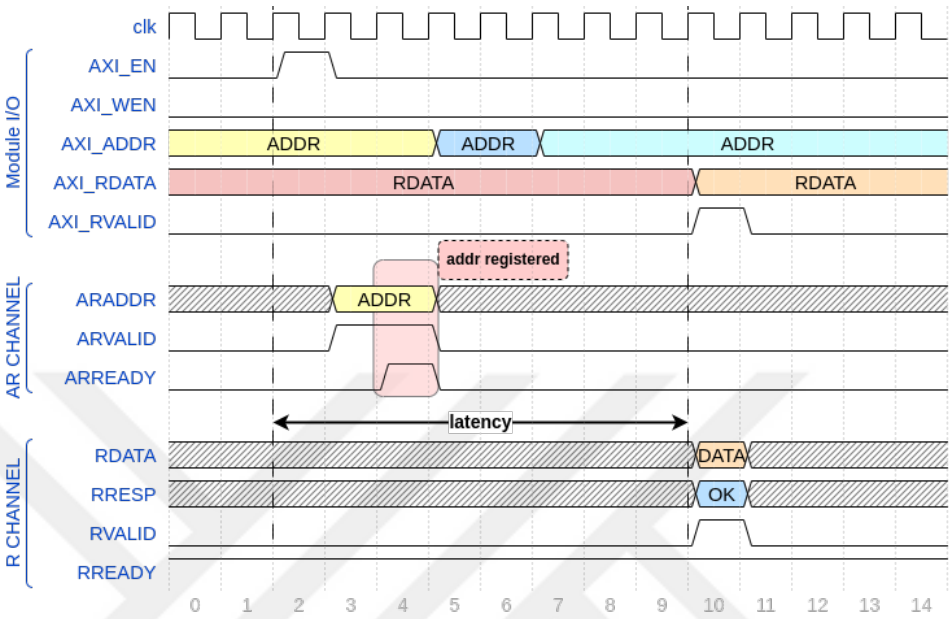


Figure 4.8: Read operation waveforms of the AXI module.

Transfer latency depends on the combination of logic, architecture, and DDR RAM delays. Logic delays are the result of the internal logic design of the module. Architecture delays caused by the SoC hardware configurations. For example, cache-coherent AXI ports on the Zynq IP have higher transfer latency than high-performance ports. DDR RAM also affects the transfer delay due to its internal dynamics.

4.4.2 L2 Cache

L2 cache is the design's last micro-architectural module before the AXI interface, and it receives data requests from a higher memory hierarchy and responds by a hit or a miss. L2 cache is designed as direct-mapped in the baseline implementation. 2-way and 4-way associative cache designs emerged as potential expansions to the baseline, aiming to broaden the scope of the experiments. The tag and data pairs of the cache are held in tables [60] and stored in the BRAMs of the Zynq device. The cache line width is chosen as 512 bits. The write-hit policy of the cache is Write-Back, which means in case of a write-hit, data and the address

is written to the cache only, and the cache block is marked as dirty. If the dirty cache block is being evicted, data is written to the lower level of the memory hierarchy. The write-miss policy uses two approaches: write-no-allocate (Write-Around) and write-allocate [61]. There are two main reasons behind designing two different cache controllers regarding the write-miss policy. In simulations, the occurrence of write-miss was relatively low in the base design, so it was decided that handling a write-miss without disturbing the cache is a viable solution. Furthermore, using a single AXI module (128-bit) is more efficient than four concatenated (512-bit) AXI modules since we do not utilize burst transactions. In the Write-Around policy, the request bypasses the L2 cache and is written directly to the memory hierarchy's lower level on a write-miss. As a result, cache block eviction only happens on a read-miss. In the write-allocate policy, request data and the address are written to the allocated cache block and only written back to the lower levels of the memory hierarchy in case of a conflict miss.
Figure 4.9 illustrates the direct-mapped L2 cache design. Cache tables are stored in BRAMs and shown in cache diagrams exterior since BRAMs are added to design via block design canvas. AXI interface is a sub-module that connects the cache to the custom AXI module.

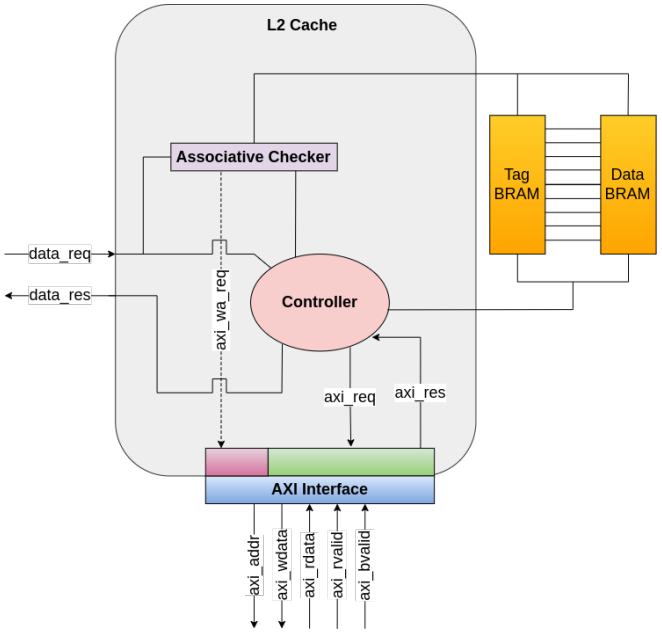


Figure 4.9: Direct-mapped cache design as the baseline.

4.4.2.1 Cache Controller

The cache controller is the heart of the L2 cache and is implemented as an extended finite state machine (FSM). Figure 4.10 shows the state diagram of the cache controller FSM. The optional state "Write-Around" is represented in dashed lines. The cache controller state diagram is identical for direct-mapped and associative cache configurations. *Idle* is the state where the controller FSM waits for data requests. A request is the front-most element of the MSHR Buffer at that time. The *Idle* state transitions to the *Compare-Tag* state if a valid request is received.

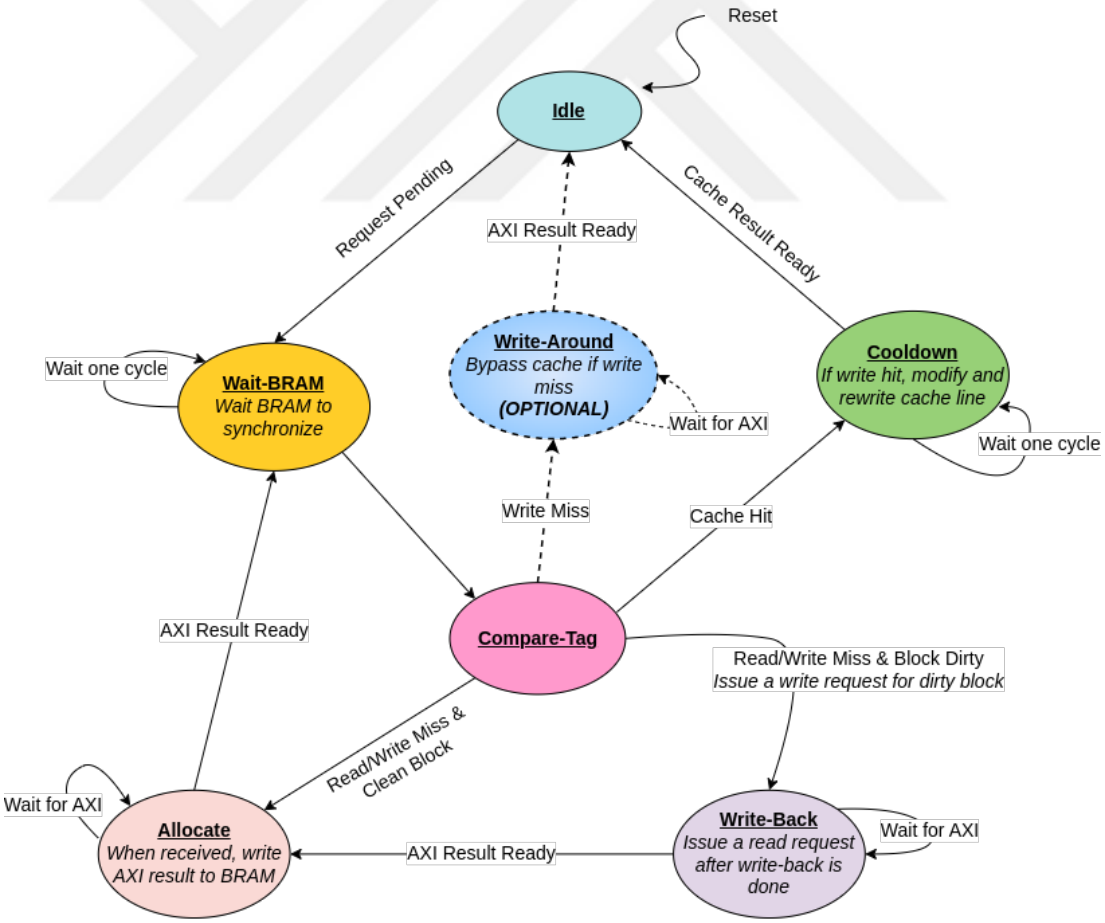


Figure 4.10: Cache controller FSM. The Write-Around state is optional, and its effects on the performance are evaluated.

The request address tag and tags stored in BRAM are compared in the

Compare-Tag state. The cache hit signal is generated if the tags are equal and the data is valid. In the write-hit scenario, the dirty bit is set. Also, 128-bit write data is placed in the correct position of the 512-bit cache line according to the word offset of the request address. Similarly, in read-hit, the word in the correct position of the cache line is returned as a result. In any hit scenario, the *Compare-Tag* state transitions to the *Cooldown* state. *Cooldown* state is a single clock cycle delay which is needed since BRAMs are synchronous, i.e., write or read operations have single clock cycle latency [62]. Also, since used BRAMs are single-port, two clock cycles are needed for written data to appear at the read port. A write-hit case modifies the BRAM's cache block after fetching it, and a read-hit case requires directly fetching the cache block from the BRAM. The *Cooldown* state sets the ready signal indicating the cache controller result is valid and transitions to the *Idle* state.

In the write-no-allocate configuration, the *Write-Around* state is added to the FSM. A write-miss in the *Compare-Tag* state causes an AXI request of one word (128-bit) write request to be issued in the AXI interface and transition to the *Write-Around* state. The *Write-Around* state stalls the cache until the ready signal is received from the AXI interface and transitions to the *Idle* state. If the cache controller configuration is write-allocate, a write-miss on a clean block (compulsory miss) causes the cache block to be allocated and marked as dirty for the write request. Furthermore, an AXI request is issued in the AXI interface to fetch the memory block, and the state transitions to the *Allocate*. The cache controller is stalled in the *Allocate* state until the ready signal is received. As the ready signal is received, the fetched memory block is written to the BRAMs, and the FSM transitions to *Compare-Tag*. The fetched cache block is modified in the *Compare-Tag* state, and the write request is written to the correct position according to the write offset. The cache controller shows similar behavior in read-miss as with write-miss. The only difference is that a compulsory read-miss does not modify the fetched cache block in *Compare-Tag*.

In case of a read-miss or a write-miss in the write-allocate configuration, if the cache block is dirty, an AXI write request is issued at the AXI interface, and the write data is sampled from the dirty cache line. The state then transitions to the

Write-Back state. In the *Write-Back* state, the cache is stalled until the ready signal is received from the AXI interface. Upon receiving the ready signal, an AXI read request for the clean cache block is made at the AXI interface, and the state transitions to the *Allocate* state. When the ready signal is received from the AXI interface, read data is written to the BRAM, and the state transitions to the *Wait-BRAM*. Finally, tags are compared in the *Compare-Tag* state, and the cache block is modified if it is a write-miss. Finally, after completion, the FSM returns to the *Idle* state.

4.4.2.2 Associative Cache Modifications

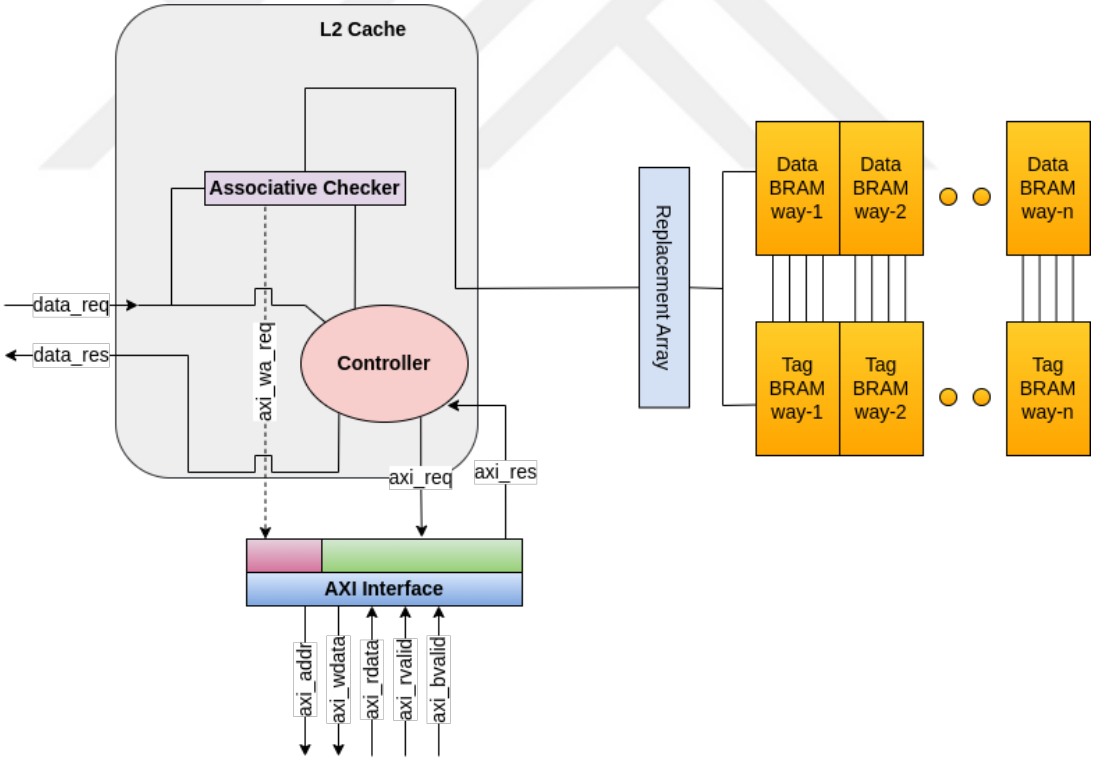


Figure 4.11: Set-associative cache design as an improvement to baseline.

Figure 4.11 shows the set-associative variant of the baseline L2 cache design. The states in the FSM of the associative caches are the same as in the direct-mapped baseline. Major changes are made in the micro-architectural design. Multiple BRAM pairs are generated to represent a "way" in the associative cache. In

case of a cache hit, the combinational logic between the BRAM banks determines the *way* of the hit. Also, set-associative caches require a replacement policy in case of a cache block eviction. Although the Least Recently Used (LRU) replacement policy is a better option performance-wise, we used a FIFO replacement policy due to its logic area efficiency and implementation simplicity [63]. For the replacement array, distributed RAM is used instead of a BRAM since the replacement array is not wide enough to utilize BRAM efficiently since BRAMs are finite in size [62]. For example, in the 2-way associative configuration, a single-bit wide array with a depth equal to the BRAMs in the design is used. In every replacement in the set of given cache index, the replacement array element in the same index is incremented. In an eviction, the value of the replacement array represents which data in the set will be evicted.

4.4.3 AXI Interface

AXI interface is the sub-module that makes memory requests using custom AXI modules. For 512-bit requests, it uses four concatenated custom modules, and for 128-bit requests, it uses a single write-only custom module. The write-only custom module is reserved for Write-Around requests. Concatenated custom modules are used for other cache requests. Moreover, these modules are responsible for reading the program instructions from DDR RAM at the system start and writing the statistical data to the DDR RAM at the program end. We could not utilize the burst feature of AXI4 due to the 4KB Boundary specification [18]. The specification implies the last twelve bits of the start address and the last twelve bits of the end address of a burst stay the same. As a result, starting address of a burst should be calibrated according to the specification, or a variable length burst should be used. As a simple workaround, we used multiple instances of a single-beat AXI module to achieve a pseudo-bursting module. Although the pseudo-bursting module has higher transaction latency than regular bursting modules, it is simpler to design and debug.

4.4.4 Block Design

The block design is a special element that is a canvas for Xilinx IPs and custom RTL modules. Although utilizing block design is not mandatory, it is the most efficient way to use the Zynq IP. Design blocks used in the block design are Zynq IP, Block Memory Generator IP, AXI Interconnect IP, and custom AXI modules. Figure 4.12 illustrates an overview of the block design canvas together with the AXI interface and L2 cache. Concatenated custom AXI modules and their I/O signals are shown in different colors.

Zynq IP provides the High-performance AXI slave port, which is used by the custom AXI master module. Although this AXI port is not coherent with the APU caches, it has the lowest latency profile among other slave ports. The HALT interrupt generated at the end of the graph program is connected to the PL-to-PS interrupt port of the IP. Zynq IP also generates the system clock with the desired frequency.

The L2 cache uses Block Memory Generators. BRAM Memory depth, data/address width, and operation mode can be adjusted in the generator block.

AXI Interconnect connects multiple AXI masters to a single AXI slave or vice-versa. The arbitration priority of the AXI Interconnect can be adjusted. For example, if multiple AXI masters send requests to a single AXI slave simultaneously, AXI Interconnect prioritizes one request while buffering others in its internal FIFOs.

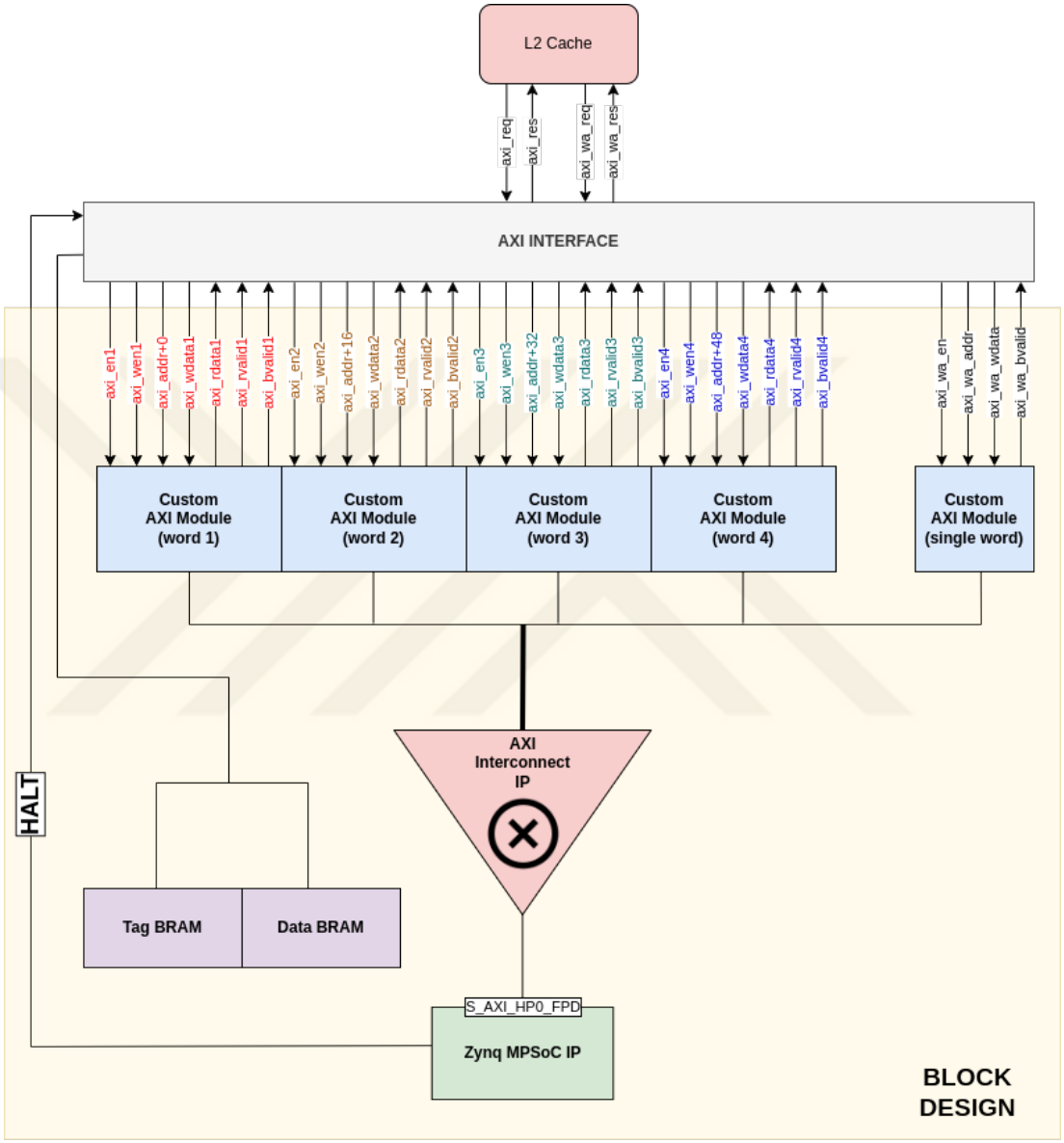


Figure 4.12: Overview of AXI interface and the block design.

4.5 Integration

L2 cache and AXI interface are directly added to the base design as RTL design files. On the other hand, the block design needs an RTL wrapper to be used in SystemVerilog modules. *Vivado* generates a Verilog wrapper for the block design.

The wrapper contains the inputs and outputs of the block design. The wrapper connects the block design to the rest of the RTL design. After completing the Synthesis and Implementation steps, a platform file is generated and exported to *Vitis*[1], the software development platform of Xilinx [64]. The platform file is called Xilinx Support Archive (XSA) and contains information about the real hardware platform the design will run on. Software design is built upon the platform exported from *Vivado*. Furthermore, Zynq is programmed using USB. Integration steps are summarized in Figure 4.13.

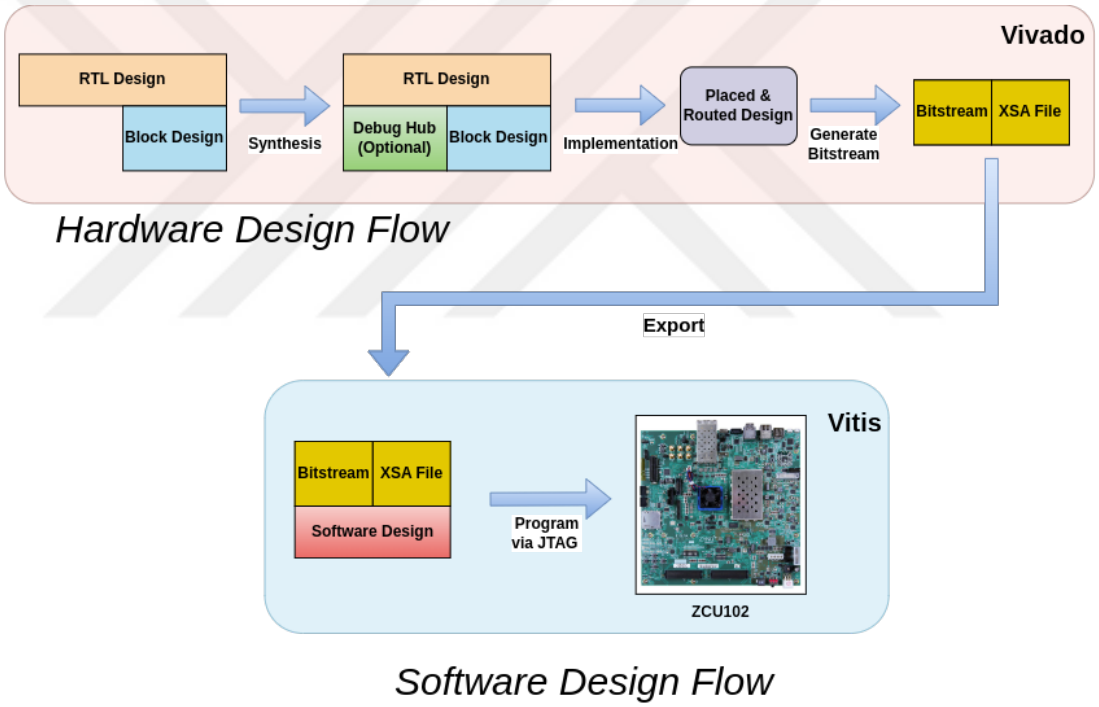


Figure 4.13: Integration steps of the system.

[1]https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

Chapter 5

Evaluation

In order to evaluate the effect of ESP on the design, the execution times of two instruction files (RISC-V binary) on benchmarks are compared; one utilizes custom instructions while the other does not. Instruction files are generated by compiling graph benchmarks on the PC with different compiler directives. In the end, both instruction files were loaded onto the SD card, and experiments were done for each case. Experiments evaluating the micro-architecture of the design are done with the instruction file utilizing custom instructions.

5.1 Benchmarks

5.1.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a widely used traversal algorithm that prioritizes exploring the nodes at the same depth before moving on to the next depth level [65]. For graphs, it is mainly used for finding edge distances and parent vertices on a path of the given source vertex. BFS is used as a common benchmark in previous works [6, 5]. The textbook implementation of BFS is used in experiments [66].

5.1.2 Depth-First Search (DFS)

Depth-First Search (DFS) is another well-known traversal algorithm that prioritizes exploring as far as possible in the depth plane before needing to backtrack [66]. DFS can be iterative or recursive; we used the iterative version in experiments.

5.1.3 PageRank

PageRank algorithm finds the importance of web pages. Importance rating is defined recursively as a web page becomes important if important web pages link to it. It is one of the foundation algorithms used in Google Web Search and used in many other applications. An iterative implementation of the algorithm is used in the experiments [67].

5.1.4 Single Source Shortest Path (SSSP)

Single Source Shortest Path (SSSP) algorithms are iterative algorithms that find the lowest cost path between two vertices on a graph. Cost is calculated by using edge weights. The Belman-Ford algorithm is a versatile example of SSSP algorithms that can run on graphs with negative edge weights [68].

5.2 Graph Data

In experiments, ten directed graphs of various sizes are used. Table 5.1 lists all graph data used in the evaluation. All graphs except Graph13 are real-life graphs taken from SNAP Dataset [69]. Graph13 is a synthetic graph generated using Graph500 [70] Kronecker Generator. The graphs are converted to CSR format, with randomly generated weights between 1 to 100.

Graph Name	Vertices	Edges	Abbreviation	Short Description
amazon0312	400,727	3,200,440	**AMA1**	Amazon product co-purchasing network
amazon0302	262,111	1,234,877	**AMA2**	Amazon product co-purchasing network
web-NotreDame	325,729	1,497,134	**WND**	Web graph of Notre Dame
as-Skitter	1,696,415	11,095,298	**AS**	Internet topology graph, from traceroutes
wiki-Talk	2,394,385	5,021,410	**WT**	Wikipedia talk (communication) network
soc-Pokec	1,632,803	30,622,564	**SP**	Pokec online social network
wiki-topcats	1,791,489	28,511,807	**TC**	Wikipedia hyperlinks
email-Eu-core	1,005	25,571	**EEU**	E-mail network
ego-Facebook	4,039	88,234	**EF**	Social circles from Facebook
Graph13	8,192	263,408	**G13**	Synthetic (Kronecker)

Table 5.1: Graphs used in experiments and their properties.

5.3 Verification

The design is verified by comparing the memory dumps generated at the end of program execution in PC and FPGA. In previous studies, at the end of the RTL simulation, memory dump and dirty lines in the cache are output in separate files [6]. Dirty cache lines are combined with the memory dump to generate the final one. The comparison is made on the PC via a Python script. This method is not useful for large graphs due to large file outputs. File transfer from ZCU102 DDR to PC is possible with the JTAG connection but at very low transfer rates. A change has been made to ease the computation. After the program termination, dirty lines in both L1 and L2 caches are written back to DDR using AXI inside the FPGA. This way, the memory dump to be compared becomes final. Moreover, the comparison step is moved to the ZCU102 entirely. A simple bare-metal C program is used on the PS side to compare memory dump from the PC and DDR. A summary of the verification steps is given in Figure 5.1.

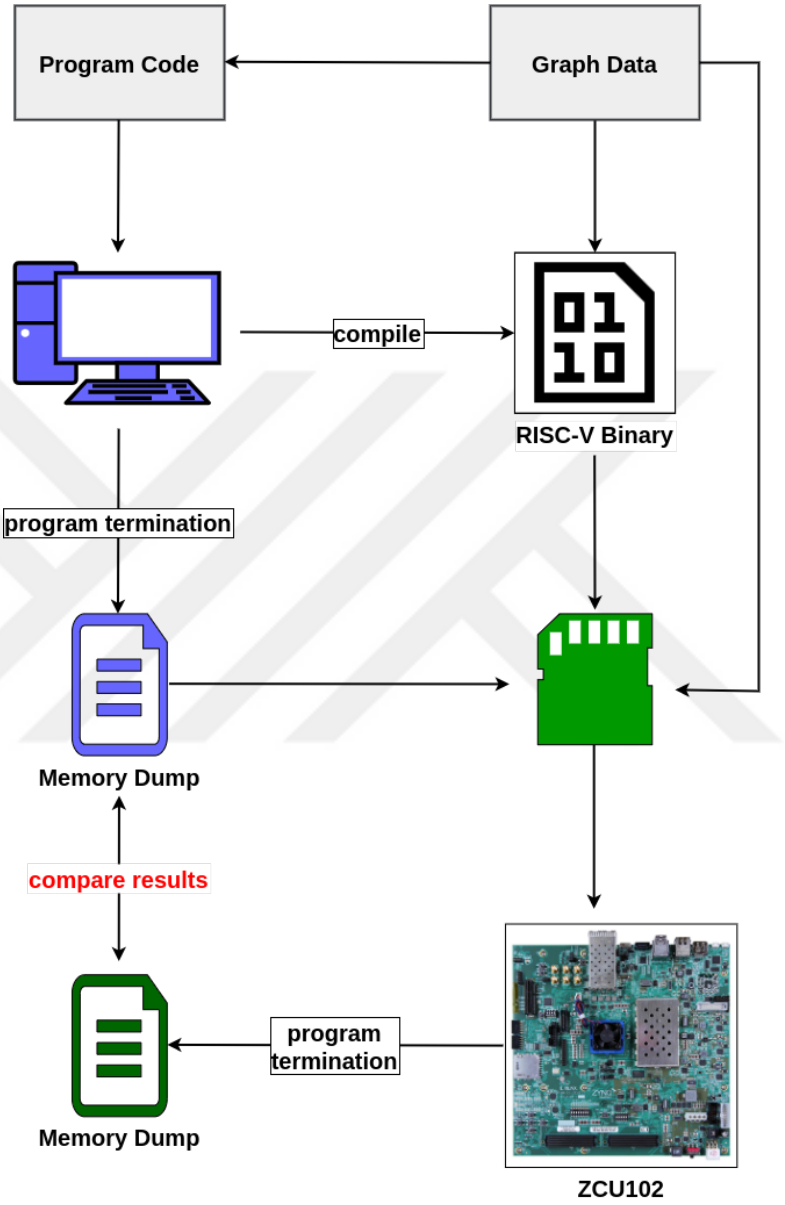


Figure 5.1: System verification summary.

5.4 Default Parameters and Setup

This section evaluates important parameters of the design and their effects on the overall design performance. Also, the platform setup used in experiments is described.

- **L1 Cache Size** is chosen as 1KB. L1 Cache is mapped as distributed RAM in ZCU102, meaning it has 0 clock cycle read and write latency; however, it uses LUTs, which are scarce.
- **L1 Cache Line Size** is set to 128 bits, representing 4 words.
- **L2 Cache Associativity:** Direct-Mapped (1-way Set Associative) cache configuration is used in the experiments as the default L2 Cache type since its synthesis and implementation phases are quicker than other configurations.
- **L2 Cache Size** is set as large as possible, within the limits of BRAM size. The default value is 2MB, utilizing over half of the total BRAM sources.
- **L2 Cache Line Size** is set to 512 bits, representing 4 words of 128 bits. Note that a word in L2 Cache equals the data width of a custom AXI module.
- **Clock Frequency** is set to 75 MHz, a sweet spot between speed and integration phase duration. Higher clock frequencies make routing and optimization harder for Vivado [71].
- **MSHR Buffer Size** is an important parameter since it holds outstanding misses between L1 and L2 cache. At first, experiments were done with 8 rows. However, overflows and data corruption happened in the FIFO buffer, and the default size was changed to 16 rows.
- **Prefetch Buffer Size:** is a major parameter that impacts the performance of the baseline design. The default prefetch buffer size is set to 4 blocks of data.
- **SPM Block Size** is set to 16 words and remains the same as in the baseline design.
- **SPM Capacity** is set to 4 blocks and remains the same as in the baseline design.

Table 5.2 summarizes the default parameter values used in the evaluation.

Parameter	Value
L1 Cache Size	1 KB
L1 Cache Line Size	128-bit
L2 Cache Associativity	Direct-Mapped
L2 Cache Size	2 MB
L2 Cache Line Size	512-bit
Clock Frequency	75 MHz
MSHR Buffer Size	16 rows
Prefetch Buffer Size	4 blocks
SPM Block Size	16 words
SPM Capacity	4 blocks

Table 5.2: Default parameter values.

The system setup used in experiments is implemented on a host PC to program and debug ZCU102 via USB. Two USB cables are connected to the host PC, one for programming/debugging and one for serial communication via the terminal. ZCU102 can be programmed from either Vivado or Vitis.

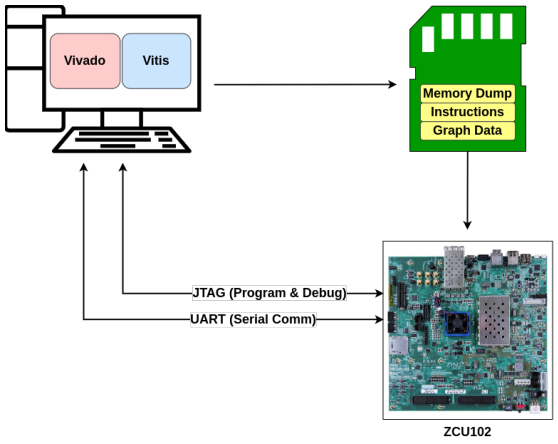


Figure 5.2: System setup summary.

5.5 Performance Evaluation

Experiments were done with the benchmarks given in Section 5.1 and default design parameters. The main goal of the graph processor is to improve execution performance in graph applications. Therefore, speed-up results on the benchmarks with the graph data set are presented.

Speed-up results are obtained by running the graph processor benchmarks with two configurations, one with ESP present and one without. Speed-up value is calculated by the ratio between the two.

The average speed-up values are calculated over all graphs used in experiments. Traversal-centric benchmarks (BFS, DFS, SSSP) generally have better performance improvements than PageRank, verifying previous studies. However, overall speed-up results are limited due to high miss penalties in real platforms.

- **BFS** shows average performance improvement around 16%. BFS is a stable benchmark, also used in the sensitivity analysis section. The lowest speed-up is achieved in WT, which has the highest number of vertex among the graphs. Figure 5.3 shows the normalized program execution times of BFS running on graph data.

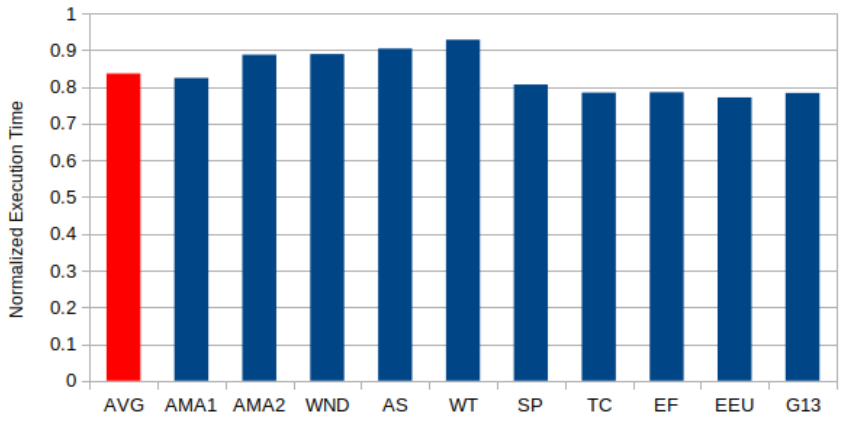


Figure 5.3: Normalized execution times of the graphs with BFS benchmark.

- **DFS** shows an average performance improvement of 18%. In contrast with the BFS, the best speed-up is achieved in the WT, with 30%. Figure 5.4 shows the normalized program execution times of DFS on graph data.

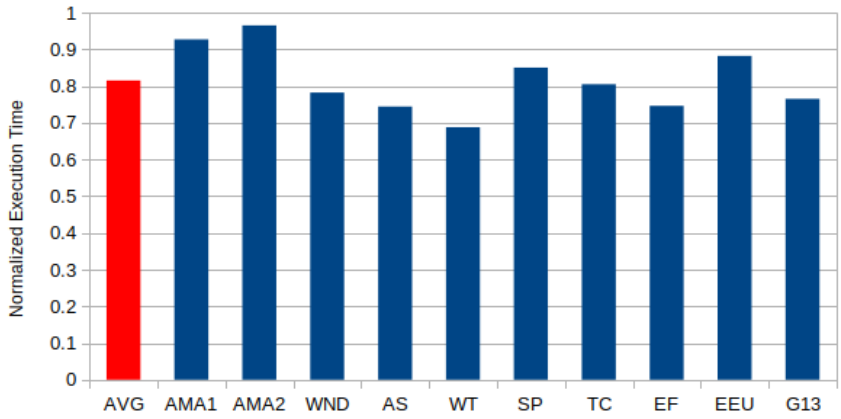


Figure 5.4: Normalized execution times of the graphs with DFS benchmark.

- **PageRank** shows average performance improvement around 14%, which is the worst among other benchmarks. PageRank also has the longest execution times. In graph SP, the real execution time with ESP exceeded 70 minutes; without ESP, it is around 85 minutes. Figure 5.5 shows the normalized program execution times of PageRank on graph data.

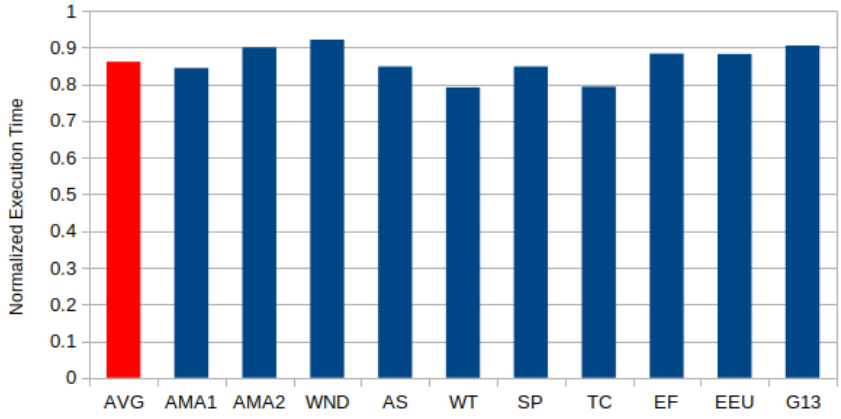


Figure 5.5: Normalized execution times of the graphs with PageRank benchmark.

- **SSSP** shows the highest average performance improvement value among the other benchmarks, around 25%. In graphs such as SP and TC, which have

the most edges, performance improvement peaked, exceeding 35%. Figure 5.6 shows the normalized program execution times of SSSP on graph data. Note that WND and G13 are excluded from the results due to inconsistent results.

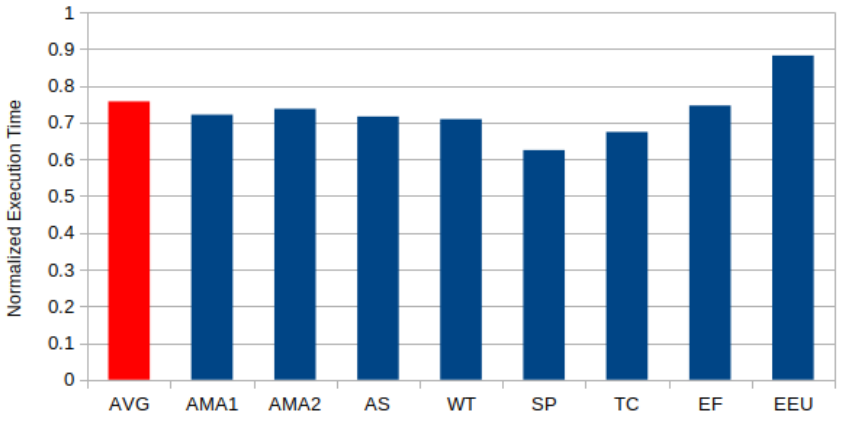


Figure 5.6: Normalized execution times of the graphs with SSSP benchmark.

5.6 Power Analysis and Resource Utilization

5.6.1 Power Analysis

The BRAM utilization and the clock frequency are the most important factors regarding energy consumption. According to Vivado's power reports, PS consumes the most energy, which we can disregard since it is independent of the rest of the design. Overall, the design's power consumption is $\lessapprox$ 4 Watts for 512 KB L2 cache, and $\lessapprox$ 20% of LUTs are utilized in Zynq PL.

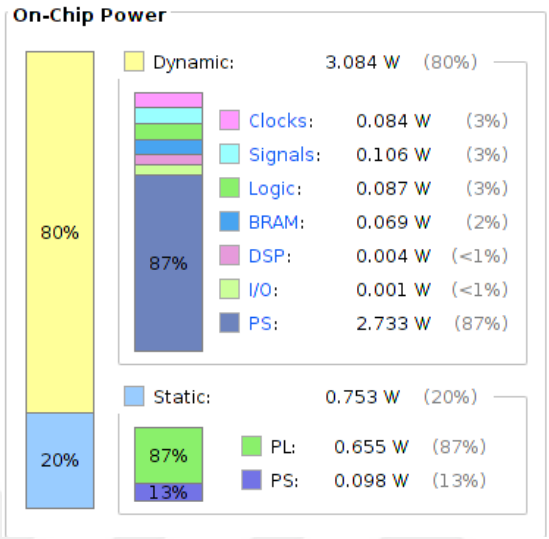


Figure 5.7: Power summary of the design with 512 KB direct-mapped L2 cache at 75 MHz clock frequency.

Figure 5.7 illustrates the power consumption of the Zynq device's hardware components. Static Power is the minimum power required to be provided by the power supply while the design operates idle, and it is a function of junction temperature. Dynamic Power, on the other hand, is the required amount of power to be supplied to the device while the application is running, and its value is based on the average switching activity of the device internals [71]. Total power is the sum of Static and Dynamic Power. As Figure 5.7 suggests PS uses a large portion of the Dynamic Power since AXI peripherals and Zynq IP is located on the PS side and highly active during program execution.

Module	Power Consumption (W)
Ibex Core	0.03
SPM Controller	0.016
Request Arbiter	0.019
L1 Cache	0.097
MSHR Buffer	0.011
L2 Cache	0.024

Table 5.3: Power consumption of the modules in the design with 512 KB direct-mapped L2 cache.

The power consumption of major logic design modules is also investigated for the base design. As it can be seen from Table 5.3, in the case of a design with a direct-mapped L2 cache, the L1 cache is the most power-consuming module in the design hierarchy since it uses logic to implement cache arrays. The power consumption of the L2 cache is fairly low as it leverages BRAMs.

5.6.2 Resource Utilization

The graph processor micro-architecture is relatively lightweight. For example, in the design with direct-mapped L2 cache, the modified Ibex core used 1%, ESP used 1.35%, the request arbiter used 1.22%, the L1 cache used 5.725%, MSHR used 0.63%, and L2 cache used 2.94% of Zynq PL's total LUT resources. The descriptions of the resource elements are given in Section 2.2.

	Utilization %		
Resource	Direct-mapped	2-way	4-way
LUT	17.24	21.10	21.98
LUTRAM	0.31	0.31	0.31
FF	10.34	11.14	11.15
BRAM	15.13	15.19	15.19
DSP	0.40	0.40	0.40
IO	1.22	1.22	1.22
BUFG	0.50	1.24	1.24

Table 5.4: Resource utilization table for the design with 512 KB L2 cache.

Table 5.4 summarizes the change in hardware resources for different L2 cache types. The sharp increase in the utilization of resources like LUTs, FFs, and Global Clock Buffer (BUFG) is observed as cache associativity increases. BUFG is a global buffer used to lower the clock skew between registers that are physically located large distances apart. An increase in BUFG usage indicates the logic added to the associative design forcing synthesis to utilize more clock buffers for efficient clock routing in the PL fabric.

5.7 Comparison with Related Work

Our work represents a general-purpose, compact architecture to ease graph computations. In the implementation step, some of the related work in the literature took the approach of storing the graph data in the BRAMs entirely [54, 72, 73] and often used multiple FPGAs to scale the computational resources [53, 56, 74]. Our design uses BRAMs as a middle level in the memory hierarchy and implements a multi-level cache hierarchy. Unfortunately, a direct performance comparison between our design and state-of-the-art proposals in the literature cannot be made since most of the related work presented is based on graph processing on FPGA accelerators rather than an ASIP approach. Also, we operate at rather lower clock

frequencies. Nevertheless, the graph processor shows promising results in terms of power efficiency. We used some of the comparison metrics presented in the study of Hu et al. [75]. Throughput, measured by millions of traversed edges per second (MTEPS), and energy efficiency, throughput per Watt, are two metrics that are used to evaluate our work. In the BFS benchmark, since all edges of the given graph are visited, MTEPS is calculated by dividing the number of edges of graph data by the program run time. On the other hand, energy efficiency is simply the ratio of MTEPS to power consumption. In other words, it is the measure of obtained processing power for unit energy consumed by the system. Throughput values in Table 5.5 are the average of throughput observed on each graph data used in evaluations with the BFS benchmark.

	Throughput (MTEPS)	**Power Consumption (W)**	**Energy Efficiency (MTEPS/W)**
Graph Processor	1.55	4.2	0.37
PC	5.1	40	0.13

Table 5.5: Overall comparison of the graph processor with the host PC, using the averaged results of BFS benchmark ran on the dataset.

Table 5.5 compares the graph processor with the host PC equipped with the Intel Core i7-9750H [76] processor operating at 2.6 GHz. The power consumption of the CPU on the host PC is queried using s-tui [77]. The graph processor's operating frequency is set to 100 MHz. As can be seen from Table 5.5, although the graph processor achieves lower throughput on average, it consumes much less power and is 2.8× more energy efficient when compared to a GPP.

	Power Consumption (W)	**Energy Efficiency (MTEPS/W)**
PC	40	0.13
Graph Processor	4.1	0.37
GraphBlast[78]	43	81
GraphLily[75]	49	86.2

Table 5.6: Comparison of graph processor, GPP, and accelerator platforms.

Although Table 5.6 gives a general idea of performance and efficiency comparison of different graph processing platforms. The metrics are calculated for the

BFS benchmark only. GraphBlast [78] is a high-performance linear algebra-based graph framework running on GTX 1080 Ti GPU, and GraphLily [75] is a graph linear algebra overlay implemented on FPGA. In terms of pure processing power, accelerators are a better option for graph processing applications.

5.8 Sensitivity Analysis

In this section, individual analysis of some architectural parameters is presented. The BFS benchmark is used in the experiments due to its shorter execution time and stability. Also, smaller graphs EF, EEU, and G13 are discarded since even for the smallest L2 cache; they fully benefit the cache, and the performance increase is saturated. Note that nothing much in the design can be changed after the integration phase. RTL modifications, such as changing the size of a logic array or the value of a logic element, are not possible. As a result, many syntheses and implementations have been done to evaluate design parameters.

5.8.1 Performance Sensitivity Analysis

5.8.1.1 Effect of L2 Cache Size on Performance

We chose the baseline cache size of 64 KB and evaluated performance increase individually in associativity types with the change in cache size. Although cache sizes are the same in all associativity types, BRAM depth differs since a set has a BRAM bank for each way. The normalized performance increase in each graph data to increased cache size is averaged and plotted as a line graph in Figure 5.8.

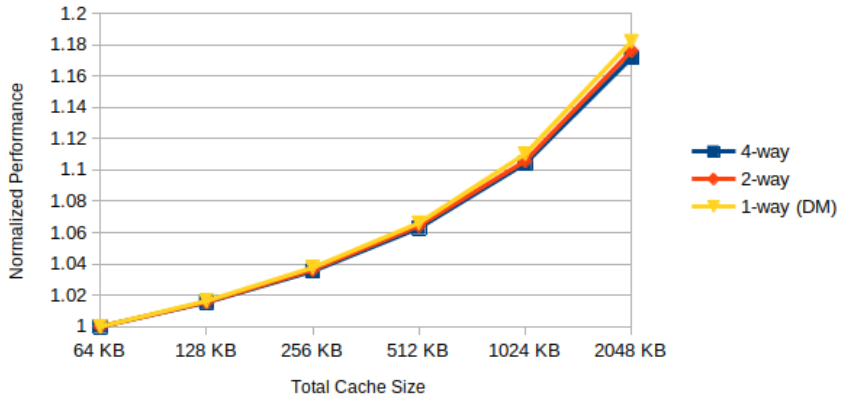


Figure 5.8: Average normalized performance increase with respect to increasing cache size in all graph data.

The slight increase in slope suggests a better overall performance may be achieved for cache sizes larger than 2 MB. Another observation is that different graph data benefits from increased cache size in different amounts. For example, performance increases in WND, AS, and WT are relatively low. Large graphs with a lower ratio of vertices to edges, such as SP and TC, show more stable performance increases. Moreover, for each associativity type, performance increase with respect to cache depth is consistent, showing a linear increase trend. Figures 5.9, 5.10, and 5.11 show the variation of execution time to total cache size in each associativity type and for all graph data.

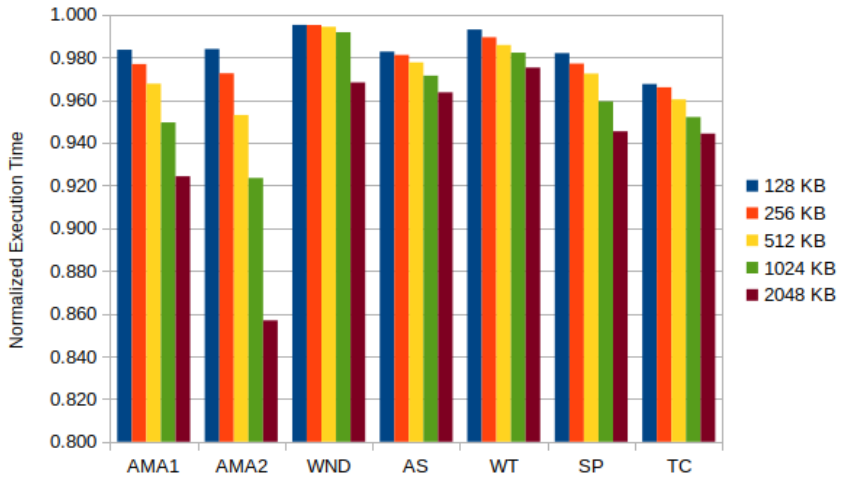


Figure 5.9: Improvement of normalized execution times by increasing cache size for direct-mapped L2 cache.

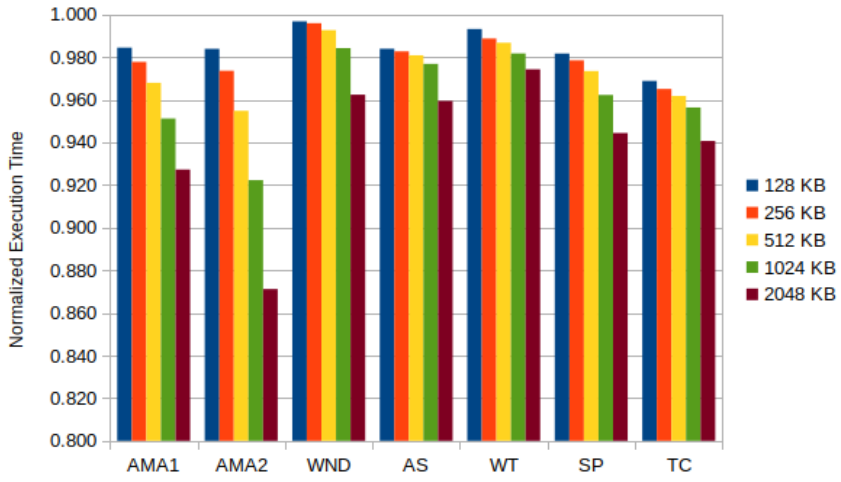


Figure 5.10: Improvement of normalized execution times by increasing cache size for 2-way set associative L2 cache.

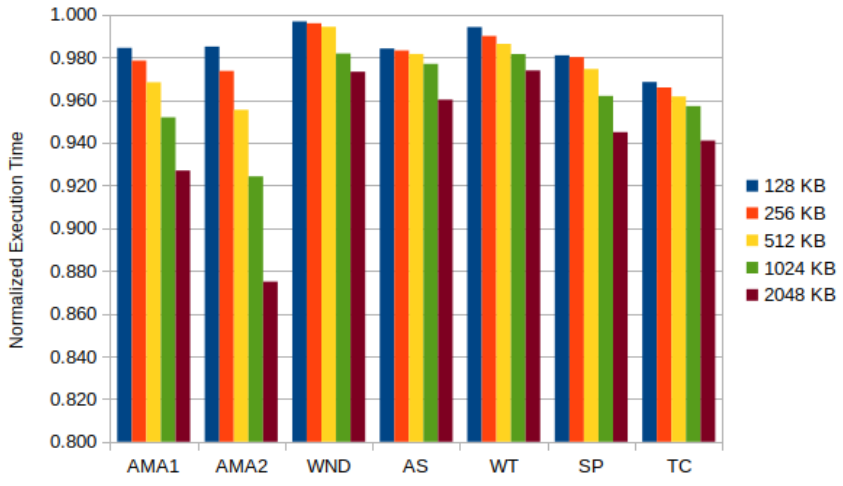


Figure 5.11: Improvement of normalized execution times by increasing cache size for 4-way set associative L2 cache.

5.8.1.2 Effect of L2 Cache Associativity on Performance

To evaluate the effect of associativity on performance, we compared the execution times of the three cache types while keeping the cache size equal. The performance of the direct-mapped L2 cache is taken as the baseline.

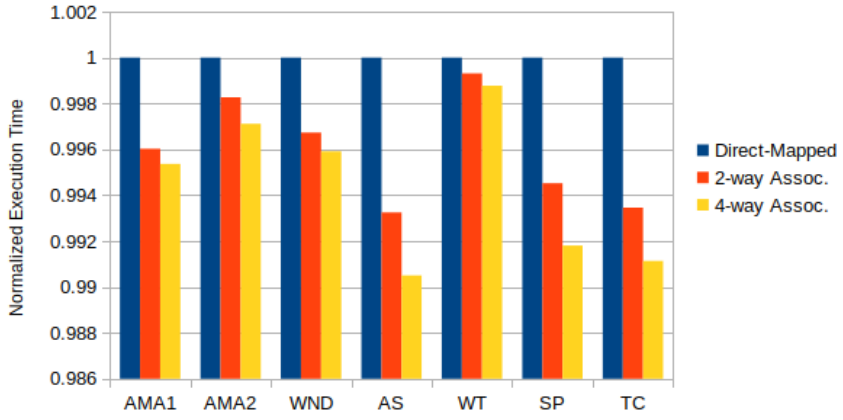


Figure 5.12: Performance comparison of L2 cache types with the total cache size of 64 KB.

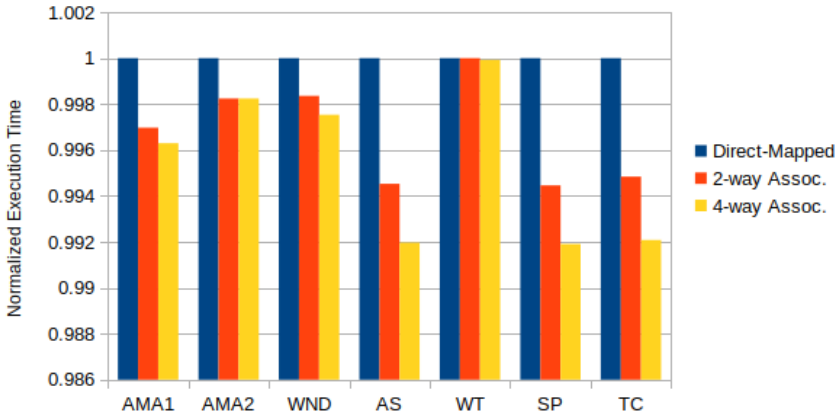


Figure 5.13: Performance comparison of L2 cache types with the total cache size of 128 KB.

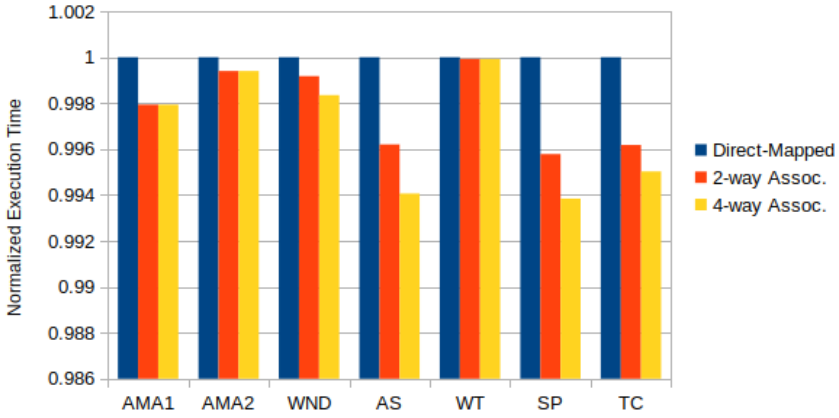


Figure 5.14: Performance comparison of L2 cache types with the total cache size of 256 KB.

Figures 5.12, 5.13, and 5.14 show the variation of normalized execution times for different associativity types under equal total cache sizes. Two conclusions can be drawn from the experiments concerning L2 cache associativity. Firstly, the performance differences between the cache types are generally small and almost insignificant in some cases. It is known that direct-mapped caches suffer from conflict misses, and adding associativity to the cache is a viable solution to reduce conflict misses. When associativity is introduced, the expected behavior of a direct-mapped cache is a relatively sharp increase in performance. As

the associativity is further increased, increase rate of performance should decline. The observed results in the experiments are consistent with the expected behavior since the performance gap between the direct-mapped cache and the 2-way associative cache is much larger than between the 2-way associative and 4-way associative. Secondly, the small differences between cache performances indicate that the number of conflict misses is low. In other words, cache block reuse is too low to significantly affect performance. Another indicator of this inference is as the total cache size increase, associative cache performance converges to direct-mapped since the increase in cache size also reduces conflict misses. As the effect of cache size on conflict misses overweight associativity, adding associativity to the direct-mapped cache makes less difference in performance.

5.8.1.3 Effect of Write-Miss Policy on Performance

The performance comparison between write-allocate and write-no-allocate cache architectures has been made. During the experiment, a direct-mapped L2 cache with the size of 32 KB is used, but the effects on performance are the same for all L2 cache types.

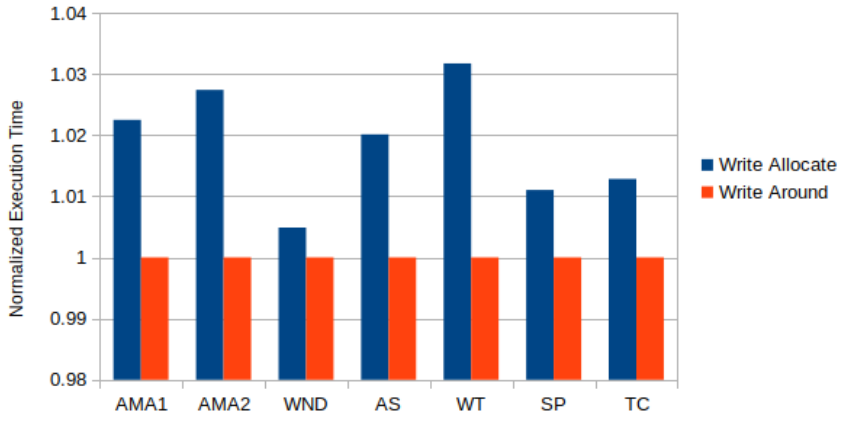


Figure 5.15: Effect of write-miss policy on performance in the L2 cache architecture. The execution time of the write-no-allocate is used as the baseline.

Figure 5.15 shows the effects of the write-miss policy on execution times. A

relatively low number of write misses cause a small difference in performance between the two architectures. The main reason for the difference in performance is because of the fact that single-module AXI transactions are more efficient than concatenated AXI module transactions. As a result, taking advantage of the single-word write requests to the AXI interface whenever possible leads to better overall performance.

5.8.1.4 Effect of Prefetch Buffer Size on Performance

In contrast to the previous study, a larger prefetch buffer size did not result in better performance. This is due to the realistic miss penalty rate of the memory subsystem. Prefetch buffer size is determined in terms of memory blocks. A single memory block corresponds to 4 words, i.e., 128 bits.

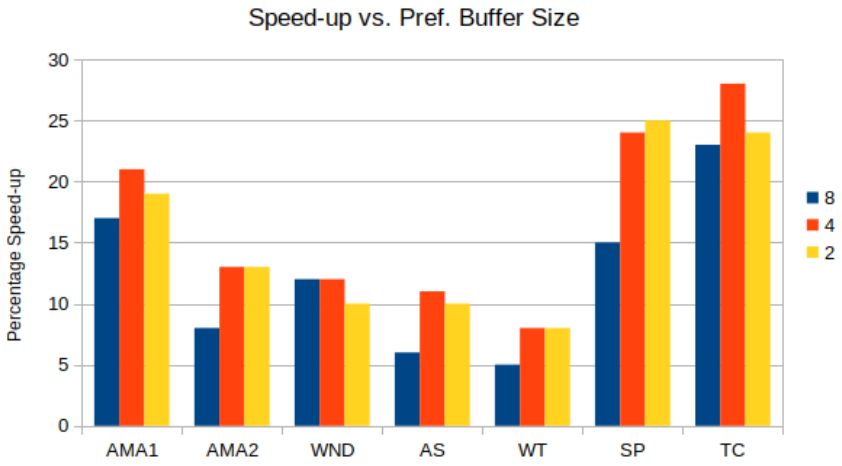


Figure 5.16: Effect of prefetch buffer size on speed-up values, with the BFS benchmark. Sizes of 2, 4, and 8 memory blocks are used.

As shown in Figure 5.16, a larger prefetch buffer size leads to aggressive memory requests from ESP. Figure 5.17 shows the variation of total memory requests for different prefetch buffer sizes. On the other hand, a smaller prefetch buffer size cannot fully utilize the system architecture. A sweet spot in size is determined as 4 in the experiments.

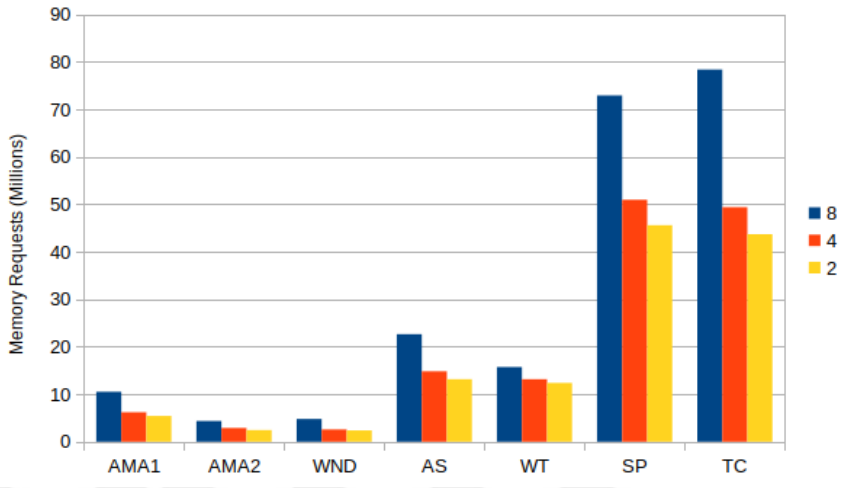


Figure 5.17: Total number of memory requests issued by SPM controller with different prefetch buffer sizes. Sizes of 2, 4, and 8 memory blocks are used.

5.8.2 Power Sensitivity Analysis

5.8.2.1 Effect of Clock Frequency on Power

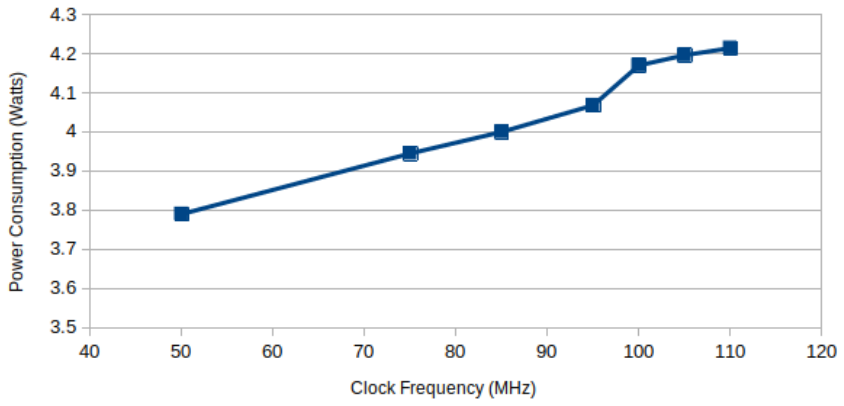


Figure 5.18: Change in power consumption with different clock frequencies. Tests are conducted with a direct-mapped L2 cache of size 2 MB.

Figure 5.18 suggests increasing the clock frequency also increases the power, but the rate of increase is not linear. Timing analysis of Vivado restricts the designs

to operate in certain frequency intervals. As the design approaches its maximum allowed clock frequency, the Vivado implementation phase struggles to place and route the design without failing endpoints.

For a design with the 2 MB direct-mapped L2 cache, the maximum operating frequency is determined as 110 MHz. Above the maximum, the design fails timing analysis, which means the design would not behave as expected. To improve the timing of our design, we used pipeline registers in the places where combinational logic paths were the longest. Long chains of combinational logic make the clock harder to route in the PL fabric.

5.8.2.2 Effect of L2 Cache Associativity and Size on Power

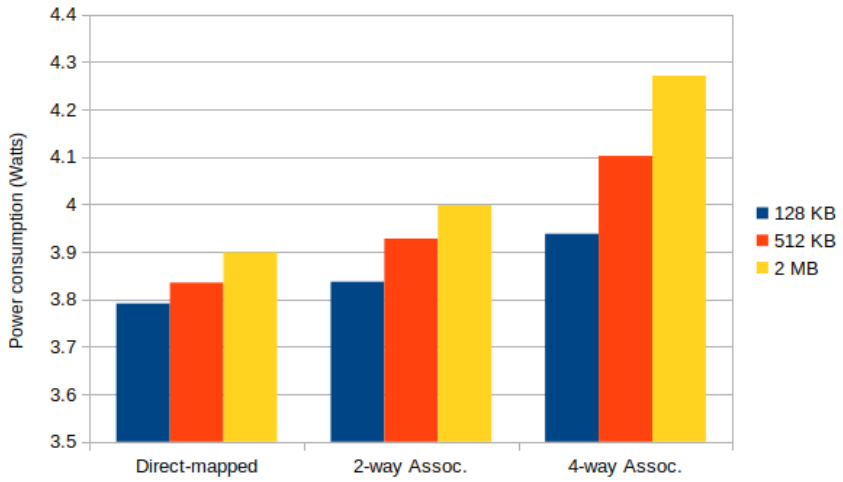


Figure 5.19: Power consumption of the designs with L2 cache sizes 128 KB, 512 KB, and 2 MB at 75 MHz clock frequency.

Figure 5.19 shows the effect of L2 cache associativity and total cache size on power. As expected, power consumption increases with increased cache size and logic complexity. A detailed examination of associativity and its effects on power is investigated further together with the power consumption of design modules.

As cache associativity increases, design complexity also increases. Doubling the associativity doubles the number of BRAM generators in the block design. Moreover, the size of the replacement array also increases.

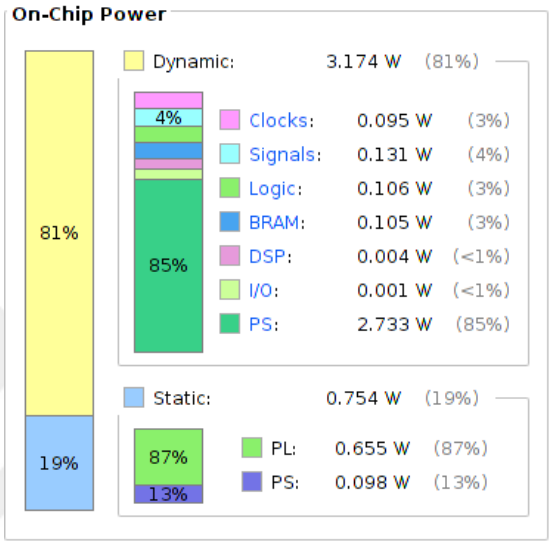


Figure 5.20: Power summary of the design with 512 KB 2-way associative L2 cache at 75 MHz clock frequency.

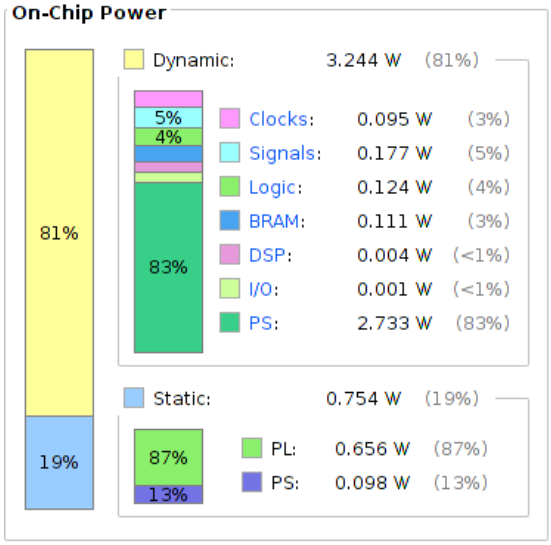


Figure 5.21: Power summary of the design with 512 KB 4-way associative L2 cache at 75 MHz clock frequency.

Figures 5.20 and 5.21 suggest that total power consumption increases in the design variants with the associative L2 cache due to the increased complexity. Comparator usage is increased due to associative tag checks, and an additional FPGA memory is needed to implement a replacement array. The replacement array is built using distributed RAM resources. However, a more efficient implementation would utilize BRAMs.

In Tables 5.7 and 5.8 we can see the sharp increase in power consumption of the L2 cache as we introduce associativity since logic needed to connect BRAM banks and replacement array is added to the design. Further increase in associativity also results in increased power consumption but to a lesser degree. Note that, although modules except the L2 cache have remained the same, changes in logic placement and routing during integration steps affected the power consumption of the modules.

Module	Power Consumption (W)
Ibex Core	0.03
SPM Controller	0.016
Request Arbiter	0.024
L1 Cache	0.111
MSHR Buffer	0.013
L2 Cache	0.081

Table 5.7: Power consumption of the modules in the design with 512 KB 2-way associative L2 cache.

Module	Power Consumption (W)
Ibex Core	0.03
SPM Controller	0.016
Request Arbiter	0.024
L1 Cache	0.114
MSHR Buffer	0.012
L2 Cache	0.098

Table 5.8: Power consumption of the modules in the design with 512 KB 4-way associative L2 cache.

Chapter 6

Conclusion and Future Work

This chapter gives a general overview of the experimental results and discusses possible future extensions to the design.

There are numerous possible improvements, mainly in the memory subsystem, to even reach higher speed-ups observed in simulation. Firstly, a bursting AXI module can be used instead of the single-beat AXI module since bursting AXI modules are highly efficient. However, the 4K Boundary requirement of the AXI protocol makes integrating a bursting AXI module into the graph processor's memory interface harder. A partial burst can be utilized for memory requests violating the 4K Boundary, which requires more complex logic. Yet, a certain improvement in memory transfer performance is possible with the utilization of the burst function of AXI.

Secondly, converting the L2 cache to a non-blocking cache would benefit the performance. An MSHR module can be integrated into the L2 cache, similar to the MSHR in the L1 cache. However, this time BRAMs can be utilized to generate the MSHR table.

Finally, in the current design, the ARM processor on the PS side is only used for I/O operations and verification. A framework can be developed to utilize the ARM processor in graph calculations as a co-processor. Furthermore, data load

can be shared between the processors on PL and PS sides in batches. The amount of work required to achieve this type of multi-processor architecture is large. The main drawback arises from the communication overhead between the processors, as the AXI latency could potentially outweigh the architecture's performance gains.

In conclusion, this thesis describes the implementation of a domain-specific processor on a real hardware platform and proposes a memory framework that uses SoC bus communication protocols. The baseline design is improved at the micro-architectural level, alternative design approaches are compared, and extensive performance evaluations have been done with real-life graph data of scale that could not be used in the simulation environment. Moreover, real-life measures such as energy consumption and resource utilization are provided. The maximum performance increase is observed in the SSSP benchmark; program execution time improved by almost 30% in large graphs. In the end, average speed-up values varying between 15% to 25% were observed in the experiments conducted with several benchmarks and graph data.

Bibliography

[1] M. E. Coimbra, A. P. Francisco, and L. Veiga, "An analysis of the graph processing landscape," *Journal of Big Data*, vol. 8, apr 2021.

[2] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 908–921, 2020.

[3] C. Gui, L. Zheng, B. He, C. Liu, X. Chen, X. Liao, and H. Jin, "A survey on graph processing accelerators: Challenges and opportunities," *CoRR*, vol. abs/1902.10130, 2019.

[4] A. Nohl, F. Schirrmeister, and D. Taussig, "Application specific processor design: Architectures, design methods and tools," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 349–352, 2010.

[5] G. Pulat, "Scratch-pad memory based custom processor design for graph applications," Master's thesis, Bilkent University, 2020.

[6] A. Yenimol, "Hardware/software co-design of domain-specific risc-v processor for graph applications," Master's thesis, Bilkent University, 2022.

[7] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, 2018.

[8] L. Chang and L. Qin, "Cohesive subgraph computation over large sparse graphs," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 2068–2071, 2019.

[9] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs*," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, CHE), p. 589–598, International World Wide Web Conferences Steering Committee, 2018.

[10] K. Akbudak, "Cache locality exploiting methods and models for sparse matrix-vector multiplication," Master's thesis, Bilkent University, 2019.

[11] G. M. Slota, "Graph theory lecture slides." Available at http://www.cs.rpi.edu/~slotag/classes/SP20t/slides/lec01.pdf (2020).

[12] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, D. Chiadmi, S. Hong, and H. Chafi, "Csr++: A fast, scalable, update-friendly graph data structure," in *International Conference on Principles of Distributed Systems*, 2020.

[13] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Ultrascale+ mpsoc and fpga families," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–37, 2015.

[14] P. L. PL, "Zynq-7000 all programmable soc overview," 2012.

[15] Xilinx, "Zcu102 evaluation board user guide." Available at https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd (2023/02/21).

[16] Xilinx, "Zynq ultrascale+ device technical reference manual." Available at https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm (2023/01/04).

[17] K. Manev, A. Vaishnav, and D. Koch, "Unexpected diversity: Quantitative memory analysis for zynq ultrascale+ systems," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 179–187, 2019.

[18] ARM, "Amba axi protocol specification." Available at https://developer.arm.com/documentation/ihi0022/latest/ (2023/03).

[19] Intel, "Avalon® interface specifications." Available at https://www.intel.com/content/www/us/en/docs/programmable/683091/22-3/introduction-to-the-interface-specifications.html (2022/09/26).

[20] IBM, "The coreconnect™ bus architecture." Available at http://www.scarpaz.com/2100-papers/SystemOnChip/ibm_core_connect_whitepaper.pdf.

[21] OpenCores, "Wishbone system-on-chip (soc) interconnection architecture for portable ip cores." Available at https://cdn.opencores.org/downloads/wbspec_b3.pdf (2022/09/7).

[22] Xilinx, "Axi interconnect v2.1 logicore ip product guide." Available at https://docs.xilinx.com/r/en-US/pg059-axi-interconnect (2022/05/17).

[23] S. S. Math, R. B. Manjula, S. S. Manvi, and P. Kaunds, "Data transactions on system-on-chip bus using axi4 protocol," in *2011 INTERNATIONAL CONFERENCE ON RECENT ADVANCEMENTS IN ELECTRICAL, ELECTRONICS AND CONTROL ENGINEERING*, pp. 423–427, 2011.

[24] Xilinx, "Amba axi4 interface protocol." Available at https://www.xilinx.com/products/intellectual-property/axi.html.

[25] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1270, 2005.

[26] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES '01, (New York, NY, USA), p. 61–66, Association for Computing Machinery, 2001.

[27] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, p. 48–57, jun 2020.

[28] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: Gpu, fpga and asic," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, pp. 100–107, 2022.

[29] Holler, Tam, Castro, and Benson, "An electrically trainable artificial neural network (etann) with 10240 'floating gate' synapses," in *International 1989 Joint Conference on Neural Networks*, pp. 191–196 vol.2, 1989.

[30] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic clouds: Specializing the datacenter," *SIGARCH Comput. Archit. News*, vol. 44, p. 178–190, jun 2016.

[31] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[32] Y. Wang, C. Ding, Z. Li, G. Yuan, S. Liao, X. Ma, B. Yuan, X. Qian, J. Tang, Q. Qiu, and X. Lin, "Towards ultra-high performance and energy efficiency of deep learning systems: An algorithm-hardware co-optimization framework," *CoRR*, vol. abs/1802.06402, 2018.

[33] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[34] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, (New York, NY, USA), p. 151–160, Association for Computing Machinery, 2014.

[35] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[36] T. Schiwietz, T.-c. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," in *Medical Imaging 2006: Physics of Medical Imaging* (M. J. Flynn and J. Hsieh, eds.), vol. 6142 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 1279–1290, Mar. 2006.

[37] T. Sorensen, T. Schaeffter, K. Noe, and M. Hansen, "Accelerating the nonequispaced fast fourier transform on commodity graphics hardware," *IEEE transactions on medical imaging*, vol. 27, pp. 538–47, 05 2008.

[38] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 317–324, 2010.

[39] E. BUBER and B. DIRI, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pp. 1–6, 2018.

[40] K. Keutzer, S. Malik, and A. Newton, "From asic to asip: the next design discontinuity," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 84–90, 2002.

[41] T. Good and M. Benaissa, "Very small fpga application-specific instruction processor for aes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 7, pp. 1477–1486, 2006.

[42] M. Rashid, L. Apvrille, and R. Pacalet, "Application specific processors for multimedia applications," in *2008 11th IEEE International Conference on Computational Science and Engineering*, pp. 109–116, 2008.

[43] J. Constantin, A. Dogan, O. Andersson, P. Meinerzhagen, J. N. Rodrigues, D. Atienza, and A. Burg, "Tamarisc-cs: An ultra-low-power application-specific processor for compressed sensing," in *2012 IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, pp. 159–164, 2012.

[44] O. Muller, A. Baghdadi, and M. Jezequel, "Asip-based multiprocessor soc design for simple and double binary turbo decoding," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, pp. 1–6, 2006.

[45] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2017.

[46] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, 2020.

[47] S. Angizi, J. Sun, W. Zhang, and D. Fan, "Graphs: A graph processing accelerator leveraging sot-mram," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 378–383, 2019.

[48] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, 2015.

[49] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[50] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *CoRR*, vol. abs/1503.07241, 2015.

[51] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[52] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 908–921, 2020.

[53] N. Kapre, "Custom fpga-based soft-processors for sparse graph acceleration," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 9–16, 2015.

[54] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Jr. Knight, and A. DeHon, "Graphstep: A system architecture for sparse-graph algorithms," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 143–151, 2006.

[55] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric

graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, 2014.

[56] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), p. 217–226, Association for Computing Machinery, 2017.

[57] S. Angizi and D. Fan, "Graphide: A graph processing accelerator leveraging in-dram-computing," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, (New York, NY, USA), p. 45–50, Association for Computing Machinery, 2019.

[58] lowRISC Company, "Ibex github page," 2021.

[59] Xilinx, "Zynq ultrascale+ mpsoc processing system ip." Available at https://www.xilinx.com/products/intellectual-property/zynq-ultra-ps-e.html.

[60] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2017.

[61] N. P. Jouppi, "Cache write policies and performance," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), p. 191–201, Association for Computing Machinery, 1993.

[62] Xilinx, "Block memory generator." Available at https://www.xilinx.com/products/intellectual-property/block_memory_generator.html.

[63] S. S. Omran and I. A. Amory, "Implementation of lru replacement policy for reconfigurable cache memory using fpga," in *2018 International Conference on Advanced Science and Engineering (ICOASE)*, pp. 13–18, 2018.

[64] V. Kathail, "Xilinx vitis unified software platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate*

Arrays, FPGA '20, (New York, NY, USA), p. 173–174, Association for Computing Machinery, 2020.

[65] A. Bundy and L. Wallen, *Breadth-First Search*, pp. 13–13. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984.

[66] S. Even, *Graph Algorithms*. Cambridge University Press, 2 ed., 2011.

[67] I. Rogers, "The google pagerank algorithm and how it works," 2002.

[68] K. Magzhan and H. M. Jani, "A review and evaluations of shortest path algorithms," *International journal of scientific & technology research*, vol. 2, no. 6, pp. 99–104, 2013.

[69] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[70] J. Ang, B. Barrett, K. Wheeler, and R. Murphy, "Introducing the graph 500," 01 2010.

[71] Xilinx, "Ultrafast design methodology guide for fpgas and socs (ug949)." Available at https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Optimization-Analysis.

[72] J. W. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, vol. 2914, pp. 225–236, SPIE, 1996.

[73] N. Engelhardt and H. K.-H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, IEEE, 2016.

[74] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 105–110, 2016.

[75] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.

[76] Intel, "Intel® core™ i7-9750h processor." Available at https://ark.intel.com/content/www/us/en/ark/products/191045/intel-core-i79750h-processor-12m-cache-up-to-4-50-ghz.html.

[77] amanusk, "The stress terminal ui: s-tui." Available at https://github.com/amanusk/s-tui.

[78] C. Yang, A. Buluç, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *ACM Trans. Math. Softw.*, vol. 48, feb 2022.