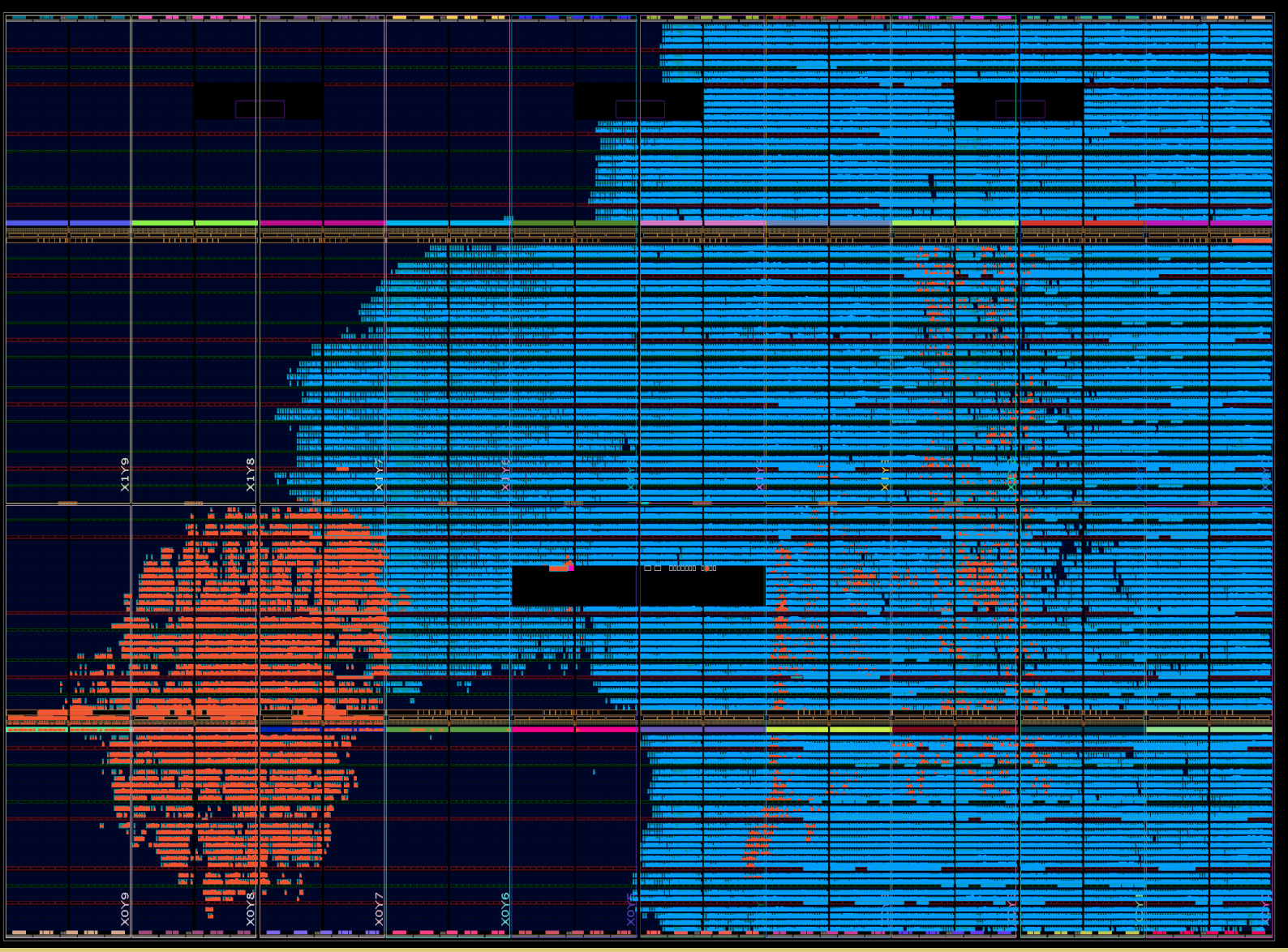


Pedro Miguel Parola Duarte

Application-specific Soft-GPGPU on Reconfigurable Substrates

Dissertação submetida para a satisfação parcial dos requisitos do grau de Mestre em Engenharia Electrotécnica e de Computadores

Julho de 2016

• U C •

Universidade de Coimbra

Background of cover image source: *FPGA implementation of a MicroBlaze (in orange) and one Compute Unit (NEKO) (in blue)*.

UNIVERSIDADE DE COIMBRA

FACULDADE DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Application-specific Soft-GPGPU on Reconfigurable Substrates

Pedro Miguel Parola Duarte

Dissertação para a obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente: Doutor Vitor Manuel Mendes da Silva
Vogal: Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Orientadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes
Co-orientador: Doutor Pedro Filipe Zeferino Tomás

Coimbra

Julho 2016

Acknowledgments

Aos meus orientadores, Professor Gabriel Falcão e Professor Pedro Tomás, tenho a agradecer toda a ajuda, sugestões e apoio, sem as quais não teria conseguido realizar este trabalho. Sem o apoio do Professor Gabriel certamente teria desistido, mas passo a passo, com pequenos avanços que muitas vezes não pareciam ajudar, conseguimos atingir o que nos tínhamos proposto a fazer. Mais ainda tenho a agradecer-lhe as perspetivas de futuro e todas as portas que me abriu ou deixou abertas. Também o Professor Pedro com as suas sugestões certeiras que sempre ajudaram a dar o passo em frente e sem as quais nunca se teriam atingido todos os objectivos propostos.

Agradeço ao Doutor João Andrade, por todas as tardes (dias) perdidas a dar-me apoio quando nada o obrigava a fazê-lo.

I thank Ziliang Guo, for all the help and support he provided to better understand MIAOW, without which I would still be in the system design stage. I would also like to thank Sam for the help provided in writing this thesis.

Ao Zé, ao Jamaro, ao Manuel, e todos os restantes colegas de laboratório, não só a ajuda que nunca me negaram, mas também todos os momentos de boa disposição (procrastinação) que proporcionaram.

Sem o apoio incondicional da minha família não seria possível ter chegado aqui. Aos meus Pais agradeço o permanente incentivo para aprender mais e maximizar o meu capital humano, bem como todo o apoio.

Por fim, o meu mais especial agradecimento, para a Diana, por todo o apoio, preocupação, e amor sem os quais teria enlouquecido provavelmente ainda antes de começar esta tese.

A todos, muito obrigado,

Bran thought about it. 'Can a man still be brave if he's afraid?'
'That is the only time a man can be brave,' his father told him.

- George R.R. Martin, A Game of Thrones

Abstract

Driven by the increasing performance requirements of modern scientific applications, general-purpose computing on modern *graphics processing units* (GPUs) have recently become a common approach. However, although GPU's massively parallel architectures provide high flexibility and performance, off-the-shelf devices have fixed designs that cannot be adapted towards the specific characteristics of the target applications. On the other hand, application-specific architectures can be designed and implemented in reconfigurable fabric that can be easily tailored to maximize the performance of a given application. However, such approaches often require a profound architectural redesign at the presence of minimal algorithmic changes.

To overcome both issues, a new solution is herein proposed that relies on emerging implementations of general-purpose massively parallel and programmable architectures on reconfigurable fabric, often referred to as soft-*general-purpose GPUs* (GPGPUs). Hence, the proposed solution adopts the recently developed MIAOW implementation of the AMD Southern Islands architecture, which is herein extended in order to support a wide set of 154 instructions (up from 42 in the original design). Furthermore, to tackle important performance bottlenecks associated with the critical path and with the memory access latency, a set of architectural improvements were introduced, providing a speed-up of up to 80x when implemented on a Xilinx Virtex 7 FPGA. Finally, a new compile-time methodology was proposed that, by trimming down the implemented resources, allows tailoring the soft-GPGPU architecture towards the application characteristics, leading up to 18% energy savings without any performance penalty.

Keywords

Soft GPGPU, application-specific architecture, re-configurable computing, architectures, power-and energy-efficient computing, parallel processing, FPGA

Resumo

Impulsionada pelas crescentes exigências de performance das aplicações científicas modernas, a computação de propósito geral (*general-purpose*) em *graphics processing units* (GPUs) tornou-se uma abordagem comum. No entanto, apesar da arquitetura massivamente paralela das GPUs fornecer alta flexibilidade e desempenho, o *design* do *hardware* em dispositivos comerciais é fixo, não podendo ser adaptado às características específicas de cada aplicação. Por outro lado, é possível conceber e implementar arquiteturas, específicas para uma dada aplicação, em tecidos reconfiguráveis. Estas, podem ser facilmente adaptadas para maximizar o desempenho de uma determinada aplicação. No entanto, estas abordagens frequentemente requerem um profundo redesenho arquitetónico na presença de alterações mínimas ao algoritmo.

Para superar ambas as questões, é aqui proposta uma nova solução que se baseia em implementações emergentes de arquiteturas general-purpose massivamente paralelas e programáveis em tecidos reconfiguráveis, muitas vezes apelidadas de soft-*general-purpose GPU* (GPGPU). Assim, a solução proposta adota a implementação recentemente desenvolvida chamada 'MIAOW' e que é baseada na arquitetura da AMD Southern Islands. O soft-GPGPU é aqui ampliado a fim de apoiar um conjunto amplo de 154 instruções (de 42 no projeto original). Além disso, para resolver os *bottlenecks* de performance associados com o caminho crítico e com a latência de acesso à memória, um conjunto de melhorias de arquitetura foram introduzidas, proporcionando uma melhoria de até 80x na performance, quando implementada numa *field-programmable gate array* (FPGA) Xilinx Virtex 7. Finalmente, uma nova metodologia de tempo de compilação foi proposta que, por diminuir os recursos consumidos pela implementadação, permite a adaptação da arquitetura soft-GPGPU para as características de uma dada aplicação, levando a uma poupança de energia de até 18% sem qualquer perda de desempenho.

Palavras Chave

GPGPU sintetizáveis, arquitectura aplicação específica, computação reconfigurável, arquiteturas, computação eficiente, processamento paralelo, FPGA

Contents

List of Figures

List of Tables

List of Acronyms

ALU	Arithmetic and logic unit
AXI	Advanced extensible interface
BRAM	Block RAM
BUFG	Global buffer
CPU	Central processing unit
CU	Compute unit
CUDA	Compute Unified Device Architecture
DDR3	Double data rate 3
DMA	Direct memory access
DSP	Digital signal processor
ECC	Error-correcting code
EDA	Electronic design automation
FF	Flip-flop
FGPU	FPGA general-purpose GPU
FIFO	First-in first-out
FPGA	Field-programmable gate array
GPGPU	General-purpose GPU
GPIO	General purpose input/output
GPU	Graphics processing unit
IoT	Internet of things
IP	Intellectual property (used in the context of hardware cores)
ISA	Instruction set architecture

JTAG	Joint test action group
LED	Light emitting diode
LSU	Load store unit
LUT	Lookup-table
MIAOW	Many-core integrated accelerator of Wisconsin
MIG	Memory interface generator
MIPS	Microprocessor without interlocked pipeline stages
MMCM	Mixed-mode clock manager
OCN	On-chip network
OpenCL	Open Computing Language
PC	Program counter
PE	Processing element
PLI	Programming language interface
PLL	Phase locked loop
RAM	Random-access memory
RTL	Register-transfer level
RTM	Runtime memory
SALU	Scalar ALU
SDK	Software development kit
SGPR	Scalar general purpose register
SIMT	Single-instruction multiple-thread
SM	Stream multiprocessor
SP	Scalar processors
Tcl	Tool command language
UART	Universal asynchronous receiver/transmitter
VALU	Vector ALU
VGPR	Vector general purpose register
VHDL	VHSIC hardware description language

List of listings

1

Introduction

Contents

Over the years, major advances in *electronic design automation* (EDA) have greatly simplified embedded system design. From the introduction of the programmable microcontroller to the general-purpose microprocessor, while passing through many smaller dedicated circuits—which are readily available—, today's system designer is able to exploit applications intrinsic parallelism, instead of focusing solely on the specific implementation details of every module. However, when it comes to high performance computing capability, embedded design is still lacking good alternatives, which is becoming critical, especially considering the rise of the *Internet of things* (IoT) and its demand for big data processing in increasingly smaller and mobile gadgets[2]. To tackle such problems, the current trends in big data processing are focusing on *general-purpose GPUs* (GPGPUs) or on exploiting *field-programmable gate arrays* (FPGAs).

1.1 Motivation

The recent evolution of *graphics processing units* (GPUs) to powerful multi-core accelerators with massive parallel processing capability created new programming paradigms, which resulted in two major frameworks, namely the *Compute Unified Device Architecture* (CUDA)[3] and *Open Computing Language* (OpenCL)[4]. With these tools, programmers can easily handle the processing of large amounts of data in relatively short periods of time. Accordingly, some manufacturers have recently released special embedded GPGPUs[5][6] in order to bring some of this computational power to smaller devices. Although efficient and fast, hard GPGPUs still constrain flexibility when creating a hardware system as they have no room for customization. On the other hand, for many application-specific systems there is no need for most of the GPU's functions, which results in wasted power and circuitry area when standard solutions are used out of the box.

Moreover, while architectures have significantly evolved in order to efficiently and massively exploit parallelism, FPGAs grew bigger to the point where dozens of soft-core processors could be crammed inside a single chip[7]. Manufacturers were quick to realize the enormous parallel capability of FPGAs and it didn't take long for the appearance of complex synthesis tools, capable of transforming programs in OpenCL and create specialized hardware, which can then be readily implemented in an FPGA (e.g., Xilinx SDAccel[8] and Altera OpenCL SDK[9]). Although not having the same throughput performance as GPUs in most applications, FPGAs offer the ability to create a fully customized system which, in addition, results in significant power savings[10]. Reconfigurable systems come with two main drawbacks: the available resources—which, throughout this document, are referred to as area—, as it restricts the size of the project that can be implemented, and the time it takes from system design (either through *register-transfer level*

(RTL) or using a specialized OpenCL-to-hardware tool) to bitstream completion. Hence, a small change in the software implies changing the hardware, which may require hours (if not days) waiting for the new hardware bitstream to be generated.

Soft-GPUs[11][12][13]—the implementation of a GPU on an FPGA—, are set out to address some of the problems mentioned above, namely the GPU customization problem, as well as the need to re-synthesize every change in the original algorithm and implement in the target FPGA. However, this recent addition still faces the limited area problem, which affects the number of computational units that can be added to the design. To mitigate this problem, we propose application-specific soft-GPUs, a minimal GPU implementation to perform a given task without the need to re-synthesize the hardware if the problem's dimension is altered. The resulting area savings can then be used to increase parallelism and, thus, throughput performance for a given task, and help saving power by reducing the number of unused hardware resources. Accordingly, the final result of the proposed work is an area and power optimized GPU core.

1.2 Objectives

Considering the above, the proposed objective is to provide future embedded system designers with dedicated, ready to use, and highly optimized GPU cores. Therefore, the main focus of this work is in the development of application-specific GPGPU cores, providing not only these but also the means so that others can create their own application-specific soft-GPUs. Furthermore, the following objectives are proposed:

- Develop application-specific GPGPU cores;
- Improve state-of-the-art GPGPU cores, such as MIAOW, by increasing functionality and throughput performance;
- Develop a framework to test the synthesized *instruction set architecture* (ISA) implementation;
- Be compatible with state-of-the-art programming languages, such as OpenCL or CUDA, in order to easily allow offloading application computational kernels to the soft-GPGPU;
- Develop a framework for easily designing application-specific GPGPU cores;
- Show the area and power benefits of allowing an adaptation of the computing resources to the application characteristics.

1.3 Main Contributions

In this thesis, we propose the modification of GPGPU cores in order to provide an optimized system, in terms of area and power, for a given task. Thus, a selection of a few popular, widely used, and computationally intensive applications is made, and a pre-existent soft-GPGPU, namely MIAOW[12], is re-engineered to be fully optimized for each application. Below, we highlight the main contributions of this work:

- *Designed system*: A Microblaze based system on a non-development board was developed, implemented, and validated. The board used in this work is primarily intended to serve as a data-center FPGA, meaning that it does not possess the features of development boards from Xilinx, which even have ready-to-use Microblaze systems. A detailed explanation on the design procedures for non-development boards is provided in Chapter 3.

- *Instruction testing script*: A comprehensive testing script was developed while validating the soft-GPGPU used in this work. This script lists the currently running instructions on the platform, and it can be seen in Appendices C, D, and E. A list of all working instruction is also provided in Chapter 5.

- *Corrections*: A number of corrections to the soft-GPGPU's functionalities were made in this work, which mostly focused in correcting broken instructions, or increasing the support for AMD's Southern Islands ISA. All the corrections were communicated to the original development team.

- *Improved Throughput Performance*: Modifications were made to the original soft-GPGPU with the intent of increasing throughput performance. These include separating the system into two clock domains—decreasing execution time by 1.22x—, and the addition of a *block RAM* (BRAM) module to the *compute unit* (CU), moving data closer to the processing cores—further decreasing execution time by, at least, 4x when compared to the original value. These changes are explained in detail in Chapter 4.

- *Benchmarks*: A timing profile for the implementation of the soft-GPGPU on the FPGA is also provided. This profile was obtained by benchmarking a set of well-known OpenCL applications. The results are provided in Chapter 5.

- *Application-Specific Cores*: Finally, the details behind the development of application specific soft-GPGPUs are provided in Chapter 4, explaining the changes made to the original core for each developed system. The resulting area and power savings

are displayed in Chapter 5, along with a consideration on the possibility of increasing the clock frequency to improve throughput performance, made on a per-system basis.

The enhancements proposed to the original compute unit resulted in a reduction of execution time of at least 76% (4x), reaching 98% (31x) in the most favorable scenario, while increasing the power requirement by only 10%. Furthermore, the cores' tailoring can result in power savings of up to 18%, leveraging the previous increase and releasing enough resources to instantiate a second CU on the design.

1.4 Outline

This thesis is organized in six chapters. After the introduction presented in this chapter, we discuss soft-GPUs, providing an insight into some of the most recent progresses in the area. Chapter 3 focuses on the details of the GPU architecture which is targeted in this work, as well as the FPGA design and validation procedures. We then focus on improving the existing cores, describing the architectural changes made in order to optimize the soft-GPGPU core for the chosen benchmarks, as detailed in chapter 4. Experimental results are presented in chapter 5 and the conclusions and future work directions are in chapter 6.

2

Soft-GPGPUs Overview

Contents

Over the years many high-performance computating accelerators have been proposed for *field-programmable gate arrays* (FPGAs). Initially, they mostly consisted of soft vector processors[14][15] which, although increasing the system's throughput for most tasks, lacked general support for conditional program execution. A few approaches of *graphics processing unit* (GPU)-like processors were also conceived[16], grappling on to some of GPU's design concepts, but they mostly focused on extending a pre-existing processor's capability to support a few GPU-type instructions (for instance, vector arithmetic operations). Recently, efforts have been made to implement complete compute units for OpenCL platforms[12][13], or *stream multiprocessors* (SMs) for CUDA platforms[11], on FPGAs. These implementations bring the parallel computing capabilities of GPUs to reconfigurable systems.

Soft-GPUs present two main advantages. First, they possess the ability to be programmed with a new binary, instead of having to recompile the entire hardware system for every small change in the application, resulting in significant time savings since hardware synthesis can take hours to complete. Furthermore, they allow developing in-depth hardware configurations, leading to finely tuned architectures and implementations.

In this chapter we focus on these recently developed platforms, introducing the proposed architectures and describing them into some level of detail. Afterwards we select the best suited candidate on which we concentrate efforts and develop our work.

2.1 FlexGrip

The very first communicated implementation of a soft-*general-purpose GPU* (GPGPU) was FlexGrip (FLEXible GRaphIcs Processor)[11]. FlexGrip is a *register-transfer level* (RTL) implementation of an SM with multiple *scalar processors* (SP). Most of the system was written in *VHSIC hardware description language* (VHDL), while a few modules were created using MATLAB's Simulink and later converted to RTL. It is based on the *single-instruction multiple-thread* (SIMT) model, in which all SPs are running the same instruction on different threads of execution. The soft-GPGPU core features a five stage pipeline which can be observed in Figure 2.1. The stages are organized as Fetch, Decode, Read, Execute and Write.

The SM receives a program in warps, i.e., a collection of threads which share the same program counter. To dispatch these warps there is a warp unit (i.e., warp scheduler) coordinating instruction execution while maintaining information about each warp, such as a thread mask, responsible for controlling which threads are executed; a state register, indicating the current state of the warp, which can be either Ready, Active, Waiting or Finished; and the program counter. The warp scheduling is done based on a round-

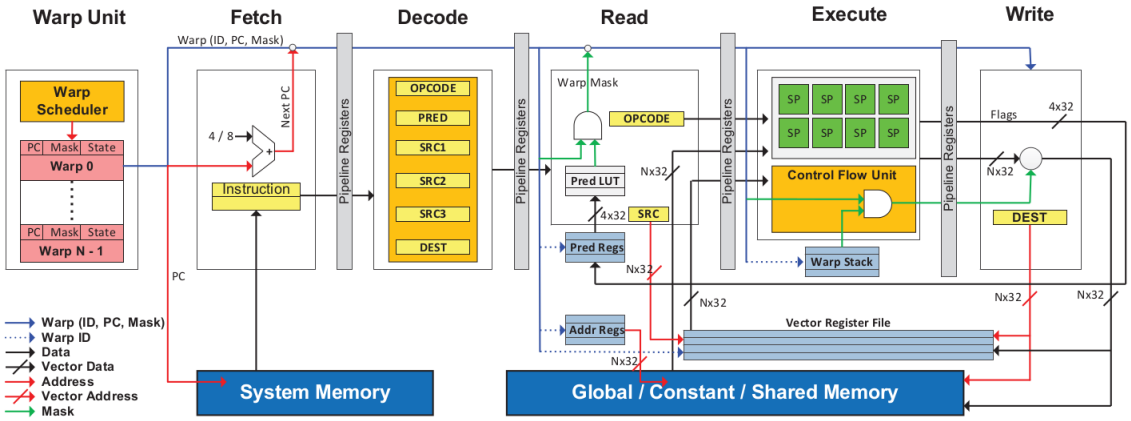


Figure 2.1: FlexGrip Streaming Multiprocessor's pipeline. The 5 basic stages are 'Fetch', 'Decode', 'Read', 'Execute' and 'Write'. The warp unit is responsible for keeping information about warps. 'Fetch' grabs the instructions from memory and delivers them for decoding. The 'decode' stage extracts key information from the instruction, for instance the operation to be performed, the source and output operands and predicate data. 'Read' is responsible for delivering the data sources identified in the decode stage to the execute stage. 'Execute' performs the operation, providing the results to the 'write' stage, which is responsible for storing data. Image courtesy of the FlexGrip development team.

robin algorithm, i.e, attributing equal time slots to each warp and going through them using a circular mode. To start execution, the warp unit passes the base program counter to the 'fetch' stage causing an instruction load for the current warp, the instruction is then delivered for decoding in the next stage and the program counter is incremented, pointing to the next instruction. The 'decoding' stage has to break down the instruction in order to extract the opcode, operation to be executed, source operands, and the result's destination, as well as predicate data.

Upon receiving the details for the source operands the 'read' unit determines if the values should be obtained from the vector register file or from memory, either shared or global, and will perform loads accordingly, delivering the obtained data to the 'execute' stage. If the data is to be read from memory, a special set of registers, designated as address register file, is used to determine the address to access. Should the instruction include the optional predicate flags, these will be used to obtain a predicated instruction and the thread mask will suffer an update, resulting from the combination between the existing mask with the obtained value. After obtaining the source operands and updating the thread mask, these values arrive at the 'Execute' stage. Here, a number of SPs run the opcode, extracted in 'Decode', on the operators, obtained in 'Read'. This execution follows the control of the thread mask, which selects the SPs that should run the instruction. Apart from the SPs there is one control flow unit which is responsible for branch and barrier instructions. Finally, the 'write' stage is responsible for storing the results. As was the case with reads, writes can be performed in either the vector register file, for

intermediate data, in the address register file, for addresses, in the predicate register file, for predicate flags, or in global memory, for final results.

The main novelty in FlexGrip is the support for direct GPU compilation, i.e., the binaries to program the system are generated by unmodified standard NVIDIA tools. The design is based on NVIDIA's G80 instruction set, which is compatible with CUDA 1.0, and has 27 working integer GPU instructions. With just these instructions FlexGrip is able to run 5 CUDA benchmarks, namely matrix multiplication, matrix correlation, matrix transpose, bitonic sort, and autocorrelation. All of these benchmarks are implemented using only integer operations as there are no floating-point arithmetical units, meaning that the core has reduced functionality for most real life applications.

In order to run, the SM needs to be connected to a MicroBlaze microprocessor, which acts as the host processor and supplies the SM with both instructions (program binary) and data, upon which processing will recall. It is worth noticing that in [11] the authors implemented an SM with eight SPs on an ML605 Virtex-6 board but were capable of creating a system with up to 32 SPs in simulation environment. This means that, had the board presented sufficient resources, a full Fermi SM (32 CUDA cores) could be implemented, which is a remarkable achievement, even if we consider the reduced functionality, like the lack of floating-point operations.

2.2 MIAOW

While FlexGrip focused on NVIDIA's architecture, a different approach was taken by *many-core integrated accelerator of Wisconsin* (MIAOW)[12]. MIAOW is based on AMD's notion of an OpenCL compute unit. Instead of presenting multiple SPs, OpenCL's compute unit has a single scalar *arithmetic and logic unit* (ALU) and multiple, up to four, *vector ALUs* (VALUs), which can operate in, up to, 64 scalar words at once. MIAOW was mostly developed in Verilog, utilizing a few C/C++ modules, through *programming language interface* (PLI), to model the memories, memory controllers and *on-chip network* (OCN). The *compute units* (CUs)' architecture can be seen in Figure 2.2.

One can visualize MIAOW's pipeline as being composed by six stages: 'Fetch', 'Decode', 'Issue', 'Register Read', 'Execute/Memory Access', and 'Write Back', where the fifth stage depends on the type of instruction being issued, since some of the stages can be decomposed in multiple sub-stages. A simplified scheme of the pipeline can be seen in Figure 2.3.

The CU receives a program in wavefronts, i.e., a collection of 64 work-items, known as threads in NVIDIA's terminology, which share the same program counter. Each wavefront has a set of associated data, such as the program counter, the wavefronts' identifier,

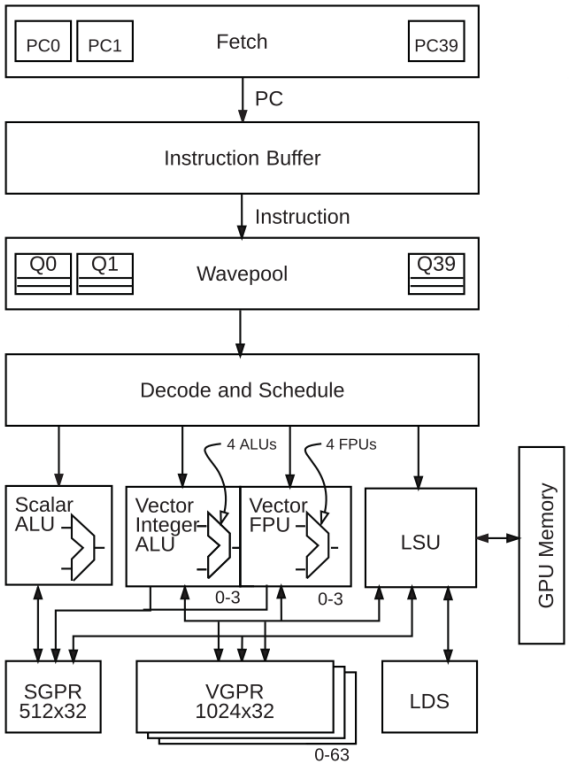


Figure 2.2: MIAOW's compute unit architecture. The architecture does not have a one on one correspondence with the pipeline. The blocks Fetch, Instruction Buffer, and Wavepool all belong to the Fetch stage of the pipeline. Decode and Schedule correspond to the Decode and Issue stages, respectively. Scalar ALU, Vector Integer ALU, Vector Floating Point ALU, and LSU all correspond to the Execute/Memory Access stage. All other modules are memory, either register files, SGPR and VGPR, local memory, LDS, or global memory, GPU Memory. Image courtesy of the MIAOW development team.

and the base address for both scalar and vector registers, and local memory. The 'Fetch' unit is the input port for instructions, therefore it has to receive the supra-mentioned data and place the wavefront on a queue, known as the 'Wavepool', where it waits until being selected for decoding. The selection process is done in round robin mode. At any given time, up to forty wavefronts can be present in the CU.

Upon selection, a wavefront is passed to the 'Decode' stage. There, a few parameters are extracted such as the opcode, operation to be executed; the source operands, which can range from one to three; the destination, and multiple flags, depending on the type of instruction. A few instructions use double word length, requiring two fetches and the

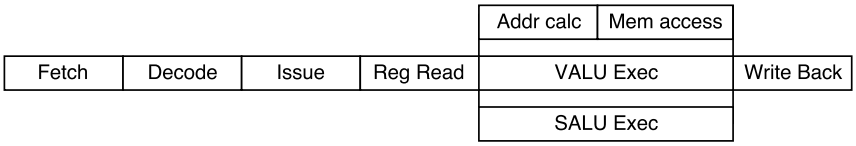


Figure 2.3: MIAOW's follows a six stage pipeline. The instruction is fetched and then decoded. After decoding it will wait until scheduling, or issuing, can be done. Once issued, the necessary operands will be read from the register file and the operation will be executed, ending with a register write back. The execution of a memory operation will require an address computation prior to the memory access.

joining of the two halves, before the extraction process—decoding—can begin. Based on the extracted values, the 'Decode' unit will automatically select the type of execution unit to be used, either VALU, SALU, or *load store unit* (LSU); and translate logical register addresses into physical addresses. The decoded instruction then reaches the 'Issue' stage where it waits until all dependencies have been resolved, only starting execution when all operands are ready to be accessed. If the instruction happens to be a "barrier" or a "halt", the 'Issue' unit will handle it immediately, not requiring any intervention from the remaining stages. For all other instructions, as soon as the operands are ready, it is scheduled for execution, causing a read from the register files, 'Register Read'. According to the unit selected in the 'Decode' stage, one of three possible types of operation will be executed. If the instruction operates only on scalar operands, then SALU will be selected and an arithmetical or logical operation will be performed on the operands. A different scenario occurs for vector instructions, as VALU will be selected and multiple values will be operated at once, each value corresponding to a different thread. To determine which threads in the vector are executed, a mask, called "execute mask", will be read. This mask can be read and written to, by normal scalar operations, meaning that there can be a fine control over which threads execute at any given time. Finally, if a memory instruction is casted, the LSU will be activated and a memory access will be performed. Before issuing the memory access request, however, the LSU performs an address calculation. Once execution finishes, a 'Write-Back' will occur to either a result register, the execution mask or to the conditional control flags, which, among other things, serve as primary output for comparison instructions.

MIAOW is based on AMD's Southern Islands *instruction set architecture* (ISA)[17], which has been used in a few of the brands' boards. The implemented CU supports 154 instructions from the ISA, being able to run unmodified OpenCL applications, i.e., it can run the kernel generated by the standard compiler without hand tuning. It is the first soft-GPGPU implementation to incorporate floating-point operations, thus considerably extending the range of possible applications.

To run a program, a system including a MicroBlaze soft-processor and DDR3 memory is required in order to supply the instructions and data to the compute unit.

In simulation, MIAOW can have up to four VALUs, each supporting 64 threads, which represents a full AMD Southern Islands CU. In the FPGA, however, developers could only implement a single VALU due to area limits on the board used, a VC707 Virtex-7[12].

In summary, MIAOW presents a very realistic approach to GPGPU design on an FPGA since it is largely based on a real GPGPUs' ISA, and has support for a wide range

of instructions, but the lack of hardware resources available on the FPGA prevented the implementation of a full Southern Islands CU.

2.3 FGPU

The most recent addition to the soft-GPGPU repertoire is *FPGA general-purpose GPU* (FGPU)[13]. Unlike the previously presented MIAOW, FGPU implements its own ISA, a subset of MIPS assembly with extra, OpenCL-inspired, instructions, in order to create a soft SIMT processor. The RTL design was performed using VHDL and optimized for FPGA implementation.

Instead of presenting a single CU, or SM, as the previous two implementations, FGPU has multiple CUs, up to eight, each containing eight *processing elements* (PEs). Having multiple CUs requires an additional coordination effort, since there has to be a dispatcher which schedules jobs to the CUs, and a global memory access module, or controller, to coordinate the memory accesses from all the units. In FGPU, job dispatch to the CUs is done by a workgroup dispatcher, meaning that a whole block of wavefronts is assigned to a CU at once.

The CU is organized in four major types of blocks, the wavefront scheduler, the *runtime memory* (RTM), the processing elements, and the CU memory controller, as can be seen in Figure 2.4. Upon receiving a workgroup, the wavefront scheduler performs its

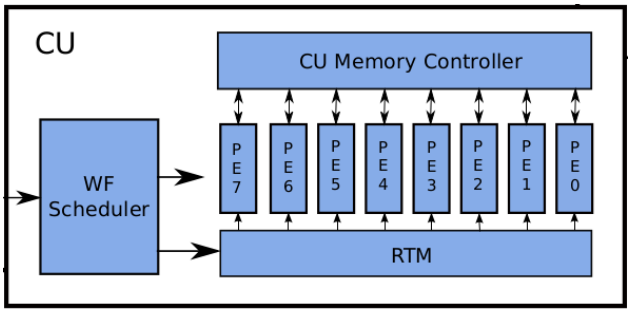


Figure 2.4: FGPU's Compute Unit main blocks. The CU has four major blocks, the wavefront scheduler, responsible for execution; the CU memory controller, responsible for memory accesses; the runtime memory, which holds data that can only be determined at runtime; and the processing elements, responsible for executing the instructions. Image courtesy of the FGPU development team.

division in wavefronts, composed of 64 work-items, or threads. A wavefront manager then controls which instruction is executed in the PEs, the current *program counter* (PC), and wakes up instructions, paused due to memory accesses, upon access completion. Before initializing execution, however, the wavefront scheduler also has to initialize the RTM. The RTM consists of a dual port *random-access memory* (RAM), which can be written by either the workgroup scheduler, when assigning a workgroup to the CU, or by the wavefront scheduler, when it schedules a wavefront for execution on the PEs. Its purpose

is to hold data which can only be determined at runtime, such as local indices of work-items, and the global offset of scheduled workgroups. Once the RTM is initialized, the wavefront can be sent to the PEs. The PEs are responsible for executing instructions. As seen in Figure 2.5, each PE has a register file, which can contain 2048 words, and an ALU, which can perform operations with, up to, three operands. Each PE will repeat a given instruction eight times, resulting in the same operation being performed 64 times, one for each work-item in the wavefront. If a memory instruction is issued, the CU memory controller is called. An incoming request is placed in a *first-in first-out* (FIFO) buffer by a controller, called station. If the request is a write then the address and data are written in the FIFO and the controller can serve the next request. In case of a read operation, the address is placed in the FIFO and the station will listen to the data read until the request is fulfilled. After the read has been served, a write-back unit will place the data in the register file.

Differently from previous implementations where the host *central processing unit* (CPU) was a soft-CPU, in FGPU the programs are sent to the soft-GPGPU unit by an ARM processor, present in the ZC706 Zynq board used[13].

FGPU's main novelty is the inclusion of multiple CUs on an FPGA. The design required not just the CUs but a control structure as well, in the form of a workgroup scheduler and a global memory controller. Designers implemented 18 assembly instructions, which, although limited, was sufficient to run four benchmarks, namely memcopy, vecmul and vecadd, FIR (5 taps), and cross correlation.

2.4 Summary

This chapter introduced the currently known soft-GPGPU approaches and described their architectural characteristics. All these approaches differentiate from each other. The first approach, FlexGrip, focused on NVIDIA's SM and on the CUDA programming model, while the remainder two focused on OpenCL's description of a CU. In what concerns the last two options, while MIAOW focused on implementing an existing ISA, FGPU developers created their own.

MIAOW's use of a real world ISA is interesting as it supports AMD legacy code, which provides the means to use a more diverse set of benchmarks and also allows taking conclusions that are applicable to real systems. Moreover, the support of floating-point operations further broadens the scope of possible applications when compared to FlexGrip. Furthermore, it also allows programs to be compiled using standard AMD tools, which means that no special compiler development is needed in order to test its functionality, as is the case with FGPUs' ISA. Since MIAOW has an extensive synthesized ISA, it

features support for currently available programs. For these reasons, MIAOW has been selected as the base CU in this thesis.

To gain further insight into the workings of MIAOW, the next chapter focuses on the Southern Islands ISA, explaining in detail the inner architecture of a Southern Islands compute unit. It also describes in detail the system's design and validation procedures, focusing not only in the instantiation and connection of modules, but the testing procedures as well.

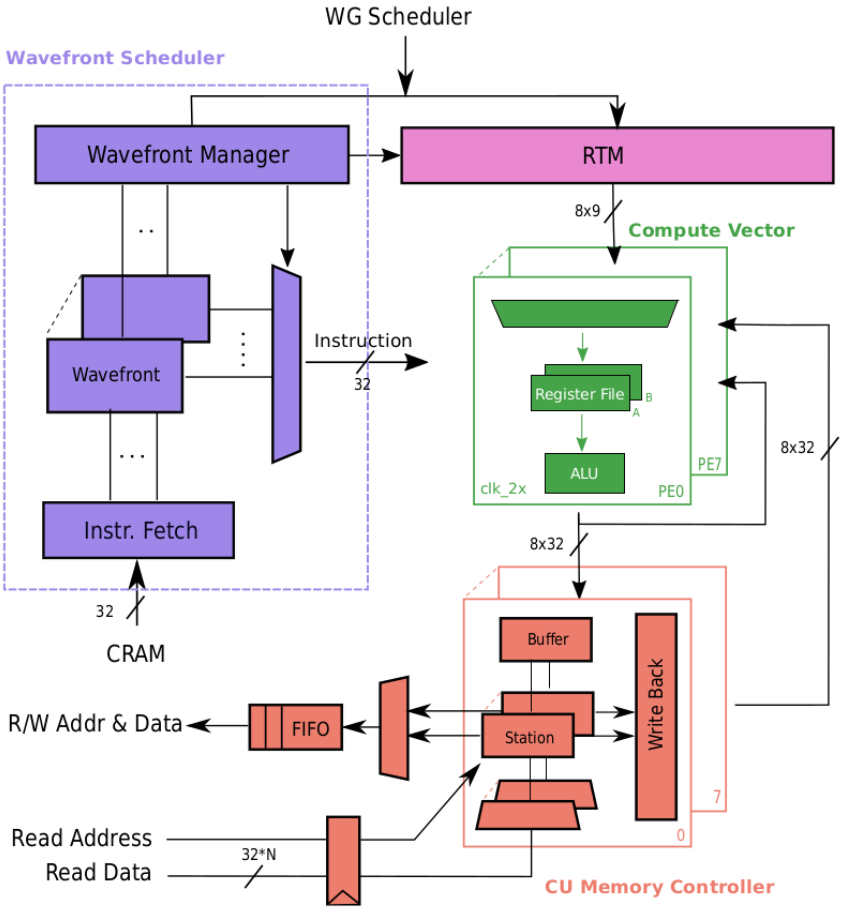


Figure 2.5: FGPU's Compute Unit main blocks internal organization. The wavefront scheduler receives complete workgroups, through the fetch, and divides them into wavefronts, these will later be dispatched by the wavefront manager. The RTM will receive data from both the workgroup scheduler and the wavefront scheduler and will allow executing wavefronts to access this data. A processing element will receive an instruction, gather the operands from the register file, and execute it in the ALU; if the instruction is a memory access it will create a request and pass it to the CU memory controller. In the CU memory controller, the controller, or station, receives memory requests placed in the buffer and dispatches them to a FIFO buffer; if a read is performed, when the requested data arrives a write back module will place this data on the register files. Image courtesy of the FGPU development team.

3

MIAOW Base Architecture

Contents

In the previous chapter the general characteristics of current soft-GPGPUs were reported, as well as their working mechanics in terms of pipeline organization. The most suitable candidate to work with was identified, and it consists of MIAOW, the implementation of a *compute unit* (CU) based on AMD's Southern Islands *instruction set architecture* (ISA)[17]. Taking this into consideration, and before delving into application-specific soft-GPGPUs, this chapter starts by describing key features of the selected ISA. Then, a few considerations are made on current limitations of the FPGA version of MIAOW, NEKO. Later, the bring-up procedures required to have a working system are outlined, detailing how the system is ported to the platform used in this work. The hardware validation process is described, along with a characterization of the instruction validation script developed, in terms of program flow. Finally, a list of corrections made to the original system is presented.

3.1 Base Instruction Set Architecture

MIAOW is based on AMD's Southern Islands ISA and its internal architecture is also inspired by the manufacturers corresponding device architecture. As with most other AMD *graphics processing units* (GPUs), MIAOW has three main modules responsible for kernel execution, namely, the host interface, the ultra-threaded dispatcher, and the compute unit. Hence, it operates as follows.

Initially, the host compiles a given kernel, which is meant to execute on the GPU, and loads it into memory. Afterwards, it specifies the memory region for the input data, and reserves space for the output. Finally, the host creates a command buffer to instruct the GPU, through the host interface, on how to execute the workload.

The host interface consists of a command processor. It is responsible for communicating with the host processor, and scheduling on-chip workloads. This interface is set to receive commands from the host through memory-mapped buffers. These can set pipeline state data, perform explicit dispatch/*direct memory access* (DMA) orders, or operations for memory/cache synchronization. After receiving the commands, the host interface acts by initializing state registers, scheduling the workloads received, performing the DMA operations required, or satisfying scheduled synchronizations.

The ultra-threaded dispatcher receives commands from the host interface, and is responsible for distributing work across the CUs. At the start of execution, this unit receives the workgroups—groups of threads which can belong to more than one wavefront—, and checks which compute unit has sufficient resources to accommodate the given wavefront. Upon finding a compute unit that can handle the workgroup, the dispatcher marks the

resources as occupied, sends the workgroup and all related information, such as allocated memory buffer addresses, to the selected unit, and emits a command to start execution.

The CU is responsible for kernel execution. Upon receiving a workgroup, the CU begins dispatching instructions to the execution units, such as, the *load store unit* (LSU), the *scalar ALU* (SALU), or the *vector ALUs* (VALUs). A CU can operate in more than one wavefront at a time, using its available execution units to achieve parallelism. Once a given workgroup finishes execution, the CU waits until a new one arrives. Each CU is composed by instruction logic units, such as fetch, instruction buffer, decode, and issue; scalar and vector ALU units; scalar and vector register files; a high-bandwidth shared memory; and a one level cache, to increase memory access efficiency.

3.1.1 Compute Unit Architecture

As explained when describing MIAOW's architecture (see Section 2.2), after receiving the start execution command, the CU starts filling its pipeline, fetching one instruction per cycle, until either all the execution units are busy, or the end of execution is reached. After fetching one instruction, the CU decodes it, extracting the operation code—which defines the type of instruction—, the operands, the results destination, among other information that depend on the type of instruction.

The Southern Islands ISA has two major types of instructions—scalar and vector. A scalar instruction operates on a single word (32-bit) which is shared between all the threads in the wavefront. A vector instruction, on the other hand, operates in up to 64 threads at the same time, applying the execution mask, a 64-bit vector in which each bit corresponds to a different thread. This mask marks the threads which are affected by the instruction, and which are not. Operations of either type are further subdivided according to the execution unit used, either the ALU, for computation, or the LSU, for memory access. The operations that use the ALU are named after either by the number or type of operands; or by the operation performed. For memory operations, division is made between the type of memory accessed, which can be local data share—local to the compute unit—, or external memory—through cache. The complete set of instruction types is described in Table 3.1.

After knowing the type of instruction to be performed and, consequently, the execution unit, the operands are fetched. The possible operands, as well as their number, depend on the specific instruction type. Scalar and vector instructions can use the general purpose registers available, either scalar or vector, respectively, as inputs for operations. Furthermore, scalar instructions can operate on masks, like the execution mask; the vector condition code, which stores results of vector comparisons; the scalar condition code, which stores either the results of comparisons between scalars, or the carry-out of SALU

operations; the program counter; and on literal constants. Vector instructions can also access the *scalar general purpose registers* (SGPRs), the execution mask, and literal constants; apart from having access to the *vector general purpose registers* (VGPRs), and to the local data share memory. The possible operand list is summarized in Table 3.2.

Upon gathering the operands, the execution unit identified when decoding the instruction can begin executing, if it is available. The implemented compute unit has four types of execution units, they are the LSU, the SALU, the floating-point VALU, and the integer VALU. Since every execution unit can perform more than one instruction, a second decode is performed inside this unit, selecting the exact operation to execute. A typical ISA[17] implementation defines the CU as having a single LSU, a single SALU, and four of each VALUs, allowing multiple instructions to be performed simultaneously.

3.2 NEKO

MIAOW follows AMD's definition of a CU closely, but has a few major differences, due to either being a work in progress, and also because of physical FPGA limits.

For instance, MIAOW does not possess either the local data share or the level one cache, as of yet. This causes two major problems. First, there is no support for data share instructions—these access the local memory of the CU—, which reduces the scope of usable applications. Furthermore, without cache, all memory requests have to access slow external memory, which increases memory access delay.

Due to FPGA limits, MIAOW can not yet entirely fit in a single design and, therefore, a reduced version was developed and named NEKO. The main difference between NEKO and MIAOW lies in the number of VALUs. Although a full CU would have eight VALUs—four integer and four floating-point—, MIAOW's developers had to limit this number to one of each, in order to fit the system on an FPGA. This reduces the available parallelism, since the ability to execute more than one instruction using the same VALU type at once is lost. When choosing the applications to run on NEKO, the current limitations of the CU have to be considered. This work uses the FPGA version of MIAOW, NEKO, since it is intended to have a working system on existing platforms. This version is distributed across 189 files, including major, sub-modules, and sub-sub-modules, which roughly amount to 35 thousand lines of code.

3.3 FPGA Design

The work in this thesis uses an Alpha Data's ADM-PCIE-7V3[1] board. This board features a Xilinx 7 series FPGA[18], specifically, XC7VX690T. The board used, and its FPGA,

Table 3.1: Type of instructions defined in Southern Islands' ISA

Operands	Type	Name	Description
Scalar	Scalar ALU	SOP2	Scalar instruction with two inputs and one output.
		SOPK	Scalar instruction with one inline constant input and one output.
		SOP1	Scalar instruction with one input and one output.
		SOPC	Scalar instruction with two inputs and producing a comparison result.
		SOPP	Scalar instruction with one inline constant input and performing a special operation (for example: branch).
	Scalar memory	SMRD	Scalar instruction performing a memory read from L1 memory
Vector	Vector ALU	VOP2	Vector instruction taking two inputs and producing one output.
		VOP1	Vector instruction taking one input and producing one output.
		VOPC	Vector instruction taking two inputs and performing one comparison.
		VOP3	Vector instruction taking three inputs and producing one output. Allows redirecting comparison outputs to scalar registers (instead of the vcc register).
	Vector Memory	Vector memory	Vector memory instructions (read / write) working on external memory.
	Local Data Share	Data share	Vector memory instructions (read / write) working on local memory.

Table 3.2: Possible instruction operands as defined in the Southern Islands' ISA

Type	Operand Name	Description
Scalar	SGPR0-SGPR103	Scalar general purpose registers
	VCC_LO	Lower 32 bits of the vector condition code
	VCC_HI	Upper 32 bits of the vector condition code
	TBA_LO	Lower 32 bits of the trap handler base address
	TBA_HI	Upper 32 bits of the trap handler base address
	TMA_LO	Lower 32 bits of the pointer to data in memory used by trap handler
	TMA_HI	Upper 32 bits of the pointer to data in memory used by trap handler
	TMP0-TMP11	Trap handler temporary registers
	M0	Memory register 0
	EXEC_LO	Lower 32 bits of the execution mask
	EXEC_HI	Upper 32 bits of the execution mask
	-16-64	Literal integer constants -16 to 64
	0.5	Literal constant 0.5
	-0.5	Literal constant -0.5
	1.0	Literal constant 1.0
	-1.0	Literal constant -1.0
	2.0	Literal constant 2.0
	-2.0	Literal constant -2.0
	4.0	Literal constant 4.0
	-4.0	Literal constant -4.0
	VCCZ	Result of the comparison between VCC and zero. Automatically updated with every change to VCC.
	EXECZ	Result of the comparison between the execution mask and zero. Automatically updated with every change to EXEC.
	SCC	Scalar condition code.
	LDS direct	Input read directly from local data share
	Literal constant	32-bit literal constant that follows the current instruction
Vector	—	All of the above
	VGPR0-255	Vector general purpose registers

Exception: According to the ISA [17], SMRD instructions cannot receive literal constants as input.

are different from the ones used by MIAOW's original development team. Therefore, it was necessary to port their system, redesigning it to attend to the board's features.

System design was made using Xilinx's Vivado Design Suite[19], which did not, originally, feature support for Alpha Data's board. To work around this issue, a modified version of the board support files from Xilinx SDAccel[8] was developed and used, effectively adding support to the board in Vivado[19].

After adding the board support files to Vivado, system design could ensue. The following subsection addresses the composition of the system supporting the CU's execution. Afterwards, the instantiation of NEKO is described, focusing on how it connects to the base system.

3.3.1 Base system

To design the system which supports the compute unit, Xilinx's tutorial for embedded design[20] was used. Since the board featured in the tutorial is different from the available one, it can not be followed directly. Nonetheless, the ideas behind system development remain valid and the work-flow is very similar to the one described in the tutorial.

The base system has four major components, which are a soft microprocessor, called Microblaze[21], a DDR3 RAM memory controller[22], a timer[23], and a debug module[24] for Microblaze.

Microblaze: The Microblaze soft processor[21] is responsible for controlling execution, like a regular host processor would. It also acts as an ultra-threaded dispatcher for the CU. Much like a regular host processor, Microblaze instructs the CU to execute a given kernel, performs the required memory operations to provide data to the CU, and retrieves results after computation. The Microblaze also controls the interaction between all components in the system. The developed system does not possess DMA capabilities, therefore, the microprocessor is responsible for handling memory transfers. Furthermore, since it acts as an ultra-threaded dispatcher, the Microblaze is also responsible for CU initialization procedures, such as setting initial register values, and satisfying the compute unit's memory requests. When instantiating a Microblaze microprocessor it is required to set the local memory size—from none to 128KB—, the cache size—from none to 64KB—, the clock domain, and whether a debug module should be present.

Memory Interface Generator: The memory controller, known as *memory interface generator* (MIG)[22], is responsible for intermediating memory accesses, as would be expected, but also for clocking and resetting the system. The latter is due to the constraint, set by Vivado, that MIG has to be connected to the boards clock and reset ports. Since only an adapted board support files is available, the memory controller needs some modifications in order to function correctly. For instance, due to Vivado's restrictions, the in-

put clock period needs to be set to a value higher than 1500ps, while the original value is 1250ps. The value 2500ps (or 400MHz) was used, since this is the highest frequency clock directly available on the board. Moreover, it is desirable to have the system clock with the highest possible frequency. MIG controls the system's clock by applying a ratio between the input and output clocks. This ratio is set to its lowest value (2:1) guaranteeing the maximum possible frequency for the system's clock. With this setting, for an input clock of 400MHz, the system's clock is 200MHz. To simplify memory initialization, and read/write operations, one needs to disable *error-correcting code* (ECC) functionality. Changing this functionality requires altering the memory part selected in MIG, to one that has a data width of 64bits, since if it had 72bits, similarly to the board's memories, it would be required, by Vivado, to have the ECC functionality enabled. This change requires searching for a memory part, on a predefined list in Vivado, with the desired data width while still matching the remaining characteristics, namely, the buffer size, the rank, and voltage. The closest memory that fits this criteria has a different total size—4GB, when the original had 8GB—, leaving part of the memory unused. Moreover, the difference in size reduces the addressable memory, which requires bounding the 'extra' address pins to zero. Otherwise, they act as antennas and alter the outcome of a memory access. After selecting the new part, its timing parameters are altered to match the ones in the board's memories. Alpha Data sets the memories to be disabled by default, the system needs to activate them by driving a pin (AA24) high ('1').

The timer module[23] is used mainly to monitor Microblaze's execution times.

Microblaze's debug module[24] allows the debugging of a Microblaze processor through a *joint test action group* (JTAG) interface. It also emulates a *Universal asynchronous receiver/transmitter* (UART) module, not originally featured in the board, enabling printing to the console.

All major modules are connected to the Microblaze by an *advanced extensible interface* (AXI)[25] bus, where the processor acts as a master, and all other peripherals are considered slaves. For debug purposes the *general purpose input/output* (GPIO) pins were also instantiated as part of an AXI slave peripheral. These are responsible for on board *light emitting diode* (LED) lights.

The final base system can be seen in Figure 3.1. System design followed an iterative process where each module was separately added, and its functionality was tested. This made system debugging easier as, at a given time, only one module was being debugged, instead of the whole system. Once the support system was designed, NEKO was added to it, along with any modules required to guarantee the CU's functionality.

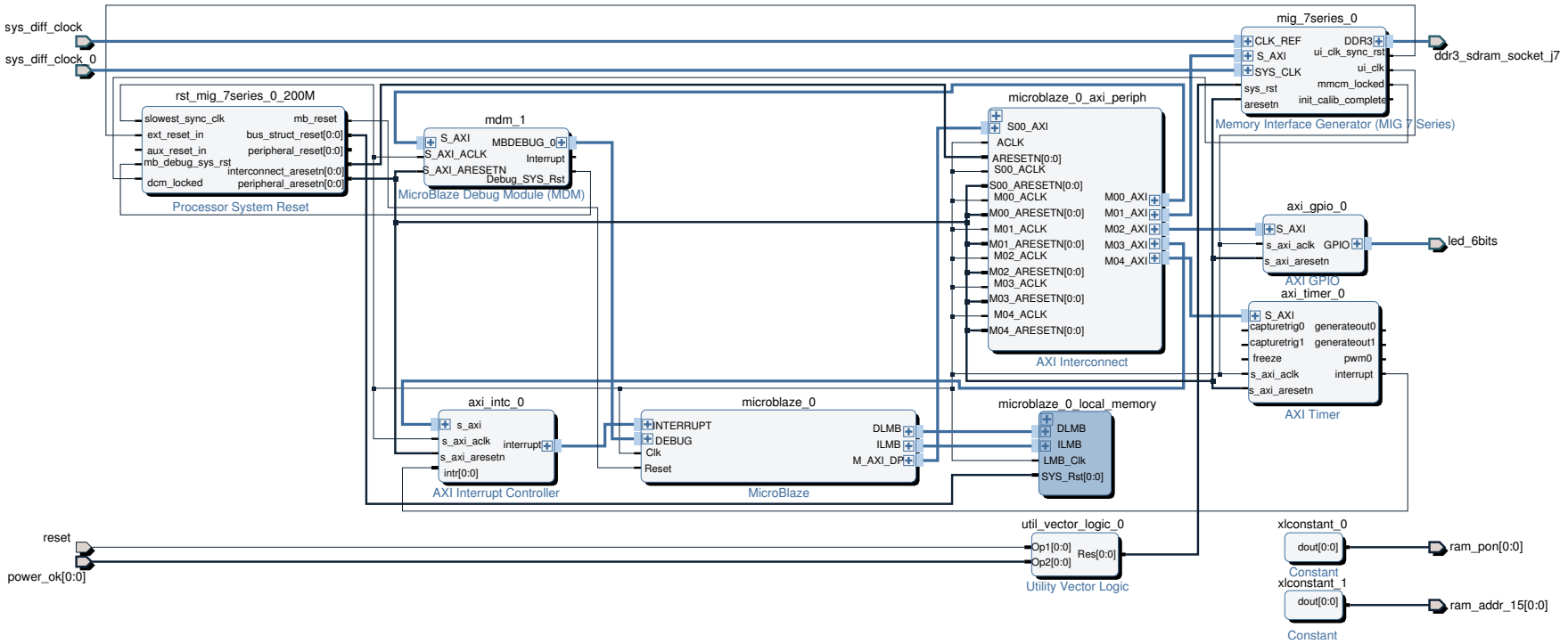


Figure 3.1: Base FPGA system. This system consists of a Microblaze microprocessor connected to a series of peripherals, namely, a set of general purpose pins (axi_gpio_0), a memory controller (mig_7series_0), a debug module (mdm_1), and a timer (axi_timer_0).

3.3.2 Full system

NEKO, like the other major components, needs to be connected to the Microblaze, through an AXI interface. To guarantee that NEKO, or MIAOW for that matter, remains compatible with all FPGA boards, the original development team decided that instead of adding AXI capability directly to the CU, this feature should be added to an intermediary interconnect peripheral. This peripheral acts as a bridge between the processor and the CU, by having a set of memory mapped registers that allows the compute unit to communicate with the processor, and vice versa. The peripheral needs to be developed and packaged through Vivado's *intellectual property* (IP) Integrator[26]. The work-flow of this tool can be seen in Xilinx's examples on how to create a custom IP module[27], and how to package it using the IP Integrator[28].

After adding the interconnect peripheral as an AXI slave, it is necessary to guarantee that all timing constraints are satisfied. In order to do so, the clock frequency should be set to 50MHz, as suggested by the original design team. This setting requires using a clock divider, known as clocking wizard[29], to reduce the system's frequency from the 200MHz that MIG[22] outputs, to the required value. The resulting clock then feeds all modules in the design, except for the MIG, that receives the board's clock signal directly.

Since NEKO is a CU independent of the FPGA vendor, manufacturer specific modules have to be separately instantiated. Thus, it is required to add a *block RAM* (BRAM) module to the system, which is used by NEKO as an instruction buffer, and as register files.

Once all modules are placed in the design, a top level entity—known as top level wrapper—is added to the project. This wrapper instantiates and connects the design and the CU and sets the system's inputs and outputs. Finally, the synthesis, implementation, and bitstream generation tools are executed. This allows the board to be programmed with the designed system. A simplified version of the design is shown in Figure 3.2.

After downloading the system's bitstream to the FPGA, it can be directly programmed through Xilinx *software development kit* (SDK)[30], in C. This programmability concedes an evaluation of the running state of NEKO. Nevertheless it does not allow for an assessment of the internal state of the CU. A lower level examination of the CU is required if a hardware debug of the system is to be performed.

3.3.3 Simulation system

To explore architectural details of a given system, at signal level, Xilinx provides the Vivado Simulator[31]. This tool provides the means to perform a timing simulation of the CU, by setting each individual port to the desired values for a preset number of clock cycles[32]. Exploring NEKO on this level of detail helps, not only to debug it, but to gain

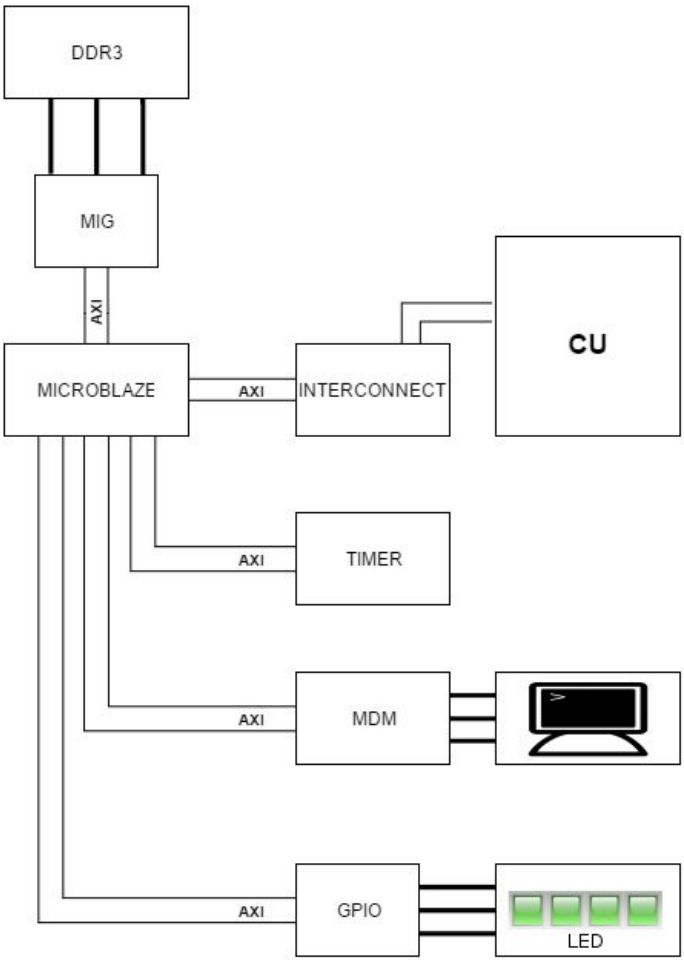


Figure 3.2: Simplified FPGA system. This system consists of a Microblaze microprocessor connected to a series of peripherals, namely, a set of general purpose pins (axi_gpio_0), a memory controller (mig_7series_0), a debug module (mdm_1), a timer (axi_timer_0), and the interconnect to the CU.

a level of familiarity with the pipeline architecture that would otherwise not be available from the high-level view given in the SDK.

The simulation system focuses on NEKO, removing all the complexity added from the remaining system. Although allowing a quicker debug, this also requires a new project to be created. Afterwards, simulation can ensue by either setting all signals manually and advancing the current time, or by running a *tool command language* (Tcl) script[33] containing commands. Either approach results in a simulation waveform, providing a deep analysis of the system.

3.4 FPGA Validation

System design and validation followed a lock-step approach, where a single major module was added, and immediately validated, at each iteration. This approach permits

a rapid identification of implementation errors, since only a module is being validated at a given time.

To start the iteration process, however, a baseline must be set. This primitive system is initially composed solely by a processor, Microblaze, and a feedback mechanism, namely, LED lights. Each iteration adds a new functionality: whether being more feedback mechanisms—like console-printing capability—, or the ability to measure time. Additionally, the memory space is extended by adding a MIG module to interact with external DDR3 memory.

Validation follows a thorough verification process, where all required functionalities for a given module are tested. Each new iteration re-validates the former one, to guarantee that there is no loss in functionality, and adds tests for the appended component. All the blocks that compose the base system are added and validated prior to inserting NEKO and the interconnect peripheral in the system.

As described in Sub-section 3.3.2, to allow a platform independent CU, an AXI peripheral that establishes communication between the Microblaze processor and NEKO needs to be instantiated. The interconnect module and NEKO are tested simultaneously since they are so intrinsically related. A two step validation is thus performed. First, the connection between the modules is assessed by writing values to the memory mapped registers in the interconnect and, afterwards, reading the values back. This tests the connection since a write to a memory mapped register is propagated to the CU and, when reading the value back, this value is directly provided by NEKO—instead of the register that was previously written to. Following this step, a set of simple programs in AMD's Southern Islands[17] machine code are run on the CU, through the Microblaze processor. These programs consist of a few instructions that operate on the data registers, created by consulting AMD's ISA[17]. The generated binaries are hard-coded to the C program in hexadecimal form, through an *unsigned int* table. The program flow can be seen in Listing 3.1. Microblaze starts by writing a few values to the CU's registers and populating the CU's instruction buffer with the given machine code. After initializing NEKO, the processor sends the start execution command and waits until the CU finishes. Finally, it recovers the resulting values present in the registers, and prints them to the screen.

Listing 3.1: Sample program flow used to test the compute unit

```
#define NEKO_EXEC (NEKO_BASE_ADDR + 24)
#define NEKO_INSTR_ADDR (NEKO_BASE_ADDR + 28)
#define NEKO_INSTR_VALUE (NEKO_BASE_ADDR + 32)
#define NEKO_REG_WRITE (NEKO_BASE_ADDR + 40)
#define NEKO_REG_ADDR (NEKO_BASE_ADDR + 44)
#define NEKO_REG_VALUE (NEKO_BASE_ADDR + 48)

unsigned int instructions[]={};//instructions are inserted here
```

```
int main(){
  //Set values in the registers
  for (i = 0; i < num_of_registers; i++){
    write(NEKO_REG_ADDR,i);  //set register address to i
    write(NEKO_REG_VALUE,i); //write register i with value i
    write(NEKO_REG_WRITE,1); //send the write command
    write(NEKO_REG_WRITE,0); //reset the flag
  }

  //Populate the instruction buffer
  for (i = 0; i < num_of_instructions; i++){
    write(NEKO_INSTR_ADDR,i);
    write(NEKO_INSTR_VALUE,instructions[i]);
  }

  //Start Execution
  write(NEKO_EXEC,1);

  //Wait for the execution to end
  while(read(NEKO_EXEC)==0);

  //Read the resulting from the registers
  for (i = 0; i < num_of_registers; i++){
    write(NEKO_REG_ADDR,i);           //set register address to i
    int value = read(NEKO_REG_VALUE); //read value in register i
    print(value);
  }


}
```

The program flow in Listing 3.1 allows the testing of each type of instruction defined in the ISA. Thus, a test script was developed with the goal of identifying the correctly implemented instructions from the complete listing present in the ISA [17]. The script is separated into three different programs, each working with either scalar, vector, or memory instructions. The main flow, for all three programs, is the same. For each type of instruction, one opcode (specific operation) is selected, following a sequential approach. The instruction binary is then generated in a set of functions which receive all the operands and output the corresponding machine code. The data used is randomly generated to perform a functional verification at the system-level. Thereupon, the CU is initialized with both the instructions and data, and execution starts. Once NEKO finishes executing, the results are recovered. Finally, the results obtained are passed to a function that compares them with the expected output. This process is repeated until all instructions are covered. The developed scripts may be seen in Appendices C, D, and E.

After exhaustive testing, a malfunction affecting all vector instructions (corresponding to a total of 104 instructions) was identified and corrected. To this effect, the system developed for simulation purposes was used to locate the faulting hardware modules, and correct their functionality.

Accordingly, after identifying incorrectly implemented instructions, the following procedure was used to identify the majority of the detected errors. Initially, the operand data is placed inside the register files (either scalar, vector, or both) and the broken instruction is implanted at the entry of the 'Fetch' unit. Upon fetching the instruction, the 'Decode' unit's output is analyzed, verifying if it selected the right execution unit and operands. Under those circumstances, the pipeline-flow progresses until the instruction is scheduled for execution. At this point, the validity of the operand data is verified, followed by an internal inspection of the execution unit, which has to select the operation to be performed by controlling its sub-modules. Afterwards, the result is analyzed, as well as the write-back procedures. This analysis permits a rapid identification of the main module responsible for an error. If at any given moment an incorrect signal or value appears, then the sub-units of the responsible module are thoroughly analyzed until the fault is corrected.

More broken instructions were identified and fixed. These include the vector memory writes, the floating-point reciprocal (needed for divisions), and correcting the usage of inline constants in memory operations. The debugging process for all these was the same as before, using Vivado Simulator. A list of all currently working instructions was compiled and is presented in the results chapter (Chapter 5). Each correction made was communicated to the original development team of MIAOW. Of those, some changes were already introduced in the original project's repository, whereas in other cases a discussion is ongoing to decide on the best approach to solve such errors.

3.5 Summary

This chapter described the features of a Southern Islands compute unit. The focus was then shifted to the differences between the CU used, NEKO, and the one described in the ISA[17], emphasizing the current limitations imposed by the FPGA technology.

Afterwards, the original system was ported to the available FPGA, requiring an iterative process of system design and validation. At each iteration, more functionalities were added, until a complete system, capable of supporting NEKO, was obtained.

Subsequently, a comprehensive script was developed to evaluate which subset of instructions from AMD's ISA[17] were functionally synthesized.

The tests made using the Microblaze system revealed a subgroup of malfunctioning instructions, which were corrected using a convenient simulation environment.

The next chapter focus on the improvements made to the base architecture, as well as on how the system was benchmarked, and, finally, on the development of application-specific GPGPU cores.

4

Application-specific GPU Architecture

Contents

In Chapter 3 useful insights over AMD's Southern Islands *compute unit* (CU)[17] were discussed. A detailed description of the system design and validation was also provided, along with a list of corrections made to NEKO[12].

Taking this into account, this chapter focuses on the improvements made to the original system to increase throughput. After describing such architectural modifications, the CU is benchmarked, establishing a comparison in the throughput performance between the original (unmodified) and the improved systems. Finally, the development of application-specific compute units is discussed.

4.1 Enhancing functionality and throughput performance

NEKO's support system uses Microblaze to perform the tasks of an ultra-threaded dispatcher. This unit is responsible for initializing state registers with data (see Section 3.1). Thus, an interface capable of directly accessing the CU's register files is required. Although NEKO featured such an interface for the *scalar general purpose registers* (SGPRs), this was not implemented for the *vector general purpose registers* (VGPRs).

Moreover, there is an added delay corresponding to the processing time for a given request, since every memory access is satisfied by the Microblaze processor. This delay can be mitigated either by accelerating Microblaze's response time, or by adding a small memory block inside the CU—moving data closer to the processing units.

4.1.1 Vector register direct access interface

Adding the interface to the vector register file requires changing two major blocks: the *advanced extensible interface* (AXI)[25] interconnect peripheral, and the compute unit's top level module.

The AXI interconnect peripheral has a set of memory mapped registers, through which the Microblaze processor[21] can communicate with NEKO. The new interface requires expanding the existing register set to support it. Since the vector length is defined as 2048 bits[17], and a Microblaze processor only outputs 32bit words, a set of 64 data registers is added to the peripheral. An address register is also set—controlling which VGPR is written. Additionally, to control which words of a vector are written, two 32bit registers contain the write mask. Finally, a special address is defined to signal a write command, which causes the values in the data registers to be propagated to the VGPR.

Due to the limited number of input ports in the VGPR file, the compute unit's top level module is responsible for multiplexing input signals. Therefore, a new entry is set to accommodate the interface port.

4.1.2 Dual clock domain

Due to architectural constraints, *graphics processing units* (GPUs) usually have lower clock frequencies—around 800MHz—when compared to *central processing units* (CPUs)—around 2-3GHz. Thus, having the entire system running at 50MHz constrains the potential system's performance. Furthermore, Microblaze[21] controls memory accesses, receiving the requests from the CU, and communicating with the *memory interface generator* (MIG)[22] to satisfy them. Increasing Microblaze's clock frequency, while retaining NEKO's, causes the CU to perceive lower access times. The addition of a second clock domain in the system requires redirecting the existing signals for all modules, except NEKO and its interconnect peripheral. The latter remain connected to the 50MHz clock from the clocking wizard module[29], while the former are directly fed by MIG's output with a frequency of 200MHz.

4.1.3 Internal block RAM memory

A further development to decrease memory access times is placing a small subset of *block RAM* (BRAM) inside the CU. This block brings the data closer to the execution units, decreasing latency. Moreover, since the BRAM is placed directly inside the CU, there is no interaction with the processor when performing memory accesses. BRAM blocks have fixed delays, which can be set to a single clock cycle, allowing complete control over access time.

To access external *double data rate 3* (DDR3) memory modules, the CU features a memory controller responsible for interacting with the interconnect peripheral and, consequently, with the Microblaze processor. This module receives a request being processed in the *load store unit* (LSU) and directs it to the host processor, to be satisfied. With the addition of the BRAM block, this module becomes responsible for directing the memory request to the right component. The selection between accessing BRAM or external DDR3 is made based on the requested address. In this work, the BRAM module is designed such that it represents the first 2^{20} (1048576) bytes in memory. The entire redirection procedure is transparent to both the LSU block and the Microblaze processor.

The Microblaze processor has access to the BRAM module through the preexisting memory interface, used to satisfy LSU's requests. Using this communication, the internal memory is initialized, containing data for processing.

The changes proposed to increase throughput performance can be visualized in Figure 4.1.

The LSU module relies on the memory controller to interact with the Microblaze processor. Adding the BRAM module does not alter this behavior, allowing the LSU to keep its modularity, even if the internal memory block is changed.

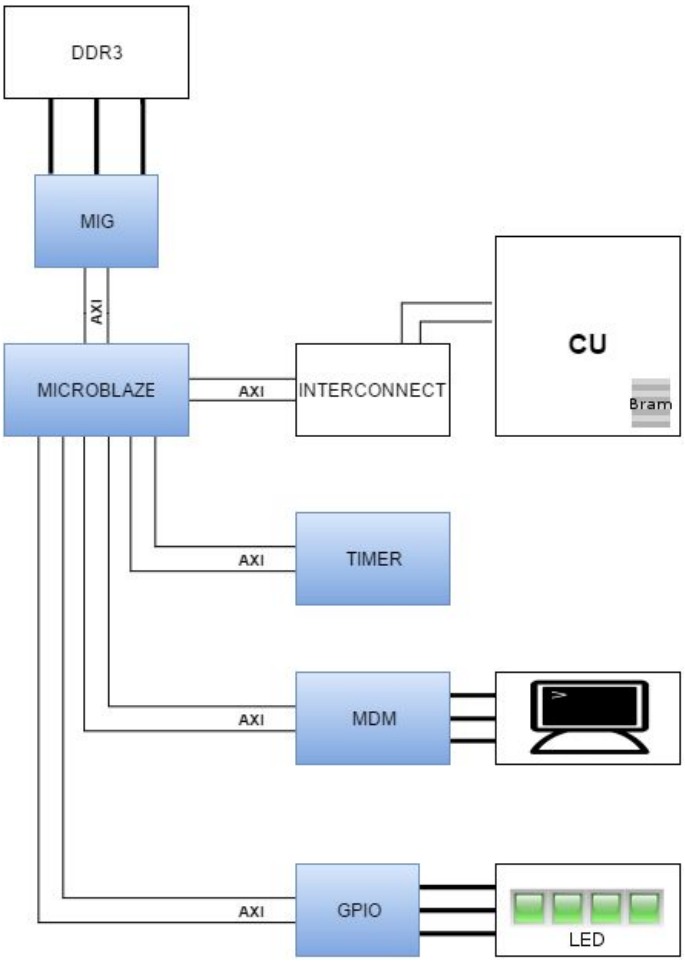


Figure 4.1: Simplified scheme of the improved FPGA system. The modules highlighted in blue are connected to the 200MHz clock domain, while the interconnect module is connected to the 50MHz clock. Inside the CU a small BRAM module was inserted.

4.2 System benchmark

Upon implementing and validating the proposed changes to increase throughput performance, a timing profile is used to quantify the improvements. This timing profile uses unmodified *Open Computing Language* (OpenCL) applications, which are compiled on AMD's standard tools, to compare the execution times of the systems.

This section focuses on the application selection and the general bring-up details, as well as on the benchmarking procedures.

4.2.1 Applications

Benchmarking applications were chosen from well established platforms, namely Multi2Sim[34] and Rodinia[35]. From these suites, a subset of five benchmarks, supported by the synthesized *instruction set architecture* (ISA), was selected. Three applications are provided by Multi2Sim's benchmark platform, namely, bitonic sort, floating-point ma-

trix multiplication, and integer matrix multiplication. The other two benchmarks used are K-means clustering, and matrix Gaussian elimination, both from the Rodinia suite.

The benchmarking suites provide the programs in OpenCL, a high-level language, which is compiled using AMD's CodeXL[36]. Figures 4.2 and 4.3 show this work-flow. Figure 4.2 provides an example of a high-level program which is then compiled into assembly code—and binary machine code—, as shown in Figure 4.3.

Due to a malfunction in the instruction that simultaneously performs a floating-point multiply-and-add, changes have to be made to the compiled binary code. All such instructions are, thus, replaced with separate floating-point multiplication and addition operations. Moreover, usage of inline constants is still conditioned in *vector ALU* (VALU) operations. This requires placing the constant value on an auxiliary register and using it as an operand.

4.2.2 Compute unit initialization

Prior to execution, the CU needs to be initialized with state data registers. This initialization is performed by the Microblaze, acting as an ultra-threaded dispatcher. Figure 4.3 clearly illustrates the use of pre-initialized registers. For instance, the first instruction loads a value on to scalar register zero (s0), using the value that was preset in scalar registers eight to eleven (s[8:11]) as the base address. Each application may set the register state by using specific system calls[37] [38]. This causes the number of pre-initialized register to vary on a per-application basis. Upon performing the compilation of a given kernel, CodeXL provides detailed information over the initial register state (see Figure 4.4).

On the selected applications, there are four major sets of scalar registers used, and three major vector registers used.

The first three sets of scalar registers contain memory descriptors. These are defined in AMD's Southern Islands ISA[17] as a combination between a 48 bit address and state data for the memory access. The first set is identified, in Figure 4.4, as IMM_UAV, and is present in scalar registers four to seven. This memory descriptor contains an offset for data gathering accesses. The second set of registers, IMM_CONST_BUFFER 0 (s[8:11]), contains the base address of OpenCL[4] call values. For instance, if a thread inquires its global ID this memory area is accessed, using a specific offset, to retrieve the required value. The third set of registers, IMM_CONST_BUFFER 1 (s[12:15]), holds a pointer to the space in memory where the kernel arguments are kept. The fourth—and final—set of scalar registers contains the thread group ID across the three possible dimensions (X, Y, and Z). In Figure 4.4, the flag *TGID_X_EN* is enabled ('1'), meaning that the register

OpenCL Source Code

```
1  __kernel void bitonicSort(__global uint * theArray, const uint stage,
2                            const uint passOfStage,const uint width,
3                            const uint direction){
4      uint sortIncreasing = direction;
5      uint threadId = get_global_id(0);
6
7      uint pairDistance = 1 << (stage - passOfStage);
8      uint blockWidth   = 2 * pairDistance;
9      uint sameDirectionBlockWidth = 1 << stage;
10
11     uint leftId = (threadId % pairDistance)
12                   + (threadId / pairDistance) * blockWidth;
13
14     uint rightId = leftId + pairDistance;
15
16     uint leftElement = theArray[leftId];
17     uint rightElement = theArray[rightId];
18
19     if((threadId/sameDirectionBlockWidth) % 2 == 1)
20         sortIncreasing = 1 - sortIncreasing;
21
22     uint greater, lesser;
23     if(leftElement > rightElement){
24         greater = leftElement;
25         lesser  = rightElement;
26     }
27     else{
28         greater = rightElement;
29         lesser  = leftElement;
30     }
31     if(sortIncreasing){
32         theArray[leftId]  = lesser;
33         theArray[rightId] = greater;
34     }
35     else{
36         theArray[leftId]  = greater;
37         theArray[rightId] = lesser;
38     }
39 }
```

Figure 4.2: OpenCL Code Example. This code is used as part of the bitonic sort benchmark, from the suite Multi2Sim.

following the third set is initialized with the workgroup ID for dimension X. Since the flags for dimensions Y and Z are not set, only the first register of this set is initialized.

The vector registers are pre-initialized to contain the thread IDs on the different dimensions (X, Y, or Z). The dimensions depend on the type of application. A program whose data consists of one dimensional arrays only operates on the X dimension. If working on a two, or three, dimensional matrix then the second, or third, dimensions—Y and Z, respectively—, are also operated upon. The first vector register (v0) contains the thread IDs on dimension X and should always be defined. The subsequent registers are only initialized if more than one dimension is used.

4.2.3 Benchmarking procedures

After compiling the applications previously selected, the kernel binary is exported from CodeXL [36] and hard-coded in the benchmarking program, running in Microblaze.

ISA Code - bitonicSort

Address	Opcode	Operands	Cycles	Functional Unit	Hex
0x000000	S_BUFFER_LOAD_DWORD	s0 s[8:11] 0x04	Varies	Scalar	C2000904
0x000004	S_BUFFER_LOAD_DWORD	s1 s[8:11] 0x18	Varies	Scalar	C2008918
0x000008	S_BUFFER_LOAD_DWORD	s2 s[12:15] 0x04	Varies	Scalar	C2010D04
0x00000C	S_BUFFER_LOAD_DWORD	s3 s[12:15] 0x08	Varies	Scalar	C2018D08
0x000010	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000014	S_MIN_U32	s0 s0 0x0000ffff	4	Scalar	8380FF00 0000FFFF
0x00001C	S_MUL_I32	s0 s16 s0	4	Scalar	93000010
0x000020	S_ADD_U32	s0 s0 s1	4	Scalar	80000100
0x000024	V_ADD_I32	v0 vcc s0 v0	4	Vector ALU	4A000000
0x000028	S_SUB_U32	s0 s2 s3	4	Scalar	80800302
0x00002C	V_LSHRREV_B32	v1 s0 v0	4	Vector ALU	2C020000
0x000030	S_LSHL_B32	s1 1 s0	4	Scalar	8F010081
0x000034	S_BUFFER_LOAD_DWORD	s3 s[12:15] 0x00	Varies	Scalar	C2018D00
0x000038	V_MUL_LO_I32	v1 s1 v1	16	Vector ALU	D2D60001 00020201
0x000040	V_LSHLREV_B32	v1 1 v1	4	Vector ALU	34020281
0x000044	V_BFE_U32	v2 v0 0 s0	Varies	Vector ALU	D2900002 00010100
0x00004C	V_ADD_I32	v1 vcc v1 v2	4	Vector ALU	4A020501
0x000050	V_ADD_I32	v2 vcc s1 v1	4	Vector ALU	4A040201
0x000054	V_LSHLREV_B32	v1 2 v1	4	Vector ALU	34020282
0x000058	V_LSHLREV_B32	v2 2 v2	4	Vector ALU	34040482
0x00005C	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000060	V_ADD_I32	v1 vcc s3 v1	4	Vector ALU	4A020203
0x000064	V_ADD_I32	v2 vcc s3 v2	4	Vector ALU	4A040403
0x000068	TBUFFER_LOAD_FORMAT_X	v3 v1 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA01000 80010301
0x000070	TBUFFER_LOAD_FORMAT_X	v4 v2 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA01000 80010402
0x000078	S_BUFFER_LOAD_DWORD	s0 s[12:15] 0x10	Varies	Scalar	C2000D10
0x00007C	S_LSHL_B32	s1 1 s2	4	Scalar	8F010281
0x000080	S_WAITCNT	lgkmcnt(0)	Varies	Flow Control	BF8C007F
0x000084	S_SUB_U32	s2 1 s0	4	Scalar	80820081
0x000088	V_AND_B32	v0 s1 v0	4	Vector ALU	36000001
0x00008C	V_CMP_EQ_I32	vcc 0 v0	4	Vector ALU	7D040080
0x000090	V_MOV_B32	v0 s0	4	Vector ALU	7E000200
0x000094	V_MOV_B32	v5 s2	4	Vector ALU	7E0A0202
0x000098	V_CNDMASK_B32	v0 v5 v0 vcc	4	Vector ALU	00000105
0x00009C	S_WAITCNT	vmcnt(0)	Varies	Flow Control	BF8C0F70
0x0000A0	V_MIN_U32	v5 v3 v4	Varies	Vector ALU	260A0903
0x0000A4	V_MAX_U32	v3 v3 v4	Varies	Vector ALU	28060903
0x0000A8	V_CMP_EQ_I32	vcc 0 v0	4	Vector ALU	7D040080
0x0000AC	V_CNDMASK_B32	v0 v5 v3 vcc	4	Vector ALU	00000705
0x0000B0	V_CNDMASK_B32	v3 v3 v5 vcc	4	Vector ALU	00060B03
0x0000B4	TBUFFER_STORE_FORMAT_X	v0 v1 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA41000 80010001
0x0000BC	TBUFFER_STORE_FORMAT_X	v3 v2 s[4:7] 0 offen format:[BUF_DATA_FORMAT_32BUF_NUM_FORMAT_FLOAT]	Varies	Vector Memory	EBA41000 80010302
0x0000C4	S_ENDPGM		1	Flow Control	BF810000

Figure 4.3: AMD CodeXL Assembly result of the compilation of the bitonic sort OpenCL code from Multi2Sim's benchmark, depicted in Figure 4.2. The figure shows the ISA instruction opcode and operands, as well as the resulting hexadecimal code.

Microblaze initially acts as the host processor, defining the data present in external memory. This includes not only the data to be processed, but the initialization data described in Sub-section 4.2.2 as well.

Afterwards, the processor operates as an ultra-threaded dispatcher, predefining the CU's instruction buffer with the given kernel binary, the scalar registers with pointers to the memory spaces set earlier, as well as the work-group ID, and the vector registers with the thread ID of each current thread.

After completing all initialization procedures, Microblaze signals NEKO to start execution. At the end of execution, every application writes-back the results to external memory. These are then validated using MATLAB.

According to the number of workers required to complete execution, a kernel may need multiple instantiations, since at most 64 threads can run simultaneously on the

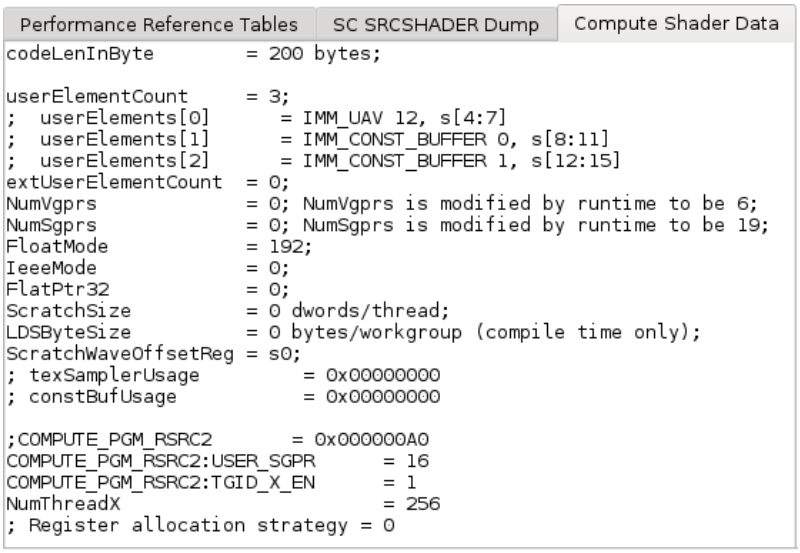


Figure 4.4: AMD CodeXL Pre-Initialized Registers. CodeXL details the pre-initialized registers and their usage. The three user elements present consist of three memory descriptors that have to be initialized. Furthermore, a few usage statistics are given, like the number of used VGPRs. To the end, there is indication that the first sixteen registers are initialized. Afterwards, there is a list of enabled flags, in this case only TGID_X_EN is set, which means that register number 16 has the workgroup ID in dimension X.

CU[17]. One such example is the multiplication of two matrices of dimension 64 by 64. Each worker computes a small four-by-four sub-matrix, therefore, 256 threads are required for the entire multiplication process. Since only 64 threads run simultaneously, the kernel must be executed four times. Prior to every run the Microblaze processor must reinitialize the register values, and update the thread IDs.

From the selected applications a small subset requires further processing from Microblaze, namely K-means clustering and Gaussian elimination. K-means is an iterative algorithm which partitions N observations into K clusters. Between iterations, Microblaze has to recompute the center of mass for each cluster. Gaussian elimination, on the other hand, only requires Microblaze to act after NEKO finishes. Initially, the CU puts the matrix in triangular form. Then, Microblaze performs the back-substitution to obtain the final result.

NEKO's execution time, for all applications, was measured using a cycle counter internal to the CU. Moreover, Microblaze's processing time, on K-means clustering and Gaussian elimination, was also quantified using the timer module[23] in the design.

To obtain a significant time profile, each application was tested with multiple problem sizes. The problem's dimension and execution time values are presented in the results chapter (Chapter 5). Furthermore, a comparison between the execution times of all developed systems is established to validate the improvements introduced in Section 4.1.

4.3 Application-Specific system development

To deal with current *field-programmable gate array* (FPGA) limits when trying to implement a system with the complexity of a compute unit, this work proposes the creation of application-specific cores. These cores discard all functionality that is not required by each specific application. By disabling non-necessary functionalities, a simpler core is obtained, with a reduced size, saving area (resources) on the board. This core has lower power requirements, since there are less board components to feed, which also results in a lower energy-consumption, since the removal of unused resources does not affect performance. The obtained core is, therefore, optimized in terms of area, power, and energy to each application.

The extra resources obtained from trimming down the core can be used to reestablish more execution units that had to be discarded initially (see Section 3.2), increasing the parallelism when executing instructions and, therefore, further improving throughput performance.

The developed cores are based in the applications selected to benchmark the CU, each being able to run one of the five programs used.

The CU is composed of eight major components, namely, fetching, decoding and scheduling units, the scalar and vector register files, and the execution units (LSU, *scalar ALU* (SALU), and VALU).

The units responsible for fetching and scheduling instructions—Fetch and Issue, respectively—do not alter their behavior for a given instruction. Furthermore, the register file units serve only as storage. Consequently, these units are not affected when developing the application-specific cores.

The Decode unit receives the fetched instruction and produces a number of control signals, including the execution unit selector and register addresses. For each instruction, the control output changes. Reducing the number of supported instructions simplifies the decode unit since part of the control circuit is eliminated.

Each execution unit performs a second instruction decode, which selects the correct operation to be executed. Furthermore, they perform complex operations—from memory accesses (LSU) to floating-point arithmetic (SALU and VALU)—requiring a great number of resources. The VALUs are also the modules that consume more resources in the design[12], hence, simplifying these units helps reducing the occupied area.

For each application, a list of used instructions is compiled—using CodeXL—, and the support for all others is removed. It is worth noticing that, for the applications which use only integer instructions, namely the integer matrix multiplication and the bitonic sort, the whole floating-point VALU is removed. This is a significant achievement since

this unit uses almost twice the resources of an integer VALU, being the single largest unit in the design[12].

The reduced cores are synthesized and their functionality is validated by running the corresponding benchmark applications and confirming the legitimacy of the results.

A comparison between the resulting resource usage and power requirements for each developed system is shown in Chapter 5. These values are also compared with those of the original system, prior to and following the improvements on throughput performance.

4.4 Summary

In this chapter, the improvements made to the original CU were described. These featured the addition of an interface for directly accessing the VGPR file, as well as two strategies to decrease memory access time, increasing throughput memory. The addition of a second clock domain improved the response time of the Microblaze processor, when performing memory accesses on behalf of the CU. Additionally, a BRAM module was instantiated inside NEKO. This block represents the first 2^{20} bytes in memory, and allows bypassing the Microblaze processor on the memory accesses corresponding to its address space.

Afterwards, the improvements made to the CU were quantified using benchmarks. A set of well known real-world applications were run on the developed systems (prior to and following the changes). Along with this time profiling, a detailed description of all initialization procedures is provided.

Finally, the focus was shifted to the development of application-specific cores. These consist of simplified CUs—with all non-required functionality removed—, which consume less resources and have lower power requirements.

The next chapter presents the experimental results obtained. First, a list of the synthesized ISA instructions—compiled using the comprehensive testing script developed in Chapter 3—is shown. Then, the results of the time profile made in this chapter are analyzed. Finally, a quantization of the savings provided by the application-specific cores is made by comparing them to the starting system.

5

Experimental Results

Contents

This chapter presents the experimental results obtained during the development of this thesis. In Chapter 3 a comprehensive script was conceived to evaluate which subset of instructions, from AMD's *instruction set architecture* (ISA)[17], are synthesized. The script's results are presented in Section 5.1.

Chapter 4 proposed changes to the *compute unit* (CU) intended to increase throughput performance. Section 5.2 quantifies the improvements made by comparing the execution times of all the developed systems on a set of five representative benchmarks, while also presenting the respective power requirements and resource usage. Also, Chapter 4 proposed the creation of application-specific CUs. A detailed analysis of area gains and power savings is made in 5.3.

5.1 Synthesized Instruction Set Architecture

Table 5.1 shows the subset of instructions, from AMD's ISA[17], currently synthesized in NEKO.

Table 5.1: Synthesized ISA.

Scalar Operations		Vector Operations	
Type	Instruction Name	Type	Instruction Name
SOP2	S_ADD_U32	VOP2	V_CNDMASK_B32
	S_SUB_U32		V_ADD_F32
	S_ADD_I32		V_SUB_F32
	S_SUB_I32		V_SUBREV_F32
	S_MIN_U32		V_MUL_F32
	S_MAX_I32		V_MUL_I32
	S_MAX_U32		V_MAX_F32
	S_CSELECT_B32		V_MAX_I32
	S_AND_B32		V_MIN_U32
	S_AND_B64		V_MAX_U32
	S_OR_B32		V_LSHRREV_B32
	S_OR_B64		V_ASHRREV_I32
	S_ANDN2_B64		V_LSHLREV_B32
	S_LSHL_B32		V_AND_B32
	S_LSHR_B32		V_OR_B32
	S_ASHR_I32		V_ADD_I32
	S_MUL_I32		V_SUB_I32
SOPK	S_MOVK_I32		V_SUBREV_I32
	S_ADDK_I32		V_ADDC_I32
	S_MULK_I32		V_RCP_F32

Continued on next page

Continued from previous page

Scalar Operations		Vector Operations	
Type	Instruction Name	Type	Instruction Name
SOP1	S_MOV_B32	VOP1	V_MOV_B32
	S_MOV_B64	VOPC	V_CMP_F_F32
	S_NOT_B32		V_CMP_LT_F32
	S_BREV_B32		V_CMP_EQ_F32
	S_AND_SAVEEXEC_B64		V_CMP_LE_F32
SOPC	S_CMP_EQ_I32		V_CMP_GT_F32
	S_CMP_LG_I32		V_CMP_LG_F32
	S_CMP_GT_I32		V_CMP_GE_F32
	S_CMP_GE_I32		V_CMP_NGE_F32
	S_CMP_LT_I32		V_CMP_NLG_F32
	S_CMP_LE_I32		V_CMP_NGT_F32
	S_CMP_EQ_U32		V_CMP_NLE_F32
	S_CMP_LG_U32		V_CMP_NEQ_F32
	S_CMP_GT_U32		V_CMP_NLT_F32
	S_CMP_GE_U32		V_CMP_TRU_F32
	S_CMP_LT_U32		V_CMP_F_I32
	S_CMP_LE_U32		V_CMP_LT_I32/U32
SOPP	S_ENDPGM		V_CMP_EQ_I32/U32
	S_BRANCH		V_CMP_LE_I32/U32
	S_BRANCH_SCC0		V_CMP_GT_I32/U32
	S_BRANCH_SCC1		V_CMP_LG_I32/U32
	S_BRANCH_EXECZ		V_CMP_GE_I32/U32
	S_BRANCH_EZECNZ		V_CMP_TRU_I32/U32
	S_BARRIER	VOP3a	All from VOPC
	S_WAITCNT		
SMRD	S_LOAD_DWORD		V_ADD_F32
	S_LOAD_DWORDX2		V_SUB_F32
	S_LOAD_DWORDX4		V_MUL_F32
	S_BUFFER_LOAD_DWORD		V_MAX_F32
	S_BUFFER_LOAD_DWORDX2		V_MAX_U32
			V_AND_B32
			V_BFE_U32
			V_BFE_I32
			V_BFI_B32
			V_MUL_LO_U32
			V_MUL_HI_U32
			V_MUL_LO_I32

Continued on next page

Continued from previous page

Scalar Operations		Vector Operations	
Type	Instruction Name	Type	Instruction Name
		Vector Mem.	TBUFFER_LOAD_FORMAT_X
			TBUFFER_LOAD_FORMAT_XY
			TBUFFER_LOAD_FORMAT_XYZ
			TBUFFER_LOAD_FORMAT_XYZW
			TBUFFER_STORE_FORMAT_X
			TBUFFER_STORE_FORMAT_XY
			TBUFFER_STORE_FORMAT_XYZ
			TBUFFER_STORE_FORMAT_XYZW

NEKO's ISA consists of 154 instructions. These perform a wide variety of operations, either bit-wise, arithmetical, or memory related. Using this ISA, a subset of five well known real-world applications could be executed in the CU. The benchmark programs were chosen from well established platforms, namely Multi2Sim[34] and Rodinia[35]. Three applications are provided by Multi2Sim's platform, notably bitonic sort, floating-point matrix multiplication, and integer matrix multiplication. The other two benchmarks used are K-means clustering and matrix Gaussian elimination, both from the Rodinia suite. Benchmarking these applications provided a comparison between the execution time of the original and improved systems.

5.2 Validation of dual-clock domain and BRAM usage

5.2.1 Benchmark results

Tables 5.2, 5.3, 5.4, 5.5, and 5.6 present the results for the benchmarks, applied to three different scenarios:

i) the original system;

ii) the system where a second clock domain was introduced;

iii) the system where a *block RAM* (BRAM) module was added to the compute unit.

Each application is benchmarked with different problem sizes, as indicated in the tables. The CU's execution times are provided by an internal cycle counter. For the subset of applications which require further processing, namely Gaussian elimination and K-means clustering, time measurements of Microblaze's execution are made using the timer module present in the system. Apart from the K-means clustering algorithm, which has an iterative nature, every benchmark has a fixed number of steps to completion.

Table 5.2: Results for integer matrix multiplication.

Matrix Size	Original system (μs)	System with dual clock domain (μs)	System with internal BRAM (μs)
4x4	0.402	0.329	0.048
8x8	1.945	1.585	0.093
16x16	13.133	10.689	0.261
32x32	98.160	79.903	1.239
64x64	761.698	620.313	9.603
128x128	5999.254	4886.868	75.604
256x256	47616.701	38792.411	599.949
512x512	380106.431	309129.876	4780.063

Table 5.3: Results for floating-point matrix multiplication.

Matrix Size	Original system (μs)	System with dual clock domain (μs)	System with internal BRAM (μs)
4x4	0.452	0.383	0.106
8x8	1.951	1.600	0.209
16x16	13.083	10.712	0.493
32x32	97.853	80.199	3.052
64x64	759.279	622.851	24.111
128x128	5980.112	4907.847	191.664
256x256	47469.927	38962.936	1528.430
512x512	378169.119	310504.064	12207.913

Table 5.4: Results for Gaussian elimination.

Matrix size	Original System (μs)		System with dual clock domain (μs)		System with internal BRAM (μs)	
	Compute Unit	Microblaze	Compute Unit	Microblaze	Compute Unit	Microblaze
4x4	1.253	0.182	1.026	0.0620	0.089	0.062
8x8	6.106	0.693	4.988	0.251	0.244	0.251
16x16	38.867	2.726	31.711	1.016	0.963	1.016
32x32	281.482	10.909	229.558	4.121	5.603	4.121

Table 5.5: Results for K-means clustering computation.

Problem size			Iterations to completion	Original System (μs)		System with dual clock domain (μs)		System with internal BRAM (μs)	
Num. of points	Features per point	Num. of clusters		Compute Unit	Microblaze	Compute Unit	Microblaze	Compute Unit	Microblaze
32	4	5	4	7.564	1.934	6.193	0.501	0.196	0.517
	8		4	14.827	3.875	12.133	1.003	0.365	1.019
	32		3	58.397	15.810	47.780	4.086	1.380	4.103
	64		2	116.492	32.660	95.309	8.432	2.734	8.449
	4	10	2	14.842	2.304	12.152	0.596	0.383	0.612
	8		2	29.367	4.675	24.033	1.208	0.721	1.224
	32		2	116.506	19.082	95.328	4.925	2.752	4.942
	64		2	232.696	39.146	190.385	10.095	5.459	10.112
64	4	5	3	14.948	3.535	12.235	0.916	0.310	0.949
	8		5	29.415	7.134	24.075	1.847	0.592	1.879
	32		5	116.218	28.890	95.115	7.470	2.281	7.503
	64		3	231.953	61.314	189.835	15.951	4.534	18.044
	4	10	3	29.433	3.903	24.095	1.011	0.610	1.044
	8		6	58.368	7.866	47.775	2.035	1.173	2.067
	32		3	231.972	32.533	189.855	8.401	4.552	8.434
	64		3	463.444	69.325	379.295	17.978	9.058	20.089
512	4	5	24	119.586	27.236	97.883	7.049	2.480	7.313
	8		15	235.322	58.011	192.603	15.072	4.732	17.446
	32		13	929.739	238.637	760.923	61.805	18.249	70.533
	64		19	1855.627	480.647	1518.683	124.598	36.272	141.773
	4	10	26	235.471	27.449	192.759	7.566	4.877	7.369
	8		18	466.943	58.582	382.199	15.250	9.383	17.597
	32		12	1855.805	241.795	1518.838	62.540	36.416	71.279
	64		13	3707.550	491.499	3034.388	127.243	72.461	144.422

Table 5.6: Results for bitonic sorting algorithm.

Array size	Original System (μs)	System with dual clock domain (μs)	System with internal BRAM (μs)
32	5.844	4.770	0.164
64	15.663	12.768	0.311
128	40.837	33.259	0.629
512	262.501	213.794	4.046
1024	641.677	522.604	9.891
2048	1540.259	1254.243	23.740

As such, for this specific program, the time per iteration is averaged and shown together with the required number of iterations. Each benchmark is repeated ten times, and the averaged results are presented.

As expected, increasing the problem size causes an increment in execution time. Furthermore, both modified systems present better results than the original one. The system with a dual clock domain always has slightly better throughput performance—compared to the original—, while the one containing an internal BRAM presents a substantial improvement to both.

The preceding results clearly identify memory accesses to be the main bottleneck in the original system, and even in the system with a dual clock domain. For instance, although floating-point operations take longer to complete than their integer counterparts—as an example, multiplication is 11x slower—, the original system benchmarks seem to indicate otherwise. Tables 5.2 and 5.3 show higher processing times for integer matrix multiplication, when compared to the floating-point scenario. This tendency is inverted when improvements are made to memory delay, and the difference in processing times is clearly visible in the system with a BRAM, where a multiplication of two 512x512 integer matrices takes almost three times less than the floating-point case.

The results in this sub-section are presented graphically in Appendix A.

5.2.2 Area and power analysis

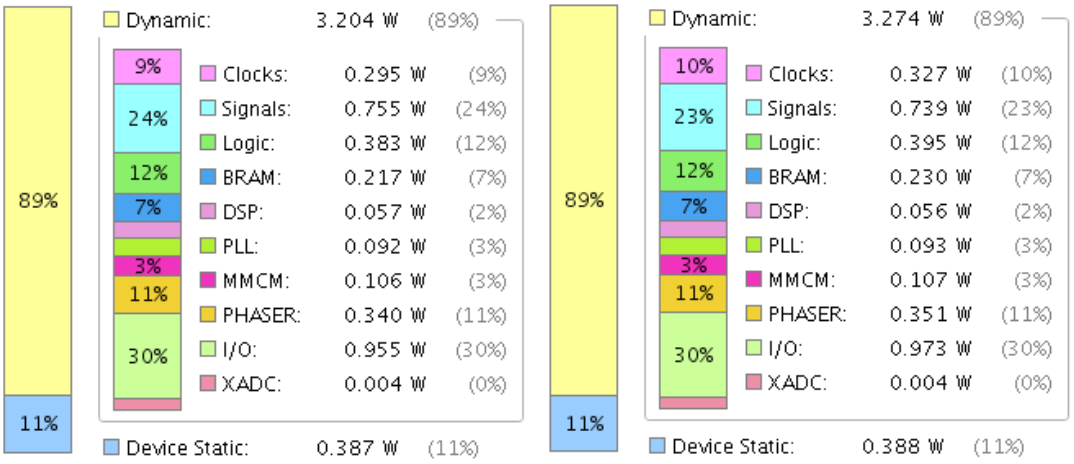
All the designs are synthesized using Xilinx's Vivado 2015.1, targeting Alpha Data's ADM-PCIE-7V3[1] board. After synthesis and implementation procedures, Vivado provides detailed statistics on resource usage and power requirements. Table 5.7 establishes a comparison between the resource usage in the original system and the ones with increased throughput.

Table 5.7: Comparison in resource utilization between the original system and the improved throughput performance systems, for Alpha Data's ADM-PCIE-7V3[1] board.

Resource	Original system	System with dual clock domain	System with dual clock domain & BRAM	Total Available Resources
	Utilization %	Utilization %	Utilization %	
FF	14.92	14.21	14.23	866400
LUT	48.71	47.54	48.75	433200
Memory LUT	1.90	1.26	1.26	174200
I/O	22.33	22.33	22.33	600
BRAM	15.14	15.14	78.27	1470
DSP48	5.64	5.50	5.50	3600
BUFG	56.25	56.25	56.25	32
MMCM	10.00	10.00	10.00	20
PLL	5.00	5.00	5.00	20

The BRAM instantiation can be clearly seen in Table 5.7, where the usage of this type of modules increases 63%. The remaining resources either remain unaltered or observe slight changes.

Figures 5.1a), 5.1b), and 5.2 show the detailed power distribution for all three systems.



(a) Original System **(b)** System with two clock domains

Figure 5.1: On-chip power distribution for the base system (a)) and the system with two clock domains (b)).

In Figure 5.1 it is shown that adding another clock domain to the system slightly increases the power requirement, mostly due to clock propagation and phase-shifting (PHASER) blocks. Adding the BRAM module to the system causes a significant increase in dissipated power of 10% (the difference in the total on-chip power between this system

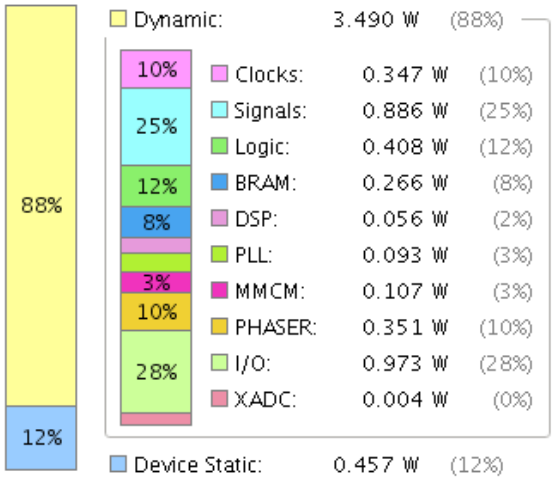


Figure 5.2: Power requirements for the system with two clock domains and a BRAM module.

and the original is 0.356W), as seen in Figure 5.2. However, the higher power requirement is compensated by the gain in throughput performance, which in the worst case is 76%, corresponding to the multiplication of two four-by-four floating-point matrices (see Table 5.3). The total energy (Energy=Power$\times$Time) consumed by each system on the respective benchmark is shown in graphical form in Appendix B.

5.3 Application-specific area gains and power savings

After development, the application-specific systems are synthesized for the Alpha Data ADM-PCIE-7V3[1] board, using Vivado 2015.1. This tool-chain provides detailed statistics on resource usage and power requirements, which are shown in the following subsections.

5.3.1 Power savings

All application-specific cores are derived from the system with a dual-clock domain and an internal BRAM. Figure 5.3 shows the total on-chip power requirements for the base and application-specific systems. Thus, a comparison of required power can be established. Figures 5.4a), 5.4b), 5.5a), 5.5b), and 5.6 detail the on-chip power distribution per each application-specific system.

Every application-specific system has a lower power requirement than the original, as depicted in Figure 5.3. Furthermore, in the worst case a 5.2% decrease in power rating is observed—from 3.947*W* to 3.743*W*, corresponding to the Gaussian elimination system, whereas the best scenario achieves a reduction in the power rating of 18%—from 3.947*W* to 3.246*W*, corresponding to the integer matrix multiplication system. The best scenar-

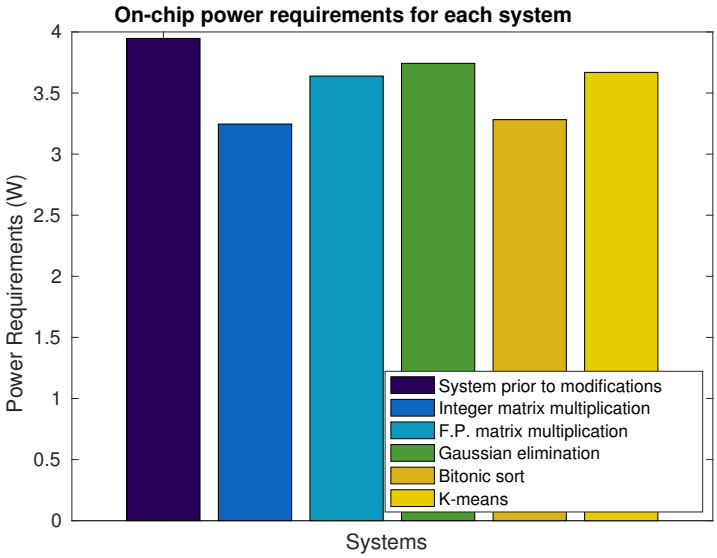
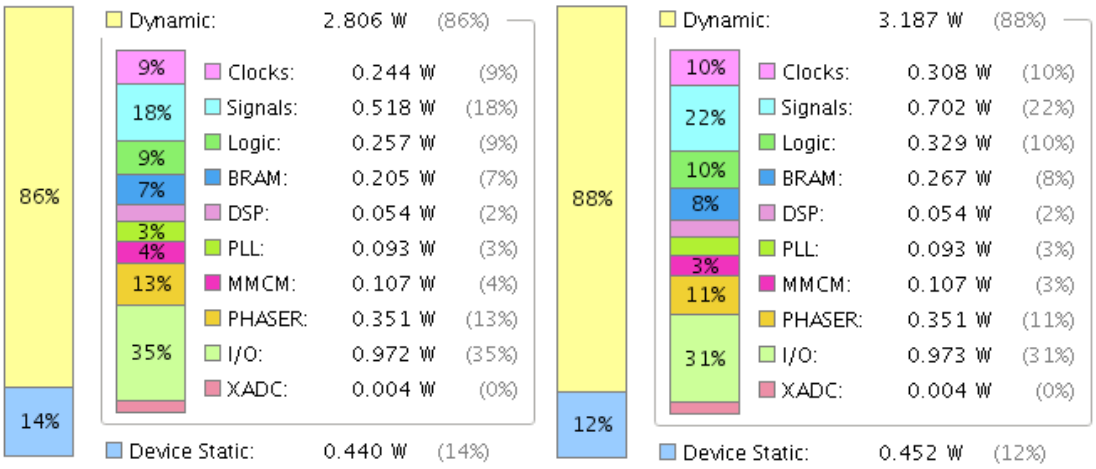
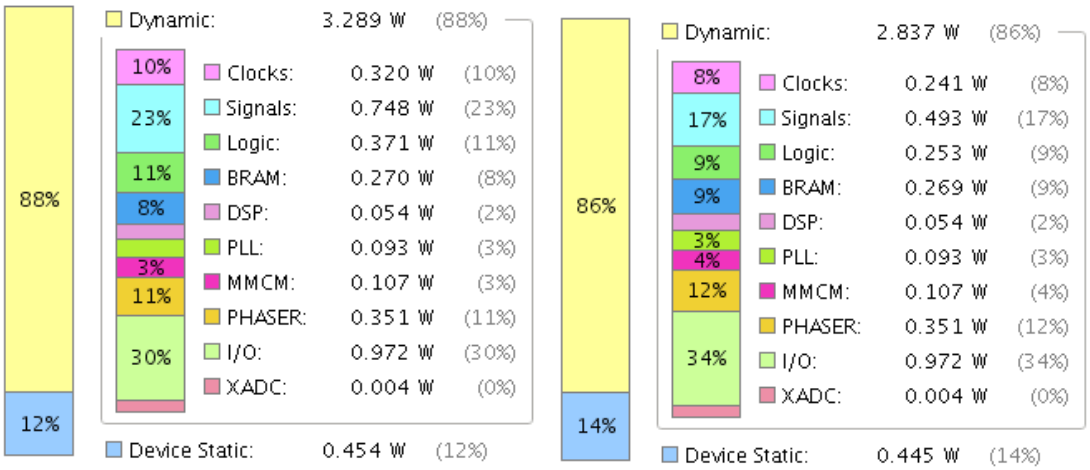


Figure 5.3: Total on-chip power per system. The first bar establishes the baseline, presenting the energy consumption of the system prior to the architectural trim-down. The latter five bars represent the power requirements for the application-specific systems.



(a) Integer matrix multiplication system **(b)** F.P. matrix multiplication system

Figure 5.4: Power requirements for the integer (a)) and floating-point (b)) matrix multiplication systems.



(a) Gaussian elimination system **(b)** Bitonic sort system

Figure 5.5: Power requirements for the Gassian elimination (a)) and bitonic sort (b)) systems.

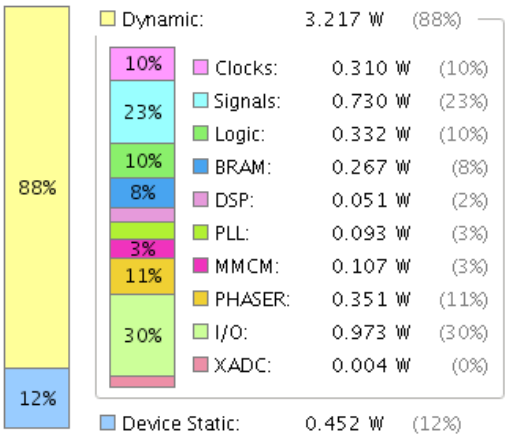


Figure 5.6: Power requirements for the K-means clustering system.

ios, namely integer matrix multiplication and bitonic sort, correspond to the cases where there is no need for the floating-point *vector ALU* (VALU), which greatly decreases the power requirements. The remaining cases also see a reduction in the requirement, although not as expressive, due to the different functionalities of each application. The best floating-point system is the matrix multiplication one with a reduction of 7.8% in the power requirement. The difference in the power requirement between all the systems shows that simply removing an entire unit significantly decreases this rating. However, even in the cases where this does not occur (floating-point systems), a reduction in the requirement is also attainable and, in the studied systems, it can go up to 7.8%. Since throughput performance is not affected by the application-specific optimizations of the CU, then the energy expenditure also decreases. The total energy consumed by each application-specific system is shown in graphical form in Appendix B.

In the remaining figures (5.4a), 5.4b), 5.5a), 5.5b), and 5.6), a detailed distribution of on-chip power is shown. These clearly identify clocks, signals, and logic as the most affected blocks by the changes proposed. On the other hand, static power only decreases on the integer matrix multiplication and bitonic sort systems, due to the lower number of modules that need powering, since it represents the power drawn by the device when it is powered up.

Digital signal processor (DSP)'s stasis could seem strange, since they can be used to perform decimal operations, and some systems completely remove all floating-point capability. However, the initial system did not, directly, instantiate DSP blocks, with floating-point operations being implemented in Verilog instead. This is due to the original design team's intention of remaining vendor-neutral[12].

5.3.2 Area gains

Figure 5.7 presents the resource usage for each application-specific system in a bar chart. Reference values are also shown, corresponding to the data from the system prior to the architectural trim-down. Resource usage is defined regarding *flip-flops* (FFs), *lookup-tables* (LUTs), Memory LUTs, I/Os, BRAMs, DSPs, *global buffers* (BUFGs), *mixed-mode clock managers* (MMCMs), and *phase locked loops* (PLLs).

Across all application-specific systems a decrease in the usage of FFs, LUTs, and DSPs can be observed. Memory LUTs, I/Os, MMCMs, and PLLs remain constant throughout all systems, while BUFG reduces to less than half on the benchmarks without floating-point operations. BRAM usage only sees a decrease in the integer matrix multiplication system.

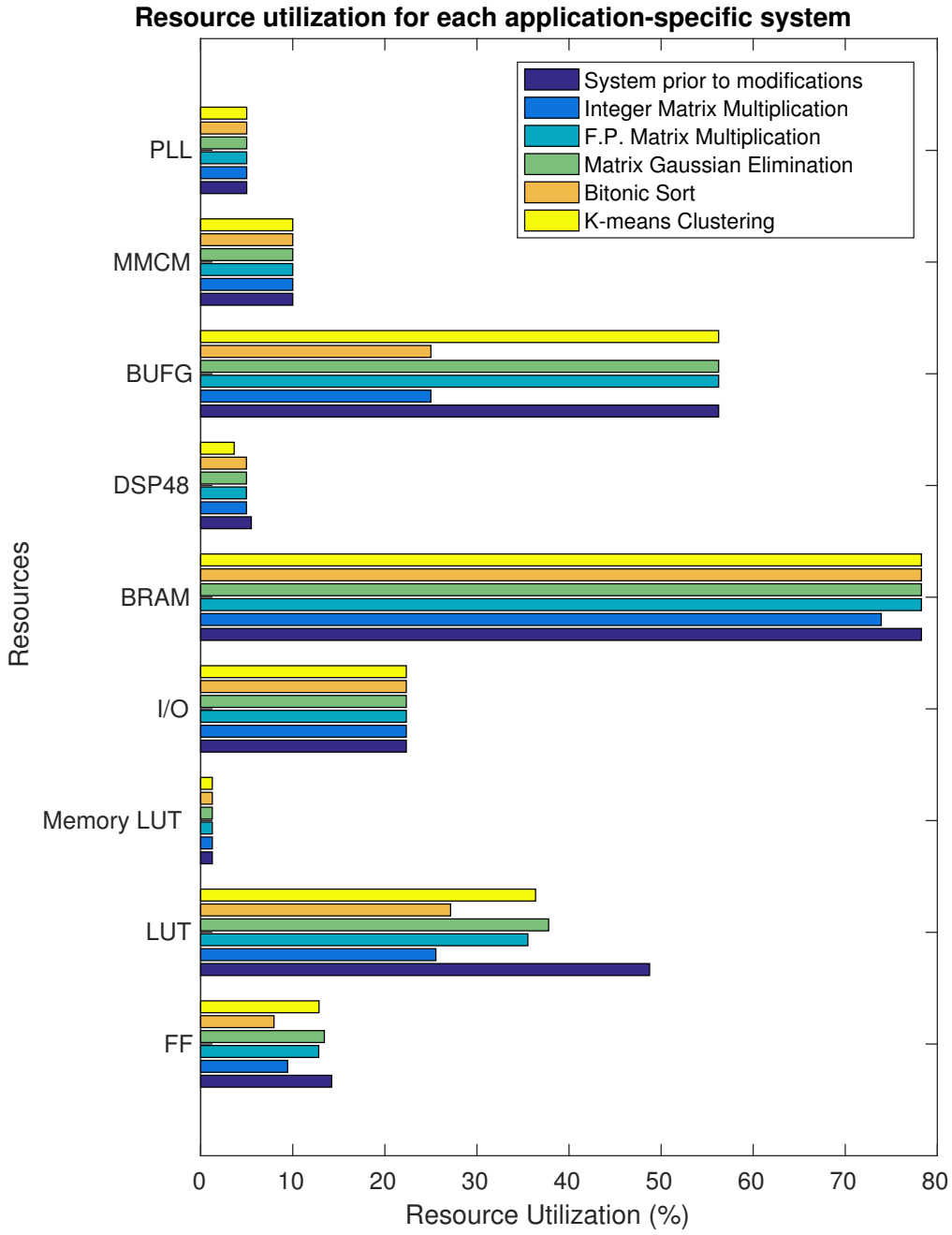


Figure 5.7: Comparison in resource utilization between the system prior to the architectural trim-down and all the application-specific systems, for Alpha Data's ADM-PCIE-7V3[1] board.

Every resource, except the BRAM and BUFG, has a usage rate below 50%. However, Table 5.7 shows that BRAM utilization can vary from the current 78.27% to a minimum of 15.14%, as defined by the system designer.

BUFG utilization decreases 31.25% for the benchmarks that do not require a floating-point unit—integer matrix multiplication and bitonic sort. Since the maximum usage for this resource is 56.25%, at least one more vector execution unit can be instantiated in all the application-specific cores and, in the integer-only systems, at least two more units can be added.

Another possible exploration is adding a new CU to the system, which is only possible in the integer matrix multiplication and bitonic sort systems, and if the size of the BRAM module instantiated is decreased.

5.4 Summary

In this chapter the experimental results obtained were presented. First, the complete listing of instructions from the synthesized ISA was provided, as a result from the comprehensive testing script developed. Then, benchmark results were shown, and the changes proposed in Chapter 4 to increase throughput performance were validated and quantified. Finally, the analysis of area and power savings for the proposed application-specific systems was made.

The next chapter presents the conclusions and future work directions.

6

Conclusions

Contents

With the increasing demand in performance requirements of modern scientific applications, two approaches have been widely used. In particular, the focus has been split between using *general-purpose GPUs* (GPGPUs) or *field-programmable gate arrays* (FPGAs) for accelerating data processing. Although *graphics processing unit* (GPU)'s massively parallel architectures provide high flexibility and performance, off-the-shelf devices have fixed designs that cannot be adapted towards the specific characteristics of the target applications. On the other hand, application-specific architectures can be designed and implemented in FPGAs that can be easily tailored to maximize the performance of a given application. However, such approaches often require a profound architectural redesign at the presence of minimal algorithmic changes. To overcome both issues, soft-GPGPUs arised as a solution that unites general-purpose massively parallel and programmable architectures (GPGPUs) with reconfigurable fabric (FPGAs). Designing this kind of system is a complex task and developers have overlooked the effect of extra functionalities not required by the targeted application, in terms of saving resource usage and energy consumption. Thus, the main goal of this thesis was to develop application-specific GPGPU cores, that feature all the required functionalities for a given application and are stripped of all others. Achieving the proposed objectives resulted in four major contributions.

A Microblaze-based system was designed and implemented on an Alpha Data ADM-PCIE-7V3[1] board, featuring a Xilinx Virtex-7 XC7VX690T FPGA. Since this is not a development board from Xilinx, there are no example projects easily available, and the system must be designed from scratch. This work provided a detailed explanation on the design procedures for non-development boards.

Moreover, while testing the *compute unit* (CU), a thorough validation script was created, identifying the complete synthesized *instruction set architecture* (ISA)—which is a subset of AMD Southern Islands, which aided in the addition of new instructions. Currently, a total of 154 instructions are synthesized and correctly implemented in the system, including vector (104) and memory (8) functionalities. Which is the result of an increase of 112 in the number of supported instructions.

Changes were implemented targeting an increase in throughput performance of the system, by reducing memory access delays, and validated using a timing profile of the GPGPU cores. The timing profile used five different benchmarks from well established platforms, namely Rodinia[35]—K-means clustering and matrix Gaussian elimination—and Multi2Sim[34]—bitonic sort, floating-point matrix multiplication, and integer matrix multiplication. The changes made, namely accelerating the memory access by increasing the clock frequency of both the processor (Microblaze) and the memory controller (MIG) by adding a second clock domain, and introducing a *block RAM* (BRAM) module to the CU to bypass the Microblaze in the memory accesses, resulted in a reduction of execution

time of up to 79.5x, namely when multiplying two 512x512 integer matrices. With this decrease in the execution time the system became more energy-efficient as a reduction of up to 72x was achieved, for the same application as before.

Finally, the application-specific cores initially proposed were developed, and the corresponding power savings showed a reduction of, up to, 18% of the base system's initial rating, which directly translates in equivalent energy savings, since the throughput performance remains unaltered. Moreover, the decrease in resource usage proved that the number of execution units (*vector ALUs* (VALUs)) in all the application-specific systems can be increased, which can would result in more instruction-level parallelism, and, in the integer matrix multiplication and bitonic sort cases, a new CU can be instantiated, doubling the available computational power.

These application-specific cores can be used in both ends of the technology spectrum. If applied to low-power devices, focused on mobile or *Internet of things* (IoT) applications like artificial intelligence, they can be used as accelerators with minimal power and resource requirements, providing energy savings. On the other hand, they can be used in larger boards, providing an efficient resource usage, which can be used to maximize parallelism for demanding applications like neural networks.

6.1 Future work

Although adding the dual-clock domain into the system and an internal BRAM module to the CU helped increase throughput performance, further improvements can be attained by exploring the newly available resources resulting from the architectural trimdown. Thus, a trade-off between the number of vector execution units (floating-point and integer), the number of CU, power requirements, and throughput performance can be studied.

Instantiating more vector execution units could increase throughput performance if the application has multiple independent computation instructions near each other, as they could be done in parallel. Adding a new CU not only doubles the computational resources available but it also increases the necessary control logic. The Microblaze processor would have to answer memory requests from both units, and control their kernel execution. Furthermore, a trade-off between the internal BRAM size and the number of compute units in the same design has to be made.

These explorations will lead to new time profiles, namely by repeating the benchmarks made in this work and by developing new ones.

Moreover, the BRAM currently works as a replacement for the first 2^{20} bytes in memory. Different approaches can be made, for instance setting this module as a cache level.

This would require altering the memory controller, creating a structure that would implement a cache scheme, in hardware.

Finally, memory access times could be improved by removing Microblaze as an intermediary for requests, allowing the CU to directly interact with the *memory interface generator* (MIG) controller.

Bibliography

[1] Alpha Data, "ADM-PCIE-7V3 User Manual," 2016. [Online]. Available: http://www.alpha-data.com/pdfs/adm-pcie-7v3%20user%20manual.pdf

[2] R. Pepper and J. Garrity, "Global Information Technology Report," pp. 35–42, 2014. [Online]. Available: http://blogs.cisco.com/wp-content/uploads/GITR-2014-Cisco-Chapter.pdf

[3] N. Corporation, "Whitepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™," 2009. [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[4] A. Bourd, "The OpenCL Specification: Version 2.2," March 2016. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf

[5] NVIDIA, "NVIDIA Jetson TK1 Development Kit: Bringing GPU-accelerated computing to Embedded Systems (Technical Brief v1.0)," NVIDIA Corporation, Tech. Rep., April 2014.

[6] R. Mijat, "Take GPU Processing Power Beyond Graphics with Mali GPU Computing," ARM Limited, Tech. Rep., Aug 2012. [Online]. Available: http://malideveloper.arm.com/downloads/WhitePaper_GPU_Computing_on_Mali.pdf

[7] S. Ma, M. Huang, E. Cartwright, and D. Andrews, "Scalable memory hierarchies for embedded many-core systems," in *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 151–162. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28365-9_13

[8] Xilinx Inc., "The Xilinx SDAccel Development Environment." [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf

[9] Altera Corp., "Altera SDK for OpenCL. Programming Guide," 2014.

[10] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, March 2015.

[11] K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft gpgpu for fpgas," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 230–237.

[12] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling gpgpu low-level hardware explorations with miaow: An open-source rtl implementation of a gpgpu," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 21:21:1–21:21:25, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764908

[13] M. Al Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-Architecture for FPGAs," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, 2016, pp. 254–263.

[14] A. Severance and G. Lemieux, "Venice: A compact vector processor for fpga applications," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 261–268.

[15] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "Vegas: Soft vector processor with scratchpad memory," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950420

[16] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W. F. Wong, "Guppy: A gpu-like soft-core processor," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 57–60.

[17] AMD, "Southern Islands Series Instruction Set Architecture Reference Guide," Dec. 2012.

[18] Xilinx, "7 Series FPGAs Overview," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[19] Xilinx, "Vivado Design Suite User Guide: Getting Started," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug910-vivado-getting-started.pdf

[20] Xilinx, "Vivado Design Suite Tutorial: Embedded Processor Hardware Design," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug940-vivado-tutorial-embedded-design.pdf

[21] Xilinx, "MicroBlaze Soft Processor Core ," 2015. [Online]. Available: http://www.xilinx.com/products/design-tools/microblaze.html

[22] Xilinx, "Zynq-7000 All Programmable SoC and 7 Series Devices Memory Interface Solutions v2.3 User Guide," Jun. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v2_3/ug586_7Series_MIS.pdf

[23] Xilinx, "AXI Timer v2.0 LogiCORE IP Product Guide," Nov. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf

[24] Xilinx, "MicroBlaze Debug Module (MDM) v3.2 LogiCORE IP Product Guide," Nov. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf

[25] Xilinx, "Vivado Design Suite: AXI Reference Guide," Jun. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[26] Xilinx, "Vivado IP Integrator," 2013. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/vivado/Vivado_IP_Integrator_Backgrounder.pdf

[27] Xilinx, "Vivado Design Suite Tutorial: Creating and Packaging Custom IP," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf

[28] Xilinx, "Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug995-vivado-ip-subsystems-tutorial.pdf

[29] Xilinx, "Clocking Wizard v5.1 LogiCORE IP Product Guide," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_1/pg065-clk-wiz.pdf

[30] Xilinx, "Getting Started with Xilinx SDK," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/SDK_Doc/index.html

[31] Xilinx, "Vivado Design Suite Tutorial: Logic Simulation," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug937-vivado-design-suite-simulation-tutorial.pdf

[32] Xilinx, "Vivado Design Suite User Guide: Logic Simulation," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug900-vivado-logic-simulation.pdf

[33] Xilinx, "Vivado Design Suite User Guide: Using Tcl Scripting," Apr. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug894-vivado-tcl-scripting.pdf

[34] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[36] AMD, "Getting Started with CodeXL," 2012. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/CodeXL_Quick_Start_Guide.pdf

[37] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[38] Edwards, A., "Getting Started with Radeon Open Compute Platform (ROCm)," apr 2016. [Online]. Available: http://gpuopen.com/getting-started-with-boltzmann-components-platforms-installation/

A

Plots of benchmark results

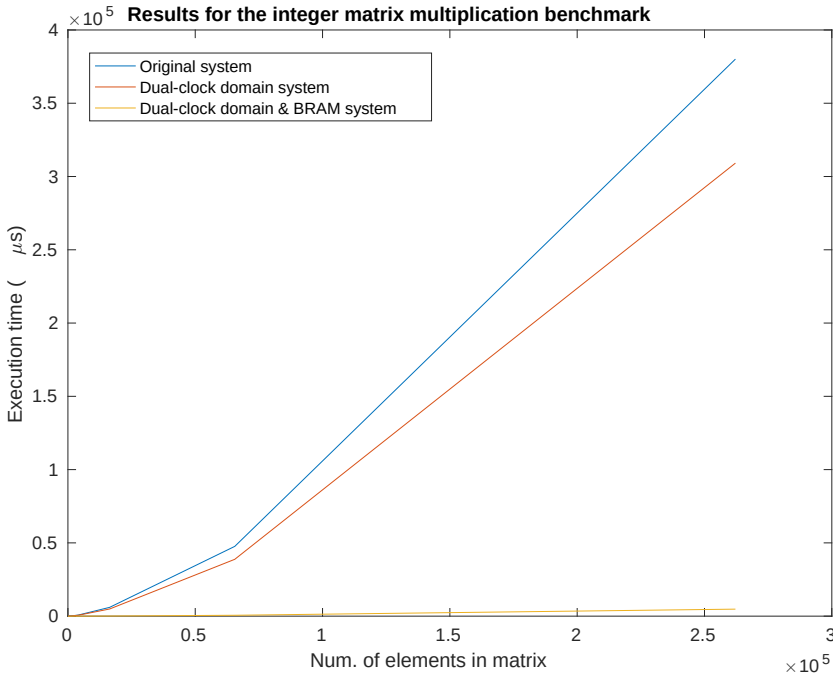


Figure A.1: Integer matrix multiplication results

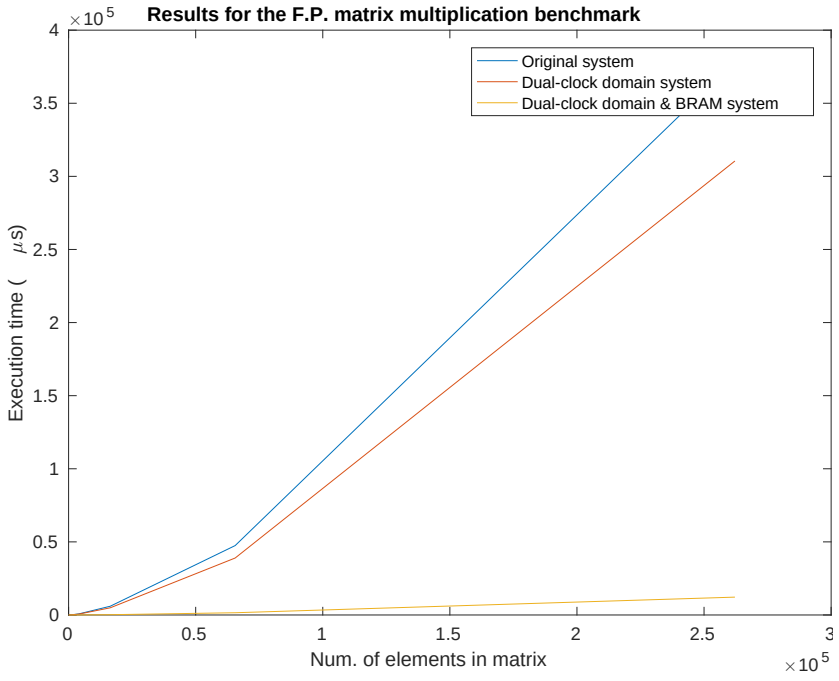


Figure A.2: Floating-point matrix multiplication results

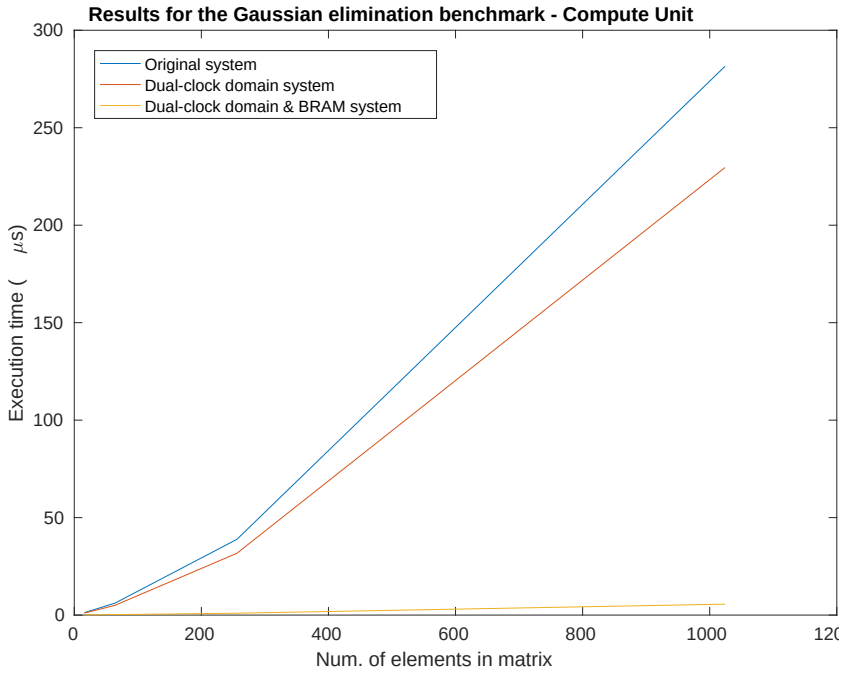


Figure A.3: Matrix Gaussian elimination results - CU

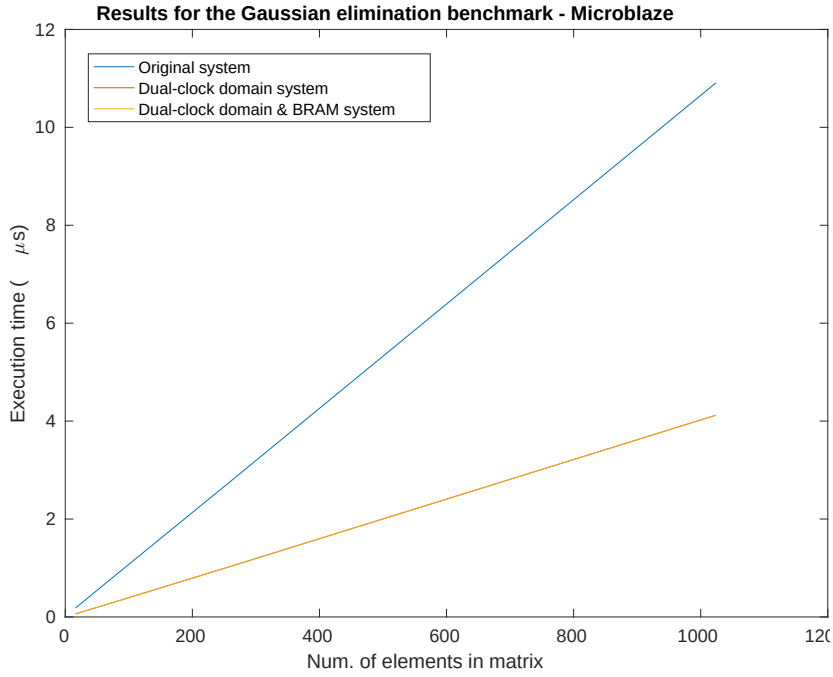


Figure A.4: Matrix Gaussian elimination results - Microblaze

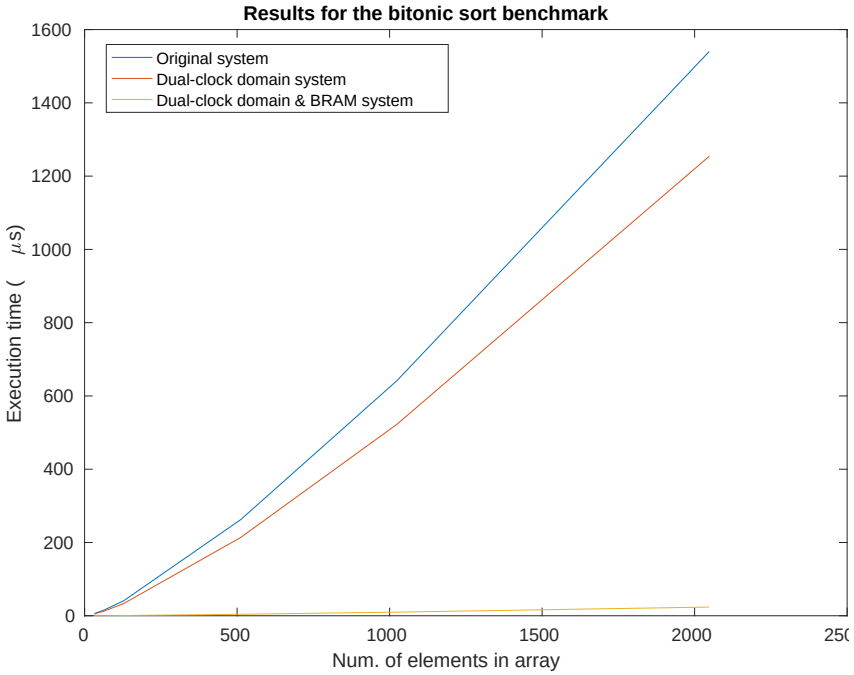


Figure A.5: Bitonic sort results

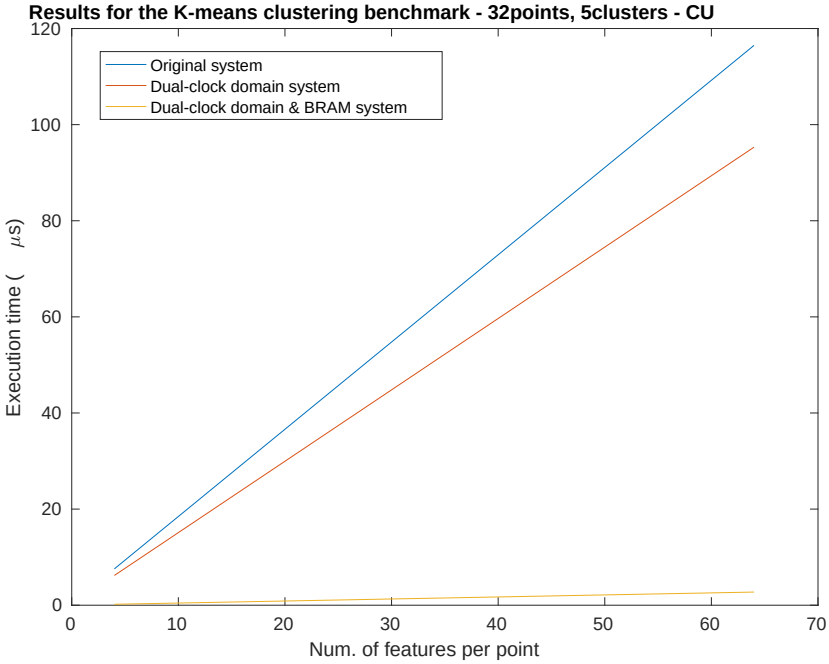


Figure A.6: K-means clustering benchmark results - 32points, 5clusters - CU

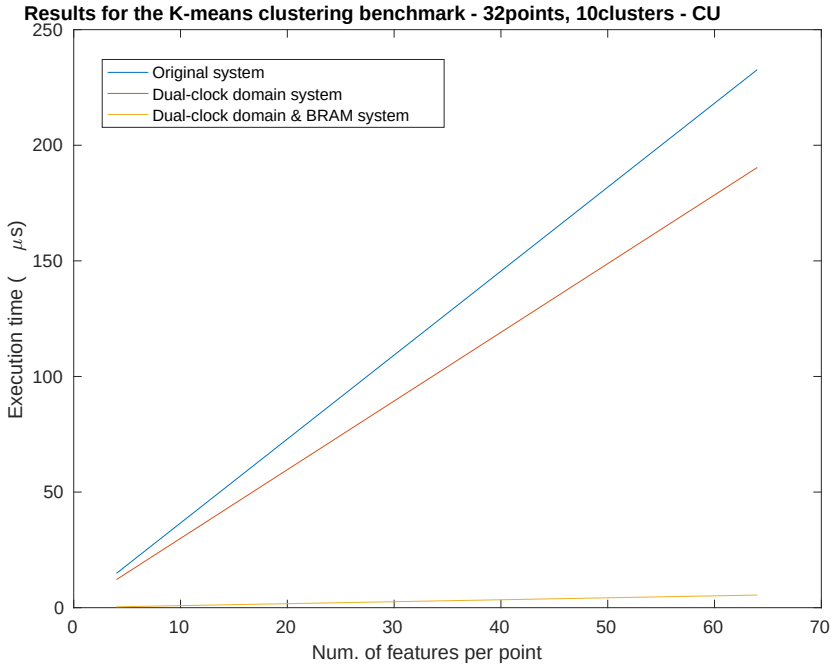


Figure A.7: K-means clustering benchmark results - 32points, 10clusters - CU

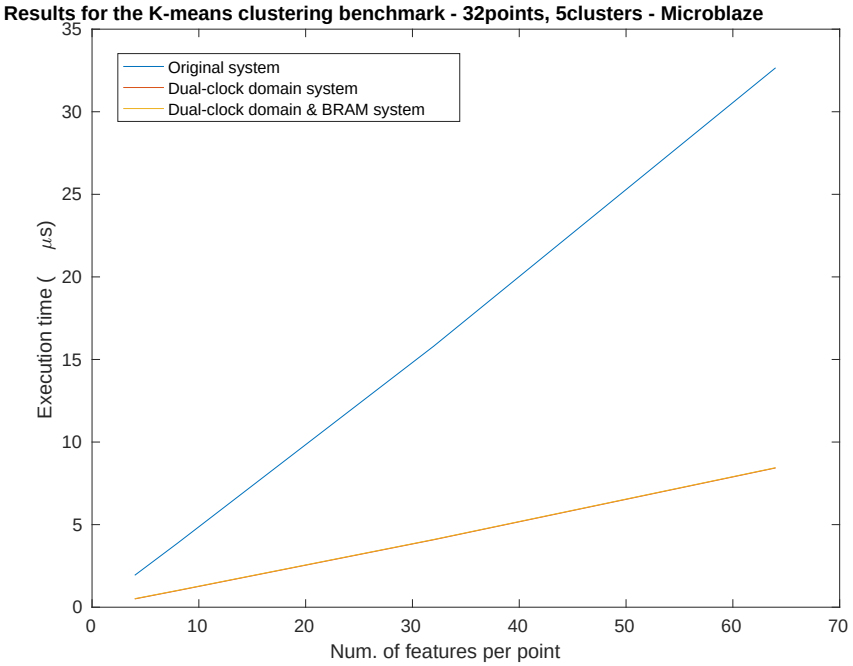


Figure A.8: K-means clustering benchmark results - 32points, 5clusters - Microblaze

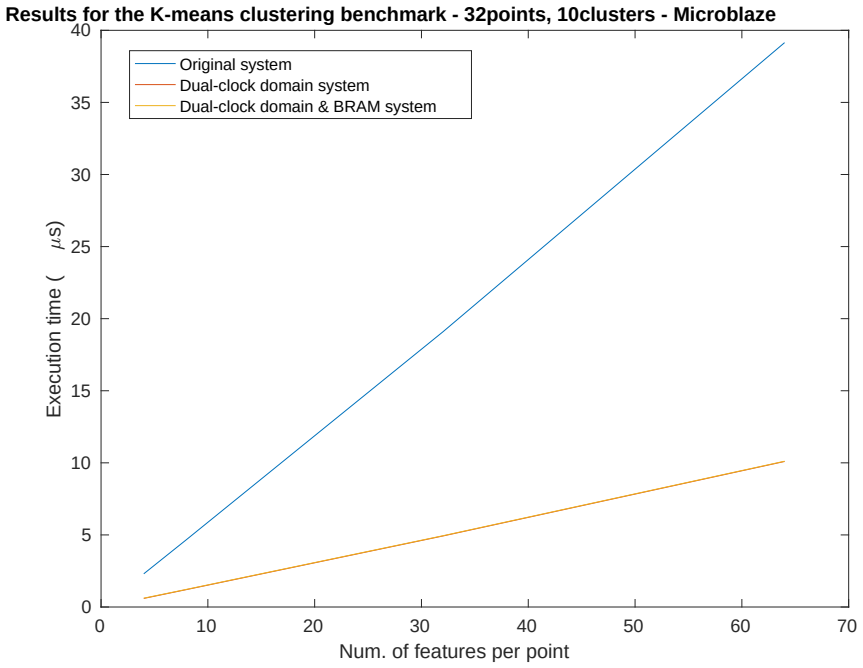


Figure A.9: K-means clustering benchmark results - 32points, 10clusters - Microblaze

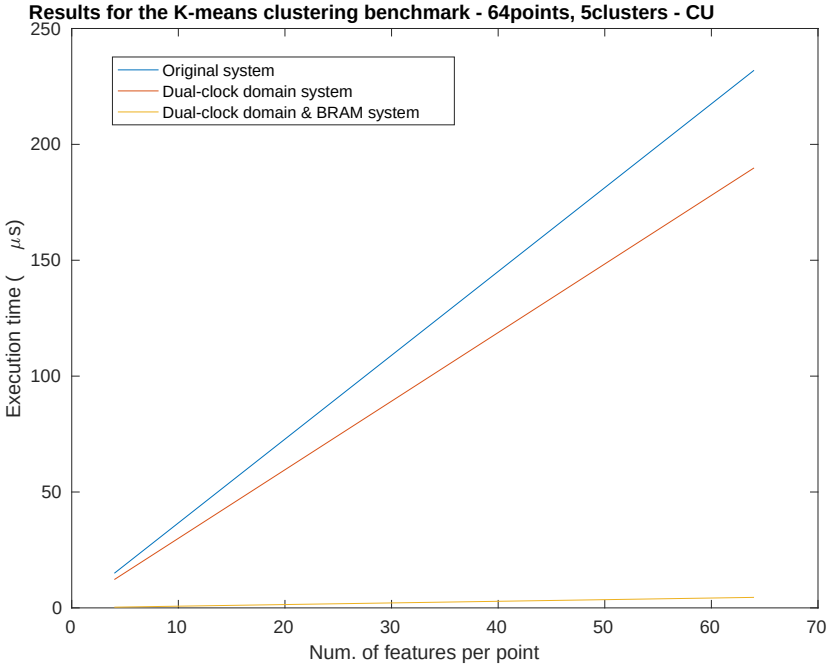


Figure A.10: K-means clustering benchmark results - 64points, 5clusters - CU

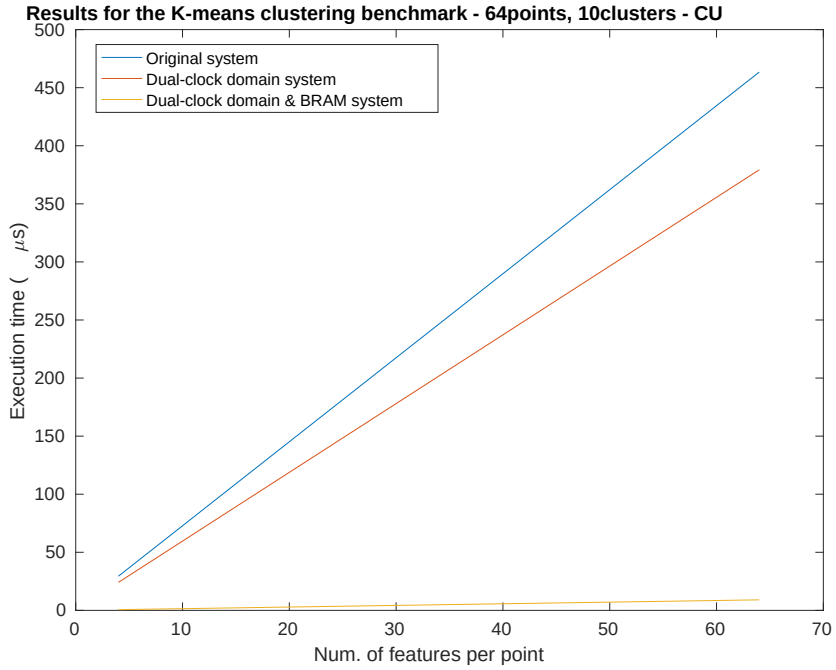


Figure A.11: K-means clustering benchmark results - 64points, 10clusters - CU

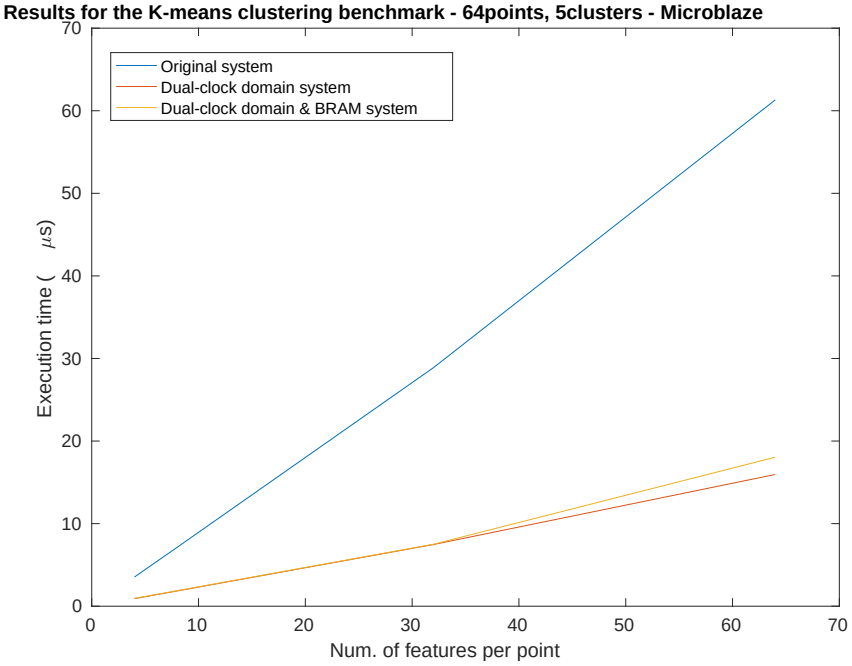


Figure A.12: K-means clustering benchmark results - 64points, 5clusters - Microblaze

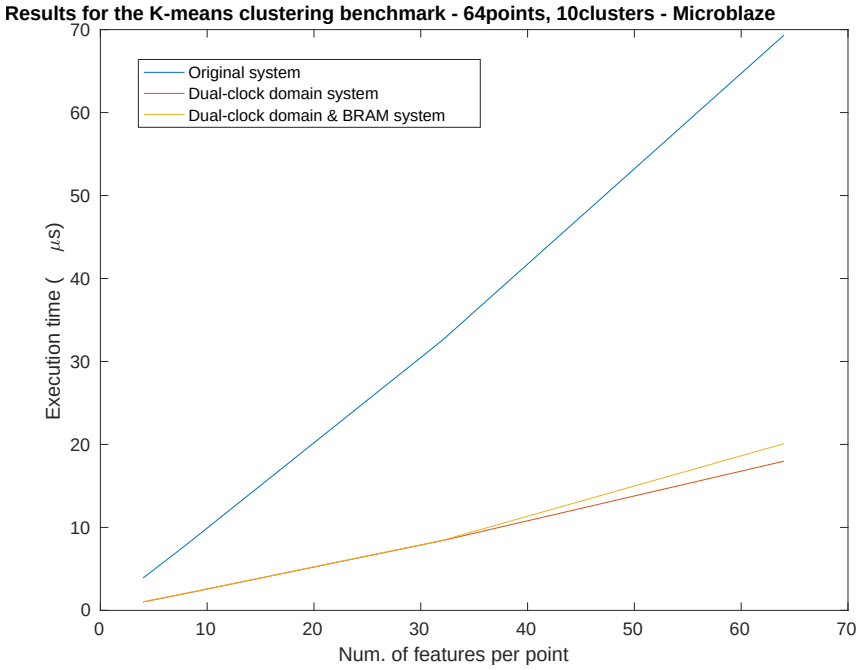


Figure A.13: K-means clustering benchmark results - 64points, 10clusters - Microblaze

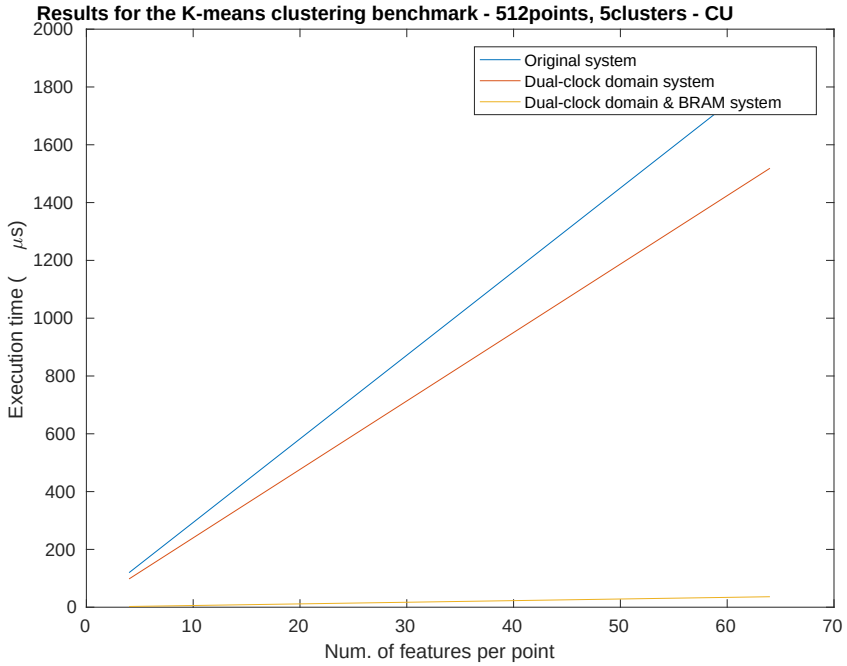


Figure A.14: K-means clustering benchmark results - 512points, 5clusters - CU

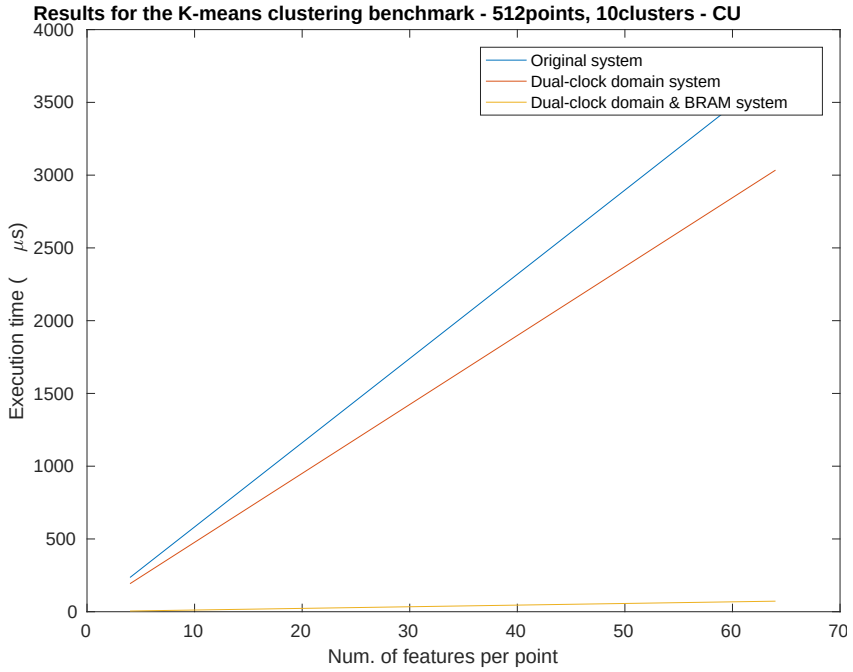


Figure A.15: K-means clustering benchmark results - 512points, 10clusters - CU

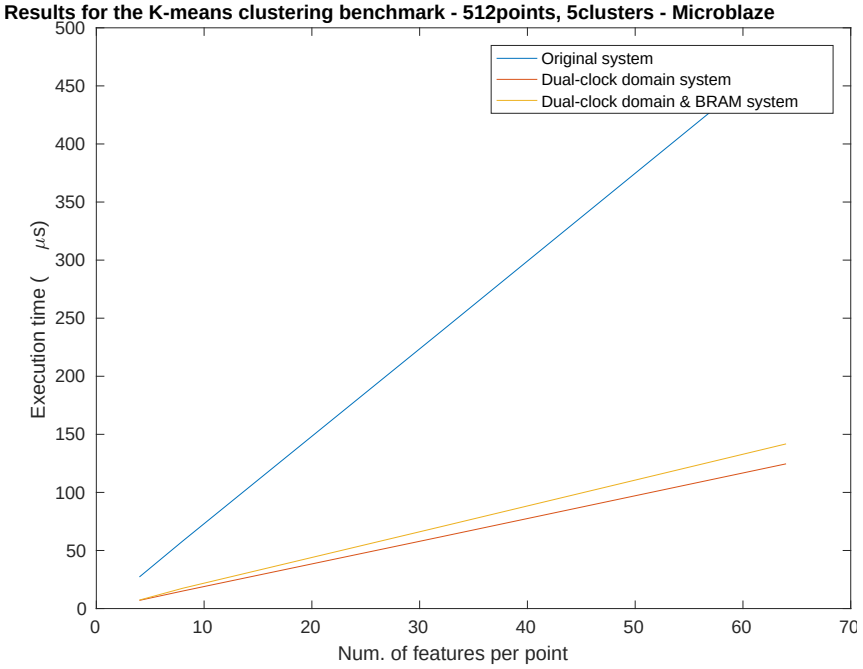


Figure A.16: K-means clustering benchmark results - 512points, 5clusters - Microblaze

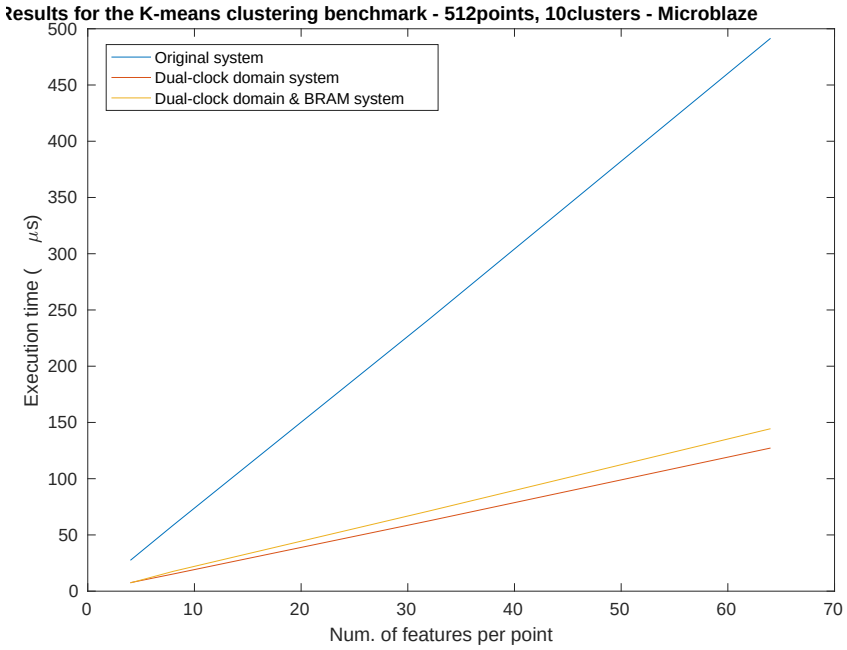


Figure A.17: K-means clustering benchmark results - 512points, 10clusters - Microblaze

B

Energy consumption results

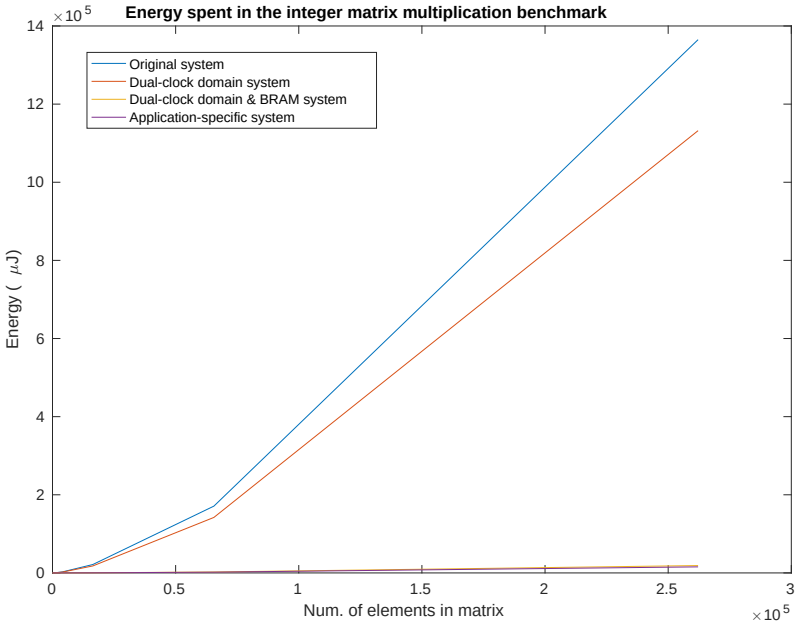


Figure B.1: Energy spent in the integer matrix multiplication benchmark

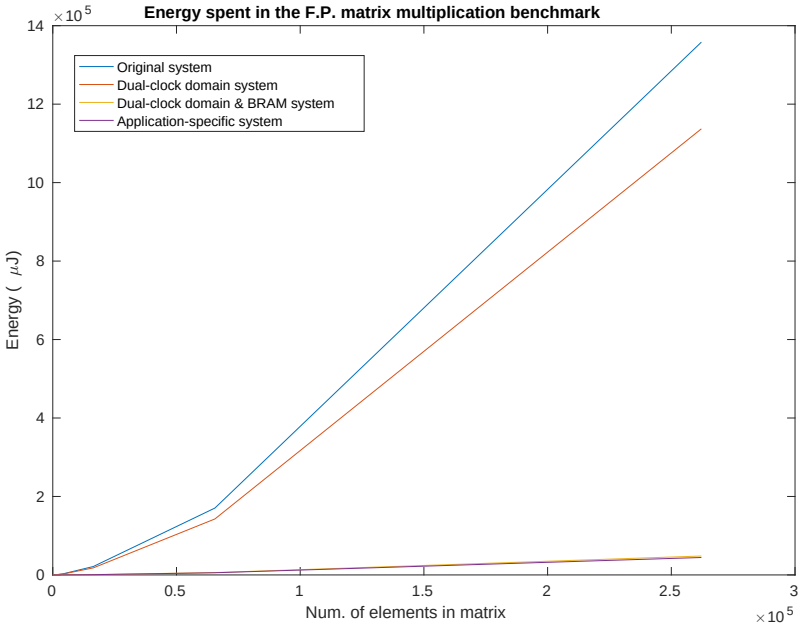


Figure B.2: Energy spent in the floating-point matrix multiplication benchmark

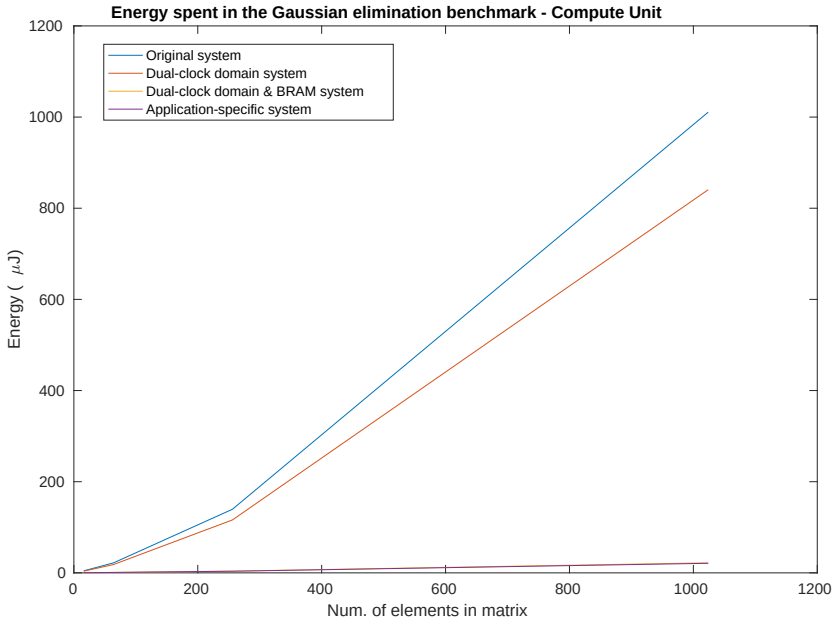


Figure B.3: Energy spent in the Gaussian elimination benchmark - Compute Unit (CU)

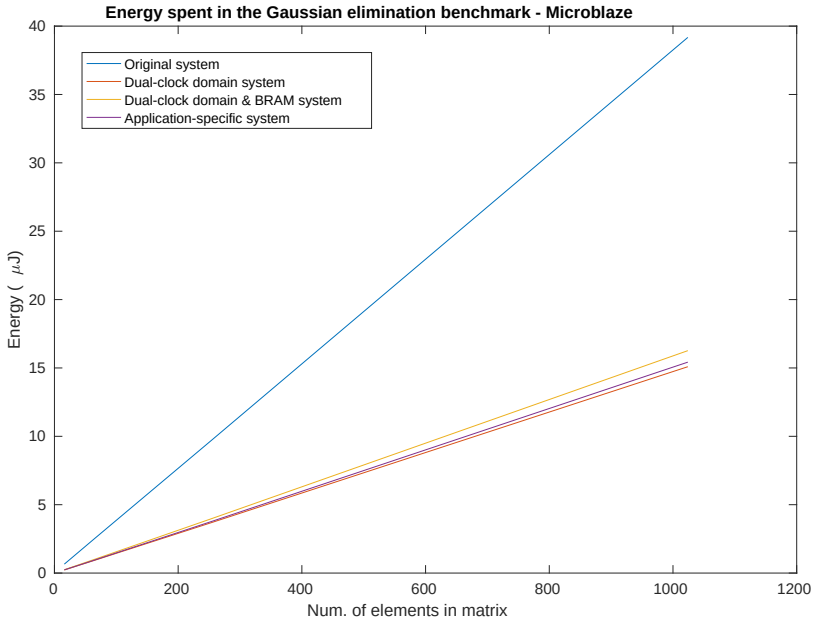


Figure B.4: Energy spent in the Gaussian elimination benchmark - Microblaze

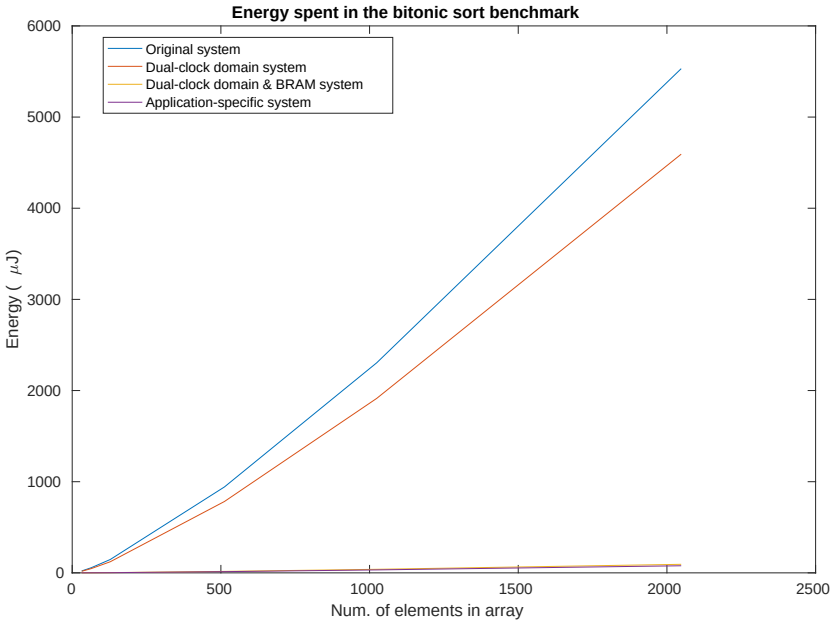


Figure B.5: Energy spent in the bitonic sort benchmark

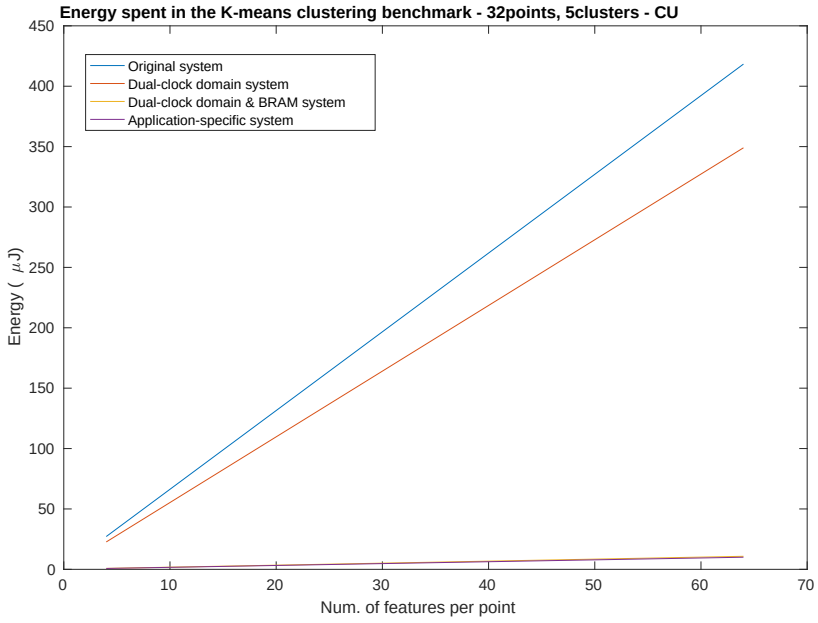


Figure B.6: Energy spent in the K-means clustering benchmark - 32points, 5clusters - CU

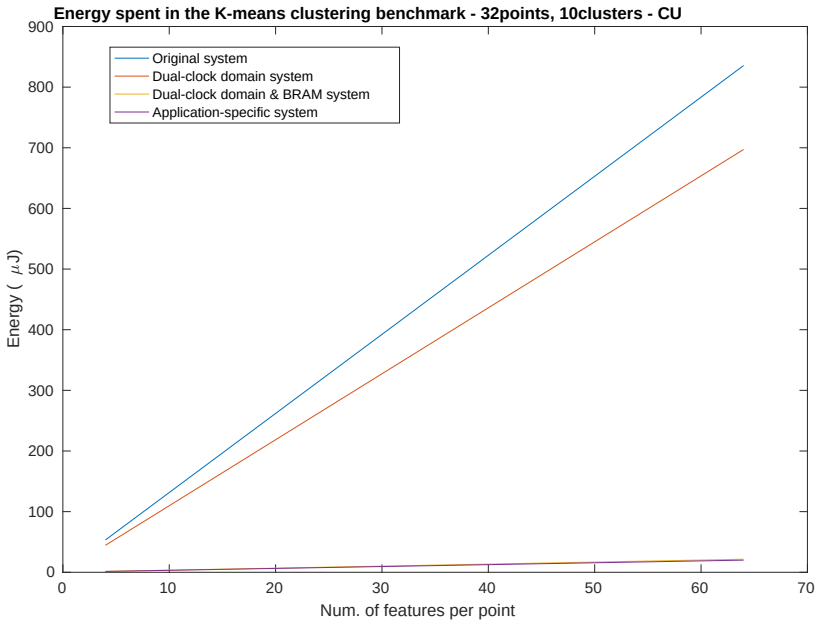


Figure B.7: Energy spent in the K-means clustering benchmark - 32points, 10clusters - CU

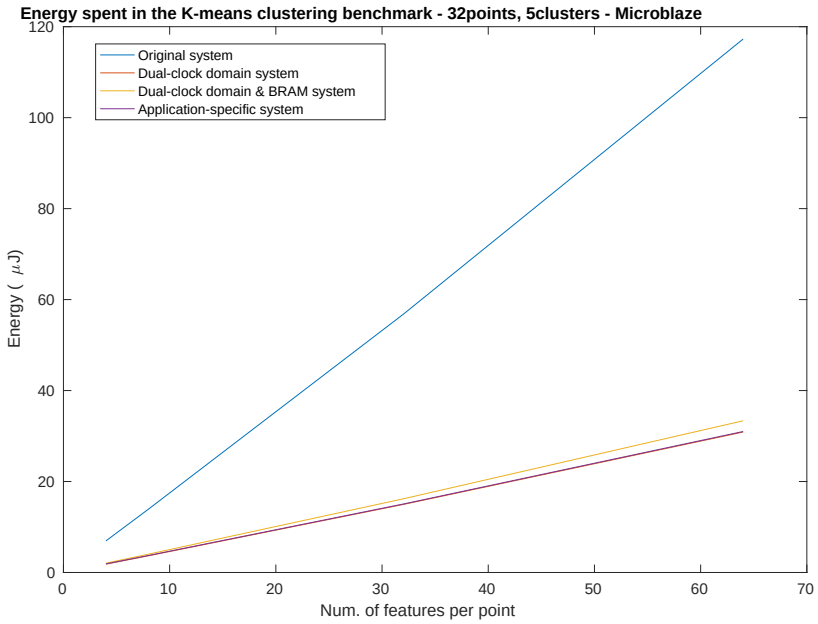


Figure B.8: Energy spent in the K-means clustering benchmark - 32points, 5clusters - Microblaze

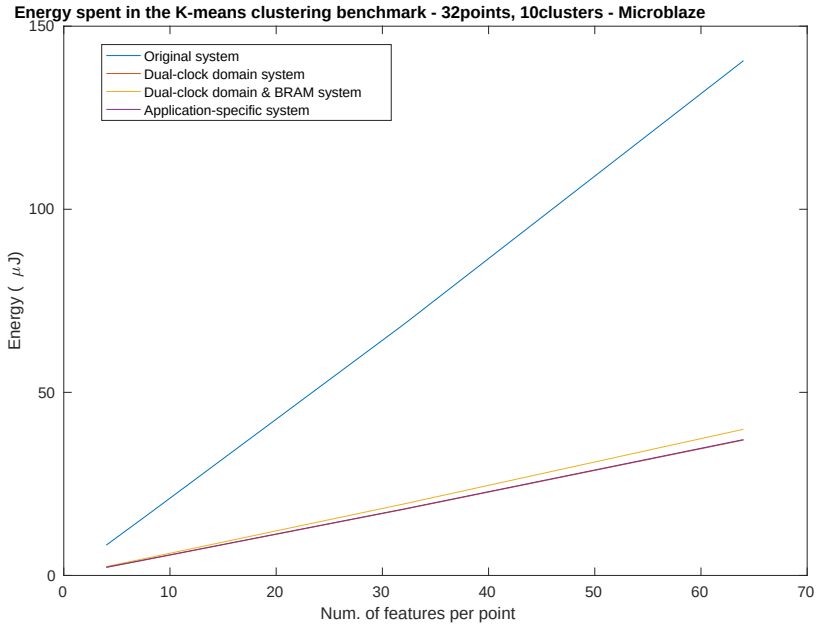


Figure B.9: Energy spent in the K-means clustering benchmark - 32points, 10clusters - Microblaze

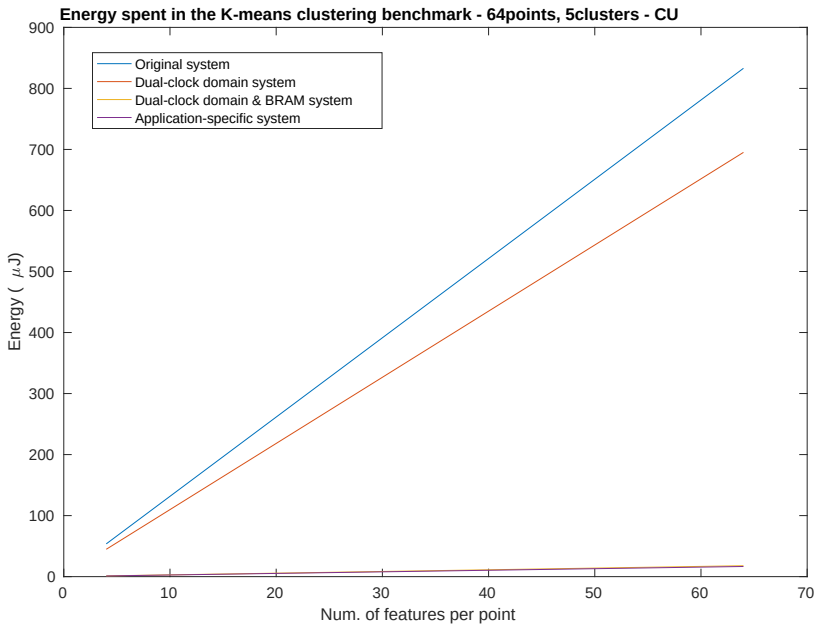


Figure B.10: Energy spent in the K-means clustering benchmark - 64points, 5clusters - CU

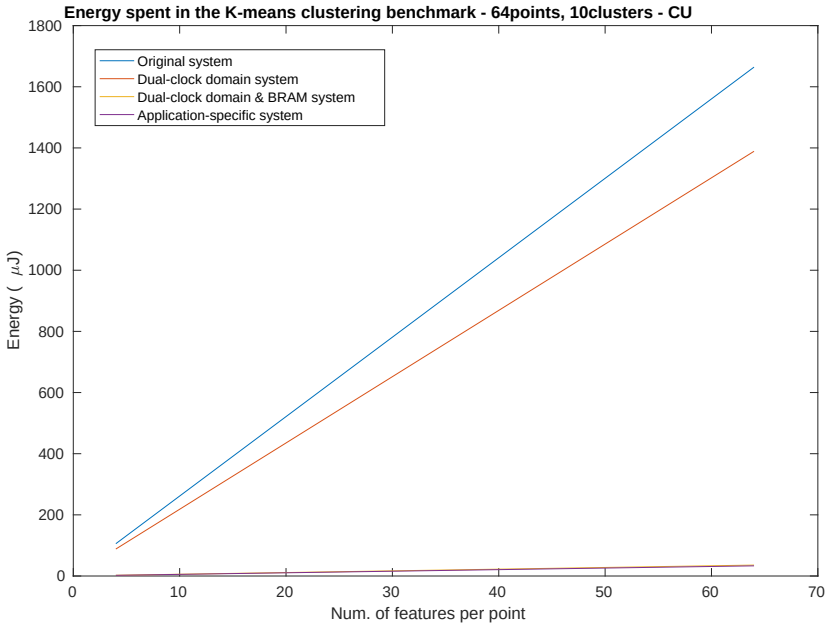


Figure B.11: Energy spent in the K-means clustering benchmark - 64points, 10clusters - CU

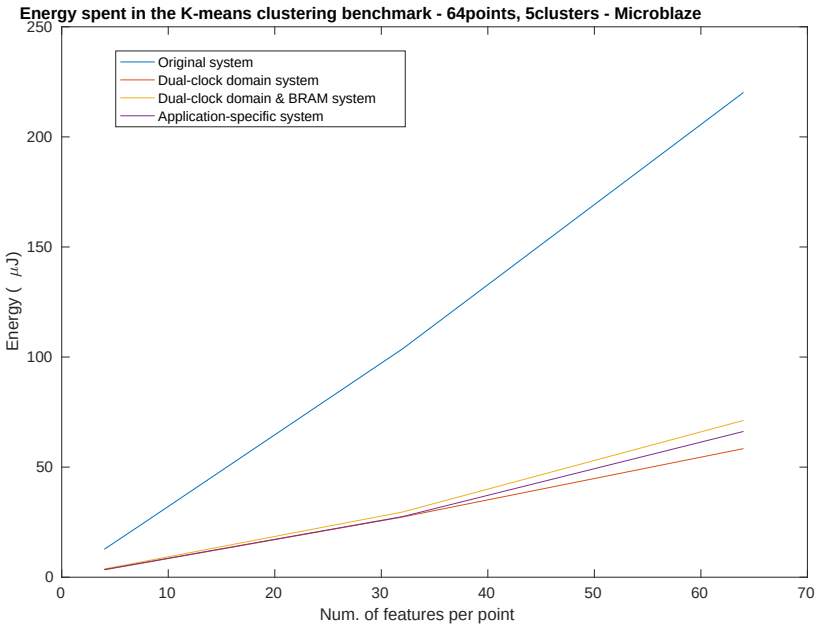


Figure B.12: Energy spent in the K-means clustering benchmark - 64points, 5clusters - Microblaze

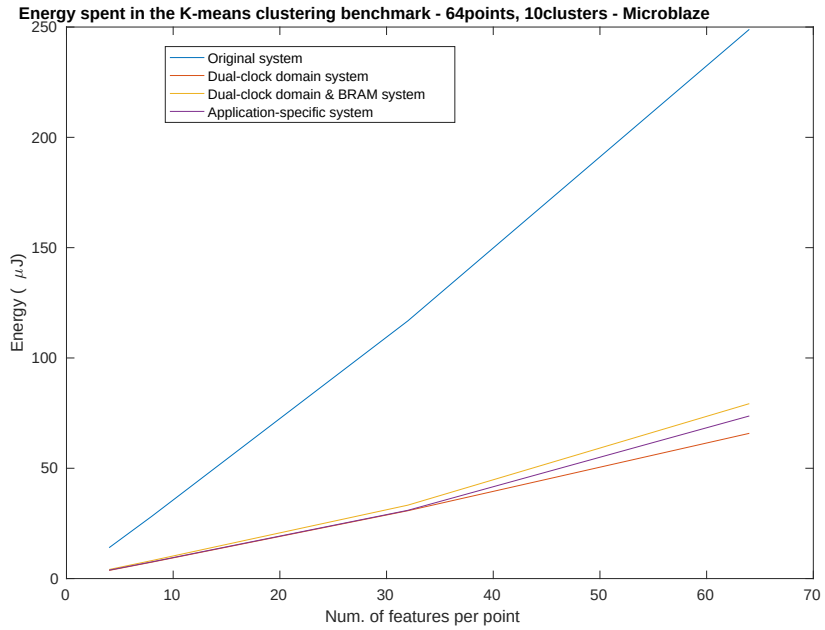


Figure B.13: Energy spent in the K-means clustering benchmark - 64points, 10clusters - Microblaze

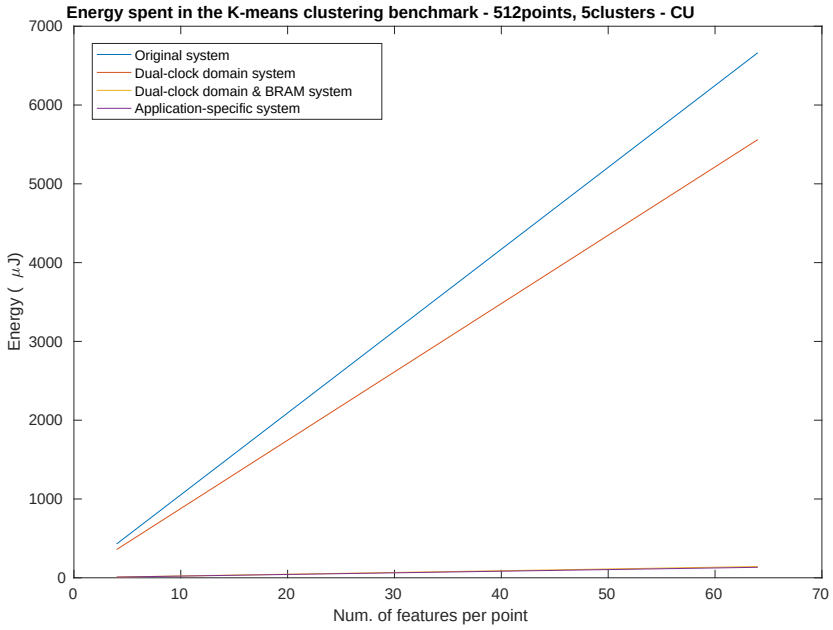


Figure B.14: Energy spent in the K-means clustering benchmark - 512points, 5clusters - CU

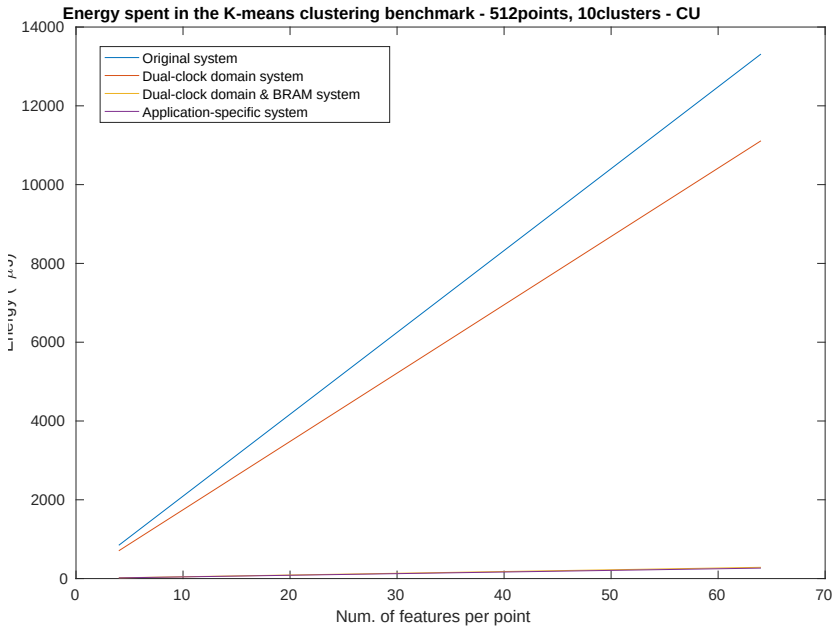


Figure B.15: Energy spent in the K-means clustering benchmark - 512points, 10clusters - CU

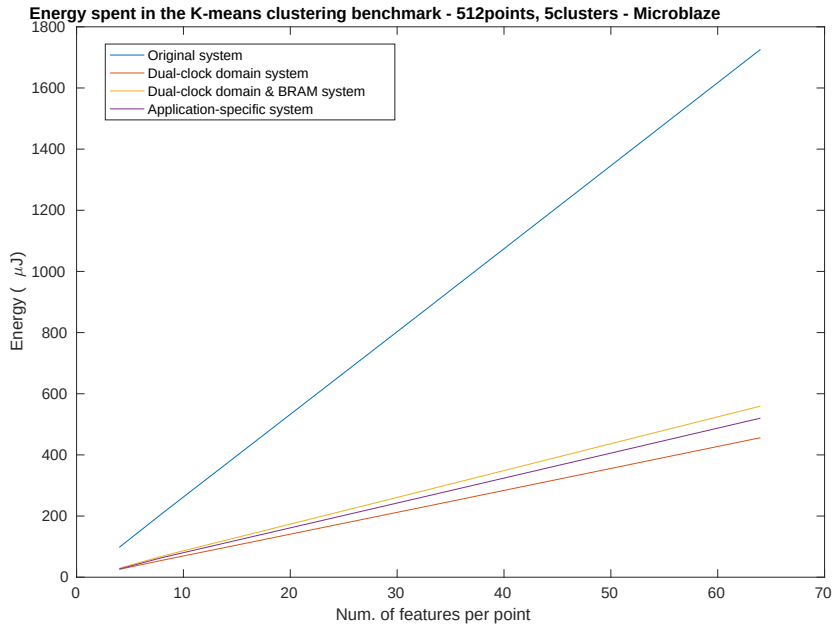


Figure B.16: Energy spent in the K-means clustering benchmark - 512points, 5clusters - Microblaze

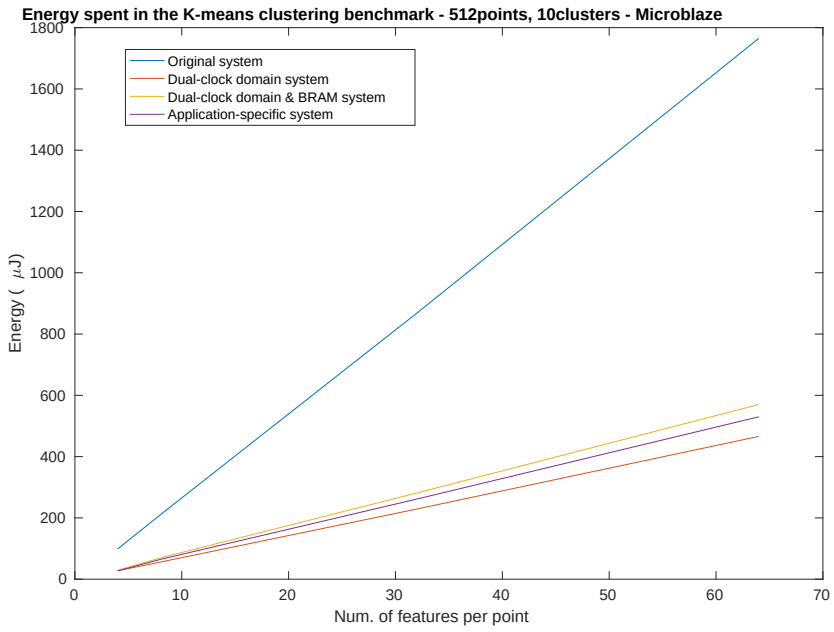


Figure B.17: Energy spent in the K-means clustering benchmark - 512points, 10clusters - Microblaze

C

Scalar Instruction Testing

```
#include <stdio.h>
#include "platform.h"
#include "xio.h"
#include "xparameters.h"

#define NEKO_CMD_ADDR XPAR_AXI_SLAVE_0_S00_AXI_BASEADDR
#define NEKO_BASE_LDS (NEKO_CMD_ADDR + 16)
#define NEKO_INSTR_ADDR (NEKO_CMD_ADDR + 28)
#define NEKO_INSTR_VALUE (NEKO_CMD_ADDR + 32)
#define NEKO_GPR_CMD (NEKO_CMD_ADDR + 40)
#define NEKO_SGRP_ADDR (NEKO_CMD_ADDR + 44)
#define NEKO_SGRP_QUAD_0 (NEKO_CMD_ADDR + 48)
#define NEKO_SGRP_QUAD_1 (NEKO_CMD_ADDR + 52)
#define NEKO_SGRP_QUAD_2 (NEKO_CMD_ADDR + 56)
#define NEKO_SGRP_QUAD_3 (NEKO_CMD_ADDR + 60)

#define NEKO_MEM_OP (NEKO_CMD_ADDR + 128)
#define NEKO_MEM_RD_DATA (NEKO_CMD_ADDR + 132) // Address for data to be
    read
//from MIAOW and written to memory
#define NEKO_MEM_ADDR (NEKO_CMD_ADDR + 136)
#define NEKO_MEM_WR_DATA (NEKO_CMD_ADDR + 192)//Addr. for writing data to
    MIAOW
#define NEKO_MEM_WR_EN (NEKO_CMD_ADDR + 196)
#define NEKO_MEM_ACK (NEKO_CMD_ADDR + 200)
#define NEKO_MEM_DONE (NEKO_CMD_ADDR + 204)

#define NEKO_CYCLE_COUNTER (NEKO_CMD_ADDR + 192)

#define NEKO_RESET (NEKO_CMD_ADDR + 36)

#define MEM_WR_ACK_WAIT 1
#define MEM_WR_RDY_WAIT 2
#define MEM_WR_LSU_WAIT 3
#define MEM_RD_ACK_WAIT 4
#define MEM_RD_RDY_WAIT 5
#define MEM_RD_LSU_WAIT 6
#define VGPR_DATA (NEKO_CMD_ADDR + 0x0D4)
#define VGPR_ADDR (NEKO_CMD_ADDR + 0x0D0)
#define VGPR_WR_CMD (NEKO_CMD_ADDR + 0x01D4)
#define VGPR_WR_CLEAN (NEKO_CMD_ADDR + 0x01D8)
#define VGPR_WR_MASK_LO (NEKO_CMD_ADDR + 0x01DC)
#define VGPR_WR_MASK_HI (NEKO_CMD_ADDR + 0x01E0)

#define END_PRGRM 0xBF810000

union ufloat{
     float f;
     unsigned u;
  };

uint32_t reverse_bit_order(register uint32_t x){
  //Inverts the bits in a 32bit word
  //Credits to: Sean Eron Anderson
  //http://graphics.stanford.edu/~seander/bithacks.html
  x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
  x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
  x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
  x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
  return((x >> 16) | (x << 16));
```

```
}
int32_t create_sop2(int32_t op, int32_t sdst, int32_t s1, int32_t s0){
  /* SOP2 instruction format:
   * MSB -> LSB
   * | ENC(2) = 2'b10 | OP(7) | SDST(7) | SSRC1(8) | SSRC0 |
   */
  int32_t inst = 0x80000000;
  op = op << 23;
  sdst = sdst << 16;
  s1 = s1 << 8;
  inst = inst | op | sdst | s1 | s0;
  return(inst);
}
int32_t create_sopk(int32_t op, int32_t sdst, int32_t simm){
  /* SOPK instruction format:
   * MSB -> LSB
   * | ENC(4) = 4'b1011 | OP(5) | SDST(7) | SIMM16(16) |
   */
  int32_t inst = 0xB0000000;
  op = op << 23;
  sdst = sdst << 16;
  inst = inst | sdst | op | simm;
  return(inst);
}
int32_t create_sop1(int32_t op, int32_t sdst, int32_t s0){
  /* SOP1 instruction format:
   * MSB -> LSB
   * | ENC(9) = 9'b101111101 | SDST(7) | OP(8) | SSRC0(8) |
   */
  int32_t inst = 0xBE800000;
  op = op << 8;
  sdst = sdst << 16;
  inst = inst | sdst | op | s0;
  return(inst);
}
int32_t create_sopc(int32_t op, int32_t s1, int32_t s0){
  /* SOPC instruction format:
   * MSB -> LSB
   * | ENC(9) = 9'b101111110 | OP(7) | SSRC1(8) | SSRC0(8) |
   */
  int32_t inst = 0xBF000000;
  s1 = s1 << 8;
  op = op << 16;
  inst = inst | op | s1 | s0;
  return(inst);
}
int32_t create_sopp(int32_t op, int32_t simm){
  /* SOPP instruction format:
   * MSB -> LSB
   * | ENC(9) = 9'b101111111 | OP(7) | SIMM16(16) |
   */
  int32_t inst = 0xBF800000;
  op = op << 16;
  inst = inst | op | simm;
  return(inst);
}
int32_t create_smrd(int32_t op, int32_t sdst, int32_t sbase, int32_t imm,
                                                             int32_t offset
                                                                ){
  /* SMRD instruction format:
```

```
   * MSB -> LSB
   * | ENC(5) = 5'b11000 | OP(5) | SDST(7) | SBASE(6) | IMM(1) | OFFSET(8)
       |
   */
  int32_t inst = 0xC0000000;
  op = op << 22;
  sdst = sdst << 15;
  sbase = sbase << 9;
  imm = imm << 8;
  inst = inst | op | sdst | sbase | imm | offset;
  return(inst);
}
int32_t run_sop_program_neko(int32_t insts[],int32_t num_insts,
                             int32_t inst_data[], int32_t num_data,
                             int32_t max_clocks){
  /*
   * Resets Neko
   * Populates NEKO's instruction buffer and the scalar registers
   * Send "start execution" command and waits for program completion or
       until
   * the timeout is reached
   * If the program reaches the end of execution before the timeout the
       data in
   * the scalar regs is read and success(1) is returned
   * Otherwise returns 0 (unsuccessful)
   */

  int32_t index, address, data;
  int32_t cycle_counter = 0;
  int32_t succeeded = 1;

  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);


  XIo_Out32(NEKO_BASE_LDS, XPAR_MIG_7SERIES_0_BASEADDR);

  //Load scalar registers with data
  for(index = 0; index < num_data; index+=4)
  {
    XIo_Out32(NEKO_SGRP_ADDR, index);
    XIo_Out32(NEKO_SGRP_QUAD_0, inst_data[index]);
     XIo_Out32(NEKO_SGRP_QUAD_1, inst_data[index+1]);
     XIo_Out32(NEKO_SGRP_QUAD_2, inst_data[index+2]);
     XIo_Out32(NEKO_SGRP_QUAD_3, inst_data[index+3]);
     XIo_Out32(NEKO_GPR_CMD, 1);
  }

  //Load the instruction buffer
  for(index = 0; index < num_insts; index++)
  {
    XIo_Out32(NEKO_INSTR_ADDR, index);
    XIo_Out32(NEKO_INSTR_VALUE, insts[index]);
  }

  //Start execution
  XIo_Out32(NEKO_CMD_ADDR, 1);
```

```
//Wait for the end of execution
while(XIo_In32(NEKO_CMD_ADDR) != 1)
{
//Verify the timeout
  cycle_counter = XIo_In32(NEKO_CYCLE_COUNTER);
  if(cycle_counter>max_clocks){
    succeeded=0;
    break;
  }
  data = XIo_In32(NEKO_MEM_OP);
  if(data != 0)
  {
    int nextValue = MEM_RD_RDY_WAIT;
    if(data == MEM_RD_ACK_WAIT)
    {
      nextValue = MEM_RD_RDY_WAIT;
    }
    else if(data == MEM_WR_ACK_WAIT)
    {
      nextValue = MEM_WR_RDY_WAIT;
    }
    else if(data == MEM_WR_LSU_WAIT || data == MEM_RD_LSU_WAIT )
      continue;//last instruction is not finished yet

    XIo_Out32(NEKO_MEM_ACK, 0);
    XIo_Out32(NEKO_MEM_ACK, 1);

    do {
      data = XIo_In32(NEKO_MEM_OP);
    } while(data != nextValue);

    address = XIo_In32(NEKO_MEM_ADDR);

    if(nextValue == MEM_RD_RDY_WAIT)
    {
      data = XIo_In32(address);
      XIo_Out32(NEKO_MEM_WR_DATA, 0x012345678);
      nextValue = MEM_RD_LSU_WAIT;
    }
    else
    {
      data = XIo_In32(NEKO_MEM_RD_DATA);
      XIo_Out32(address, data);
      nextValue = MEM_WR_LSU_WAIT;
    }

    XIo_Out32(NEKO_MEM_DONE, 0);
    XIo_Out32(NEKO_MEM_DONE, 1);

    do {
      data = XIo_In32(NEKO_MEM_OP);

    } while(data != 0 && data != nextValue && data != MEM_RD_ACK_WAIT
                                           && data != MEM_WR_ACK_WAIT);
  }
}

if(succeeded){
  //Retrieve the data in the scalar registers (results)
  for(index = 0; index < num_data; index+=4)
```

```
    {
      XIo_Out32(NEKO_SGRP_ADDR, index);
      inst_data[index]=XIo_In32(NEKO_SGRP_QUAD_0);
      inst_data[index+1]=XIo_In32(NEKO_SGRP_QUAD_1);
      inst_data[index+2]=XIo_In32(NEKO_SGRP_QUAD_2);
      inst_data[index+3]=XIo_In32(NEKO_SGRP_QUAD_3);
    }
  }
  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);

  return(succeeded);
}
int32_t analyze_sop2(int32_t opcode, int32_t s1, int32_t s0,
                     int32_t res,int32_t scc){
  //Checks the result of the operation (opcode) with operands s1 and s0 and
//results in res
  //Some instructions write to the scc register

  //TODO: Correctly implement 64bit functionality in opcodes: 15, 17, 21
  int32_t correct = 0;
  //Convertion of operands and result to unsigned int which is used by some
//of the operations
  uint32_t u_s1 = (uint32_t)s1;
  uint32_t u_s0 = (uint32_t)s0;
  uint32_t u_res = (uint32_t)res;

  //Result storage variables
  int32_t ver_res;
  uint32_t u_ver_res;

  switch(opcode){
  case 0://s_add_u32
    u_ver_res = u_s0 + u_s1;
    if(u_res==u_ver_res)
      correct = 1;
    break;
  case 1://s_sub_u32
    u_ver_res = u_s0 - u_s1;
    if(u_res==u_ver_res)
      correct = 1;
    break;
  case 2://s_add_i32
    ver_res = s0+s1;
    if(ver_res==res)
      correct = 1;
    break;
  case 3://s_sub_i32
    ver_res = s0-s1;
    if(ver_res==res)
      correct = 1;
    break;
  case 7://s_min_u32
    ver_res = s0 < s1 ? s0 : s1;
    if(ver_res==res)
      correct = 1;
    break;
  case 8://s_max_i32
```

```
  ver_res = s0 > s1 ? s0 : s1;
  if(ver_res==res)
    correct = 1;
  break;
case 9://s_max_u32
  u_ver_res = u_s0 > u_s1 ? u_s0 : u_s1;
  if(u_ver_res==u_res)
    correct = 1;
  break;
case 10:
  u_ver_res = (scc & 0x01) ? u_s0 : u_s1;
  correct = (u_ver_res == u_res);
  break;
case 14://s_and_b32
case 15://s_and_b64
  if((s0 & s1) == res){
    if(res!=0)
      correct = scc == 1 ? 1 : 0;
    else
      correct = scc == 1 ? 0 : 1;
  }
  break;
case 16://s_or_b32
case 17://s_or_b64
  if((s0 | s1) == res){
    if(res!=0)
      correct = scc == 1 ? 1 : 0;
    else
      correct = scc == 1 ? 0 : 1;
  }
  break;
case 21://s_andn2_b64
  if((s0 & ~s1) == res){
    if(res!=0)
      correct = scc == 1 ? 1 : 0;
    else
      correct = scc == 1 ? 0 : 1;
  }
  break;
case 30://s_lshl_b32
  u_ver_res = u_s0 << (u_s1 & 0x01f);
  if(u_ver_res == u_res){
    if(u_res!=0)
      correct = scc == 1 ? 1 : 0;
    else
      correct = scc == 1 ? 0 : 1;
  }
  break;
case 32://s_lshr_b32
  u_ver_res = u_s0 >> (u_s1 & 0x01f);
  if(u_ver_res == u_res){
    if(u_res!=0)
      correct = scc == 1 ? 1 : 0;
    else
      correct = scc == 1 ? 0 : 1;
  }
  break;
case 34://s_ashr_i32
  ver_res = s0 >> (s1 & 0x01f);
  if(ver_res == res){
```

```
      if(res!=0)
        correct = scc == 1 ? 1 : 0;
      else
        correct = scc == 1 ? 0 : 1;
    }
    break;
  case 38://s_mul_i32
    ver_res = s0 * s1 ;
    if(ver_res == res)
      correct = 1;
    break;
  default:
    correct = -1;
  }
  return correct;
}

int32_t analyze_sopk(int32_t opcode, int16_t simm, int32_t s0,
                     int32_t res,int32_t scc){
  //Checks the result of the operation (opcode) with operands simm and
//s0(original register value) and results in res(final register value)
  //Some instructions write to the scc register

  int32_t correct = 0;
  int32_t i_simm = (int32_t)simm;

  switch(opcode){
  case 0://s_movk_i32
    if(res == i_simm)
      correct = 1;
    break;
  case 15://s_addk_i32
    if((s0 + i_simm) == res)
      correct =  1;
    break;
  case 16://s_mulk_i32
    if((s0 * i_simm) == res)
      correct =  1;
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_sop1(int32_t opcode, int32_t s[], int32_t res[],int32_t scc
    ){
  //Checks the result of the operation (opcode) with operands s[] and
      results
//in res[]
  //Some instructions write to the scc register

  //TODO: Correctly implement 64bit functionality in opcodes: 4, 8

  int32_t correct = 0;
  uint32_t u_ver_res;

  switch(opcode){
  case 3://s_mov_b32
  case 4://s_mov_b64
    if(res[0] == s[2])
```

```
      correct = 1;
    break;
  case 5:
    u_ver_res = (scc & 0x01) ? (uint32_t)s[2] : (uint32_t)s[0];
    correct = (u_ver_res == (uint32_t)res[0]);
    break;
  case 7://s_not_b32
  case 8://s_not_b64
    if(~s[2] == res[0])
      correct =  1;
    break;
  case 11:
    correct = (reverse_bit_order(s[2]) == (uint32_t)res[0]);
    break;
  case 36://s_and_saveexec_b64
    //needs to read exec!
    /* exec_lo = s2 (==1)
     * exec_hi = s1 (==0)
     * s_and_saveexec_b64 s0, s2
     * s2 = exec_lo
     * s3 = exec_hi
     *  */
    if(res[0] == s[2] && res[1] == s[1]){// res[0] = (exec_lo = s2)
                        // res[1] = (exec_hi = s1)
      if(res[2] == (res[0] & s[2]) && res[3] == (res[1] & s[3])){
                        //res[2] = exec[31:0] & s2
                        //res[3] = exec[63:32]& s3
        if(res[2]!=0 || res[3]!=0)//exec!=0 -> scc = 1
          correct = scc == 1 ? 1 : 0;
        else
          correct = scc == 1 ? 0 : 1;
      }
    }
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_sopc(int32_t opcode, int32_t s1, int32_t s0, int32_t scc){
  //Checks the result(scc) of the comparison (opcode) with operands s1 and
      s0

  int32_t correct = 0;

  //Conversion for the unsigned comparisons
  uint32_t u_s0 = (uint32_t)s0;
  uint32_t u_s1 = (uint32_t)s1;

  switch(opcode){
  case 0://s_cmp_eq_i32
    if((s0 == s1) == scc)
      correct = 1;
    break;
  case 1://s_cmp_lg_i32
    if((s0 != s1) == scc)
      correct = 1;
    break;
  case 2://s_cmp_gt_i32
    if((s0 > s1) == scc)
```

```
      correct = 1;
    break;
  case 3://s_cmp_ge_i32
    if((s0 >= s1) == scc)
      correct = 1;
    break;
  case 4://s_cmp_lt_i32
    if((s0 < s1) == scc)
      correct = 1;
    break;
  case 5://s_cmp_le_i32
    if((s0 <= s1) == scc)
      correct = 1;
    break;
  case 6://s_cmp_eq_u32
    if((u_s0 == u_s1) == scc)
      correct = 1;
    break;
  case 7://s_cmp_lg_u32
    if((u_s0 != u_s1) == scc)
      correct = 1;
    break;
  case 8://s_cmp_gt_u32
    if((u_s0 > u_s1) == scc)
      correct = 1;
    break;
  case 9://s_cmp_ge_u32
    if((u_s0 >= u_s1) == scc)
      correct = 1;
    break;
  case 10://s_cmp_lt_u32
    if((u_s0 < u_s1) == scc)
      correct = 1;
    break;
  case 11://s_cmp_le_u32
    if((u_s0 <= u_s1) == scc)
      correct = 1;
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_sopp(int32_t opcode, int32_t res, int32_t exec, int32_t scc
    ){
  //Test if the branches were taken (jumps over s0 = 16)

  //TODO: Correctly test opcodes 10 and 12

  int32_t correct = 0;

  switch(opcode){
  case 0://s_nop
    correct = (res == 17);
    break;
  case 1://s_endpgm
    correct = (res == 0);
    break;
  case 2://s_branch
    correct = (res == 1);
```

```
    break;
  case 4://s_branch_scc0
    if(scc == 0)
      correct = (res == 1);
    else
      correct = (res == 17);
    break;
  case 5://s_branch_scc1
    if(scc == 1)
      correct = (res == 1);
    else
      correct = (res == 17);
    break;
  case 6://s_branch_vccz
  case 7://s_branch_vccnz
    break;
  case 8://s_branch_execz
    if(exec == 0)
      correct = (res == 1);
    else
      correct = (res == 17);
    break;
  case 9://s_branch_execnz
    if(exec == 1)
      correct = (res == 1);
    else
      correct = (res == 17);
    break;
    //TODO: not sure how to test these 2
  case 10://s_barrier
  case 12://s_waitcnt
    correct = 1;
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_smrd(int32_t opcode, int32_t s[]){
  //Tests if the right number of registers got the word "0x12345678"

  switch(opcode){
  case 0://S_LOAD_DWORD
  case 8://S_BUFFER_LOAD_DWORD
    return (s[0] == 0x12345678);
    break;
  case 1://S_LOAD_DWORDX2
  case 9://S_BUFFER_LOAD_DWORDX2
    return (s[0] == 0x12345678 && s[1] == 0x12345678);
    break;
  case 2://S_LOAD_DWORDX4
  case 10://S_BUFFER_LOAD_DWORDX4
    return (s[0] == 0x12345678 && s[1] == 0x12345678 && s[2] == 0x12345678
                               && s[3] == 0x12345678);
    break;
  case 3://S_LOAD_DWORDX8
  case 11://S_BUFFER_LOAD_DWORDX8
    return (s[0] == 0x12345678 && s[1] == 0x12345678 && s[2] == 0x12345678
                               && s[3] == 0x12345678 && s[4] == 0x12345678
                               && s[5] == 0x12345678 && s[6] == 0x12345678
```

```
                                && s[7] == 0x12345678);
    break;
  case 4://S_LOAD_DWORDX16
  case 12://S_BUFFER_LOAD_DWORDX16
    return (s[0] == 0x12345678 && s[1] == 0x12345678 && s[2] == 0x12345678
         && s[3] == 0x12345678 && s[4] == 0x12345678 && s[5] == 0x12345678
         && s[6] == 0x12345678 && s[7] == 0x12345678 && s[8] == 0x12345678
         && s[9] == 0x12345678 && s[10] == 0x12345678 && s[11] == 0
             x12345678
         && s[12] == 0x12345678 && s[13] == 0x12345678 && s[14] == 0
             x12345678
         && s[15] == 0x12345678);
    break;

  default:
    return 0;
  }
}
int32_t test_sop2(){
  //Tests are done by issuing the following instructions and reading the
      results
  //s0 = s2 op s3 -> sop2 inst
  //s1 = scc -> sop1 inst

  int32_t insts[3];
  insts[1] = create_sop1(3, 1, 0xFD); //s1 = scc - sop1 inst
  insts[2] = END_PRGRM;

  int32_t inc_counter, data_counter, res, res_check, inst_data[12];
  int32_t running_insts_counter = 0;

  //Generates SOP2 instructions and verifies their state (running with/
      without
//correct result or not running)
  for(inc_counter = 0; inc_counter<45; inc_counter++){
    if(inc_counter==12 || inc_counter == 13)
      continue;

    //Data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++)
        inst_data[data_counter] = data_counter;

    //Generate SOP2 Instruction
    insts[0] = create_sop2(inc_counter, 0, 2, 3);

    //Check running state
    res = run_sop_program_neko(insts,3,inst_data,12,500);

    if(res){//If the program finishes verifies the result
      res_check = analyze_sop2(inc_counter, 2, 3, inst_data[0],inst_data
          [1]);
      if(res_check == 1){
        xil_printf("SOP2: OPCODE %d OK\n\r",inc_counter);
        running_insts_counter++;
      }
      else if(res_check == 0)
        xil_printf("SOP2: OPCODE %d Wrong Result\n\r",inc_counter);
      else
          xil_printf("SOP2: OPCODE %d No test available\n\r",inc_counter);
    }
```

```
    else
      xil_printf("SOP2:␣OPCODE␣%d␣did␣not␣finish␣running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_sopk(){
  //Tests are done by issuing the following instructions and reading the
      results
  //s0 = op imm(=2) -> sopk inst
  //s1 = scc -> sop1 inst

  int32_t insts[3];
  insts[1] = create_sop1(3, 1, 0xFD); //s1 = scc -> sop1 inst
  insts[2] = END_PRGRM;
  int32_t inc_counter,data_counter;
  int32_t res, res_check, inst_data[12];
  int32_t running_insts_counter = 0;

  //Generates SOPK instructions and verifies their state (running with/
      without
// correct result or not running)
  for(inc_counter = 0; inc_counter<17; inc_counter++){//Last 4 (17-21)
    if(inc_counter==1)                                  //are not implemented
      continue;

    //Data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++)
      inst_data[data_counter] = data_counter+1;

    //Generate SOPK Instruction
    insts[0] = create_sopk(inc_counter, 0, 2);

    //Check running state
    res = run_sop_program_neko(insts,3,inst_data,12,500);

    if(res){//If the program finishes verifies the result
      res_check = analyze_sopk(inc_counter, 2, 1, inst_data[0],inst_data
          [1]);
      if(res_check == 1){
        running_insts_counter++;
          xil_printf("SOPK:␣OPCODE␣%d␣OK\n\r",inc_counter);
      }
      else if(res_check == 0)
        xil_printf("SOPK:␣OPCODE␣%d␣Wrong␣Result\n\r",inc_counter);
      else
          xil_printf("SOPK:␣OPCODE␣%d␣No␣test␣available\n\r",inc_counter);
    }
    else
      xil_printf("SOPK:␣OPCODE␣%d␣did␣not␣finish␣running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_sop1(){
  //Tests are done by issuing the following instructions and reading the
      results
  //exec_hi = s1 -> sop1 inst
  //exec_lo = s2 -> sop1 inst
  //s0 = op s2 -> sop1 inst
  //s2 = exec_lo -> sop1 inst
  //s3 = exec_hi -> sop1 inst
```

```
  //s4 = scc -> sop1 inst

  int32_t insts[7];
  insts[0] = create_sop1(3, 127,1);//exec_hi = s1 (==0)
  insts[1] = create_sop1(3, 126,2);//exec_lo = s2 (==1)
  insts[3] = create_sop1(3, 2,126);//s2 = exec_lo (==0)
  insts[4] = create_sop1(3, 3,127);//s3 = exec_hi (==0)
  insts[5] = create_sop1(3, 4, 0xFD); //s4 = scc -> sop1 inst
  insts[6] = END_PRGRM;

  int32_t inc_counter, data_counter, res, res_check, inst_data[12],ori_data
      [12];
  int32_t running_insts_counter = 0;

  //Generates SOP1 instructions and verifies their state (running with/
      without
//correct result or not running)
  for(inc_counter = 3; inc_counter<54; inc_counter++){
    if(inc_counter == 35 || inc_counter == 51)
      continue;

    //Data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter-1;
      ori_data[data_counter] = data_counter-1;
    }

    //Generate SOP1 Instruction
    insts[2] = create_sop1(inc_counter, 0, 2);

    //Check running state
    res = run_sop_program_neko(insts,7,inst_data,12,500);

    if(res){//If the program finishes verifies the result
      res_check = analyze_sop1(inc_counter, ori_data, inst_data,inst_data
          [4]);
      if(res_check == 1 ){
        running_insts_counter++;
        xil_printf("SOP1: OPCODE %d OK\n\r",inc_counter);
      }
      else if(res_check == 0)
          xil_printf("SOP1: OPCODE %d wrong result\n\r",inc_counter);
      else
          xil_printf("SOP1: OPCODE %d No test available\n\r",inc_counter);
    }
    else
      xil_printf("SOP1: OPCODE %d did not finish running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_sopc(){
  //Tests are done by issuing the following instructions and reading the
      results
  //scc = s2 comp s3 -> sopc inst
  //s1 = scc -> sopc inst

  int32_t insts[3];
  insts[1] = create_sop1(3, 1, 0xFD); //s1 = scc -> sop1 inst
  insts[2] = END_PRGRM;
```

```
  int32_t inc_counter, res, res_check, inst_data[12];
  int32_t running_insts_counter = 0;

  //Data for the scalar registers
  for(inc_counter = 0; inc_counter<12; inc_counter++)
    inst_data[inc_counter] = inc_counter-1;

  //Generates SOPC instructions and verifies their state (running with/
      without
//correct result or not running)
  for(inc_counter = 0; inc_counter<17; inc_counter++){//Last 5 (12-16)
                                                      // are not
                                                         implemented

    //Generate SOPC Instruction
    insts[0] = create_sopc(inc_counter, 2, 3);

    //Check running state
    res = run_sop_program_neko(insts,3,inst_data,12,500);

    if(res){//If the program finishes verifies the result
        res_check = analyze_sopc(inc_counter, 2, 3, inst_data[1]);
      if(res_check == 1){
        running_insts_counter++;
        xil_printf("SOPC:␣OPCODE␣%d␣OK\n\r",inc_counter);
      }
      else if(res_check == 0)
        xil_printf("SOPC:␣OPCODE␣%d␣wrong␣result\n\r",inc_counter);
          else
              xil_printf("SOPC:␣OPCODE␣%d␣No␣test␣available\n\r",
                  inc_counter);
    }
    else
      xil_printf("SOPC:␣OPCODE␣%d␣did␣not␣finish␣running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_sopp(){
  /*Tests are done by issuing the following instructions and reading the
      results
   * scc = s2 comp s3  == 1 -> sopc inst
   * exec = 0x00000000 0x00000001 -> sop1 instructions
   * vcc = ??? (not implemented yet)
   * sopp instruction with imm=1
   * s0 = 16 (may be jumped) -> sopk
   * s0 = s0 + 1 -> sopk
   */

  int32_t insts[7];
  insts[0] = create_sopc(1, 2, 3);//set scc as 1->(scc=(s2 != s3) which is
      true)

  //set exec as 1
  insts[1] = create_sop1(3, 127,0);//exec_hi = s0 (==0)
  insts[2] = create_sop1(3, 126,1);//exec_lo = s1 (==1)

  //branch testing instructions
  insts[4] = create_sopk(0, 0, 16);//s0 = 16
  insts[5] = create_sopk(15, 0, 1);//s0 = s0 +1
```

```
  insts[6] = END_PRGRM;

  int32_t inc_counter,data_counter, res, res_check, inst_data[12];
  int32_t running_insts_counter = 0;


  for(inc_counter = 0; inc_counter<13; inc_counter++){
////Last 10 are not implemented nor tested
    if(inc_counter == 3 || inc_counter == 11)
      continue;
    //Data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++)
      inst_data[data_counter] = data_counter;

    //Generate sopp instruction
    insts[3] = create_sopp(inc_counter, 1);

    //Test if the instruction is running
    res = run_sop_program_neko(insts,7,inst_data,12,500);

    if(res){//If the instruction is running verifies the result
        res_check =  analyze_sopp(inc_counter, inst_data[0], 1, 1);
      if(res_check == 1){
        running_insts_counter++;
        xil_printf("SOPP:␣OPCODE␣%d␣OK\n\r",inc_counter);
      }
      else if(res_check == 0)
        xil_printf("SOPP:␣OPCODE␣%d␣wrong␣result\n\r",inc_counter);
          else
              xil_printf("SOPP:␣OPCODE␣%d␣No␣test␣available\n\r",
                  inc_counter);
    }
    else
      xil_printf("SOPP:␣OPCODE␣%d␣did␣not␣finish␣running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_smrd(){
  //Tests are done by issuing the following instructions and reading the
      results
  //exec_lo = 0xffffffff
  //exec_hi = 0xffffffff
  //s[0..] = mem_access (accesses can get 1,2,4,8,or 16 words from memory,
      which
//will be written contiguosly in the registers)
  //s0 = memory access sbase = s4, imm = 0;

  int32_t insts[4];

  insts[0] = create_sop1(3, 126, 0);//0xBEFE0302;//exec_lo = s0 = 0
      xFFFFFFFF
  insts[1] = create_sop1(3, 127, 0);//0xBEFF0302;//exec_hi = s0 = 0
      xFFFFFFFF


  insts[3] = END_PRGRM;
  int32_t inc_counter, data_counter;
  int32_t res;
  int32_t inst_data[16];
  int32_t running_insts_counter = 0;
```

```
  inc_counter = 0;
  for(inc_counter = 0; inc_counter<13; inc_counter++){//Last 2 (30-31) are
      not
//implemented
    if(inc_counter>4 && inc_counter<8)
      continue;

    for(data_counter = 0; data_counter<16; data_counter++){
      inst_data[data_counter] = data_counter-1;
      if(data_counter == 0)
        inst_data[data_counter] = 0xffffffff;
    }

    insts[2] = create_smrd(inc_counter, 0, 2, 1, 0);

    res = run_sop_program_neko(insts,4,inst_data,16,300000000);
    if(res){
      if(analyze_smrd(inc_counter, inst_data)){
        running_insts_counter++;
        xil_printf("SMRD: OPCODE %d OK\n\r",inc_counter);
      }
      else
        xil_printf("SMRD: OPCODE %d Wrong Result\n\r",inc_counter);
    }
    else
      xil_printf("SMRD: OPCODE %d did not finish running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_scalar_inst_creation(){
  //Verifies the instruction creation by comparing against manually
      generated
//instructions

  xil_printf("Instruction generation\n\r");
  if(create_sop2(2, 1, 4, 1) != 0x81010401){//s_add_i32 s1, s1, s4 -
      81010401
    xil_printf("SOP2: Failed\n\r");
    return 0;
  }
  else
      xil_printf("SOP2: Ok\n\r");

  if(create_sopk(15, 5, 2) != 0xB7850002){//s_addk_i32 s5, 0x0002 -
      B7850002
      xil_printf("SOPK: Failed\n\r");
      return 0;
  }
  else
     xil_printf("SOPK: Ok\n\r");

  if(create_sop1(36, 2, 6) != 0xBE822406){//s_and_saveexec_b64 s[2:3], s
      [6:7]
                                          // - BE822406
     xil_printf("SOP1: Failed\n\r");
     return 0;
  }
  else
```

```
      xil_printf("SOP1: Ok\n\r");

  if(create_sopc(11, 0, 4) != 0xBF0B0004){//s_cmp_le_u32 s4, s0 - BF0B0004
    xil_printf("SOPC: Failed\n\r");
    return 0;
  }
  else
    xil_printf("SOPC: Ok\n\r");

  if(create_sopp(12, 1) != 0xBF8C0001){//s_waitcnt vmcnt(1) & lgkmcnt(0)
                                       //& expcnt(0) -BF8C0001
    xil_printf("SOPP: Failed\n\r");
    return 0;
  }
  else
    xil_printf("SOPP: Ok\n\r");

  if(create_smrd(2, 0,6,1,2) != 0xC0800D02){//s_load_dwordx4 s[0:3], s
      [12:13], 2
    xil_printf("SMRD: Failed\n\r");
    return 0;
  }
  else
    xil_printf("SMRD: Ok\n\r");

  return 1;
}

void test_scalar_instructions(){
  uint32_t total = 0, type_count = 0;
  type_count = test_sop2();
  total += type_count;
  xil_printf("SOP2 - %d instructions running\n\r",type_count);
  type_count = test_sopk();
  total += type_count;
  xil_printf("SOPK - %d instructions running\n\r",type_count);
  type_count = test_sop1();
  total += type_count;
  xil_printf("SOP1 - %d instructions running\n\r",type_count);
  type_count = test_sopc();
  total += type_count;
  xil_printf("SOPC - %d instructions running\n\r",type_count);
  type_count = test_sopp();
  total += type_count;
  xil_printf("SOPP - %d instructions running\n\r",type_count);
  type_count = test_smrd();
  total += type_count;
  xil_printf("SMRD - %d instructions running\n\r",type_count);
  xil_printf("Total scalar instructions running: %d\n\r",total);
}
int main()
{
  init_platform();

  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);

  if(!test_scalar_inst_creation())
    return 1;
```

```
  test_scalar_instructions();

  cleanup_platform();
  return 0;
}
```

D

Vector Instruction Testing

```
#include <stdio.h>
#include <math.h>
#include "platform.h"
#include "xio.h"
#include "xparameters.h"

#define NEKO_CMD_ADDR XPAR_AXI_SLAVE_0_S00_AXI_BASEADDR
#define NEKO_BASE_LDS (NEKO_CMD_ADDR + 16)
#define NEKO_INSTR_ADDR (NEKO_CMD_ADDR + 28)
#define NEKO_INSTR_VALUE (NEKO_CMD_ADDR + 32)
#define NEKO_GPR_CMD (NEKO_CMD_ADDR + 40)
#define NEKO_SGRP_ADDR (NEKO_CMD_ADDR + 44)
#define NEKO_SGRP_QUAD_0 (NEKO_CMD_ADDR + 48)
#define NEKO_SGRP_QUAD_1 (NEKO_CMD_ADDR + 52)
#define NEKO_SGRP_QUAD_2 (NEKO_CMD_ADDR + 56)
#define NEKO_SGRP_QUAD_3 (NEKO_CMD_ADDR + 60)

#define NEKO_MEM_OP (NEKO_CMD_ADDR + 128)
#define NEKO_MEM_RD_DATA (NEKO_CMD_ADDR + 132) // Address for data to be
    read
//from MIAOW and written to memory
#define NEKO_MEM_ADDR (NEKO_CMD_ADDR + 136)
#define NEKO_MEM_WR_DATA (NEKO_CMD_ADDR + 192)//Addr. for writing data to
    MIAOW
#define NEKO_MEM_WR_EN (NEKO_CMD_ADDR + 196)
#define NEKO_MEM_ACK (NEKO_CMD_ADDR + 200)
#define NEKO_MEM_DONE (NEKO_CMD_ADDR + 204)

#define NEKO_CYCLE_COUNTER (NEKO_CMD_ADDR + 192)

#define NEKO_RESET (NEKO_CMD_ADDR + 36)

#define MEM_WR_ACK_WAIT 1
#define MEM_WR_RDY_WAIT 2
#define MEM_WR_LSU_WAIT 3
#define MEM_RD_ACK_WAIT 4
#define MEM_RD_RDY_WAIT 5
#define MEM_RD_LSU_WAIT 6
#define VGPR_DATA (NEKO_CMD_ADDR + 0x0D4)
#define VGPR_ADDR (NEKO_CMD_ADDR + 0x0D0)
#define VGPR_WR_CMD (NEKO_CMD_ADDR + 0x01D4)
#define VGPR_WR_CLEAN (NEKO_CMD_ADDR + 0x01D8)
#define VGPR_WR_MASK_LO (NEKO_CMD_ADDR + 0x01DC)
#define VGPR_WR_MASK_HI (NEKO_CMD_ADDR + 0x01E0)

#define END_PRGRM 0xBF810000
union ufloat{
  float f;
  uint32_t u;
};
union ufloat64{
    double f;
    uint64_t u;
};

int32_t create_sop1(int32_t op, int32_t sdst, int32_t s0){
  /* SOP1 instruction format:
   * MSB -> LSB
   * | ENC(9) = 9'b101111101 | SDST(7) | OP(8) | SSRC0(8) |
   */
```

```
  int32_t inst = 0xBE800000;
  op = op << 8;
  sdst = sdst << 16;
  inst = inst | sdst | op | s0;
  return(inst);
}
int32_t create_vop2(int32_t op, int32_t vdst, int32_t vsrc1, int32_t src0){
  /* VOP2 instruction format:
   * MSB -> LSB
   * | ENC(1) = 1'b0 | OP(6) | VDST(8) | VSRC1(8) | SRC0(9) |
   */
  int32_t inst = 0x00000000;
  op = op << 25;
  vdst = vdst << 17;
  vsrc1 = vsrc1 << 9;
  inst = inst | op | vdst | vsrc1 | src0;
  return(inst);
}
int32_t create_vop1(int32_t op, int32_t vdst, int32_t src0){
  /* VOP1 instruction format:
   * MSB -> LSB
   * | ENC(7) = 7'b0111111 | VDST(8) | OP(8) | SRC0(9) |
   */
  int32_t inst = 0x7E000000;
  op = op << 9;
  vdst = vdst << 17;
  inst = inst | op | vdst | src0;
  return(inst);
}
int32_t create_vopc(int32_t op_base,int32_t op_offset,
                    int32_t vsrc1, int32_t src0){
  /* VOPC instruction format:
   * MSB -> LSB
   * | ENC(7) = 7'b0111110 | OP_Base + OP_Offset (8) | VSRC1(8) | SRC0(9) |
   */
  int32_t inst = 0x7C000000;
  int32_t op = (op_base + op_offset) << 17;
  vsrc1 = vsrc1 << 9;
  inst = inst | op | vsrc1 | src0;
  return(inst);
}
void create_vop3a(int32_t op, int32_t op_offset, int32_t vdst,int32_t src2,
                  int32_t src1, int32_t src0,int32_t abs, int32_t clamp,
                  int32_t omod, int32_t neg, int32_t *inst){
  /* VOP3a instruction format:
   * MSB -> LSB
   * | NEG(3) | OMOD(2) | SRC2(9) | SRC1(9) | SRC0(9) |
   * | ENC(6) = 6'b110100 | OP(9) | RESERVED(1) | CLAMP(1) | ABS(3) | VDST
       (8) |
   */
  inst[0] = 0xD0000000;
  op = (op + op_offset) << 17;
  clamp = clamp << 11;
  abs = abs << 8;
  inst[0] = inst[0] | vdst | abs | clamp | op;

  inst[1] = 0x00000000;
  src1 = src1 << 9;
  src2 = src2 << 18;
  omod = omod << 27;
```

```
  neg = neg << 29;
  inst[1] = inst[1] | neg | omod | src2 | src1 | src0;
}
void create_vop3b(int32_t op, int32_t sdst, int32_t vdst,int32_t src2,
                  int32_t src1, int32_t src0, int32_t omod, int32_t neg,
                  int32_t *inst){
  /* VOP3b instruction format:
   * MSB -> LSB
   * | NEG(3) | OMOD(2) | SRC2(9) | SRC1(9) | SRC0(9) |
   * | ENC(6) = 6'b110100 | OP(9) | RESERVED(2) | SDST(7) | VDST(8) |
   */
  inst[0] = 0xD0000000;
  op = op  << 17;
    sdst = sdst << 8;
  inst[0] = inst[0] | sdst | vdst | op;

  inst[1] = 0x00000000;
  src1 = src1 << 9;
  src2 = src2 << 18;
  omod = omod << 27;
  neg = neg << 29;
  inst[1] = inst[1] | neg | omod | src2 | src1 | src0;
}

int32_t run_vop_program_neko(int32_t insts[],int32_t num_insts,
                             int32_t inst_scalar_data[], int32_t
                                 num_scalar_data
                            ,int32_t inst_vect_data[], int32_t
                               num_vect_data,
                             int32_t max_clocks){
  /*
   * Execution Flow:
   * Resets Neko
   * Populates NEKO's instruction buffer, the scalar registers and the
       vector
   * registers (all 64 words of a register are initialized with the same
       value)
   * Send "start execution" command and waits for program completion or
       until
   * the timeout is reached
   * If the program reaches the end of execution before the timeout the
       data in
   * the registers is read and success(1) is returned
   * Otherwise returns 0 (unsuccessful)
   */
  int32_t index, cycle_counter = 0, succeeded = 1;
  int32_t vgpr, vgpr_word;
  int32_t * vgpr_data_pointer = (int32_t*)VGPR_DATA;

  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);

  XIo_Out32(NEKO_BASE_LDS, XPAR_MIG_7SERIES_0_BASEADDR);

  //Load scalar registers with data
  for(index = 0; index < num_scalar_data; index+=4){
    XIo_Out32(NEKO_SGRP_ADDR, index);
    XIo_Out32(NEKO_SGRP_QUAD_0, inst_scalar_data[index]);
```

```
    XIo_Out32(NEKO_SGRP_QUAD_1, inst_scalar_data[index+1]);
    XIo_Out32(NEKO_SGRP_QUAD_2, inst_scalar_data[index+2]);
    XIo_Out32(NEKO_SGRP_QUAD_3, inst_scalar_data[index+3]);
    XIo_Out32(NEKO_GPR_CMD, 1);
  }

  //Load vector registers with data (replicating the data for every word of
//the register)
  for(vgpr=0;vgpr<num_vect_data;vgpr++){
    XIo_Out32(VGPR_ADDR, vgpr);
    XIo_Out32(VGPR_WR_CLEAN, 1);
    XIo_Out32(VGPR_WR_CMD, 1);
    for(vgpr_word=0;vgpr_word<64;vgpr_word++){
      vgpr_data_pointer[vgpr_word] = inst_vect_data[vgpr];
    }
    XIo_Out32(VGPR_WR_CMD, 1);
  }

  //Load the instruction buffer
  for(index = 0; index < num_insts; index++){
    XIo_Out32(NEKO_INSTR_ADDR, index);
    XIo_Out32(NEKO_INSTR_VALUE, insts[index]);
  }


  //Start execution
  XIo_Out32(NEKO_CMD_ADDR, 1);

  //Wait for the end of execution
  while(XIo_In32(NEKO_CMD_ADDR) != 1){
    //Verify the timeout
    cycle_counter = XIo_In32(NEKO_CYCLE_COUNTER);
    if(cycle_counter>max_clocks){
      succeeded=0;
      break;
    }
  }

  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);

  if(succeeded){
    //Retrieve the data in the scalar registers (results)
    for(index = 0; index < num_scalar_data; index+=4){
      XIo_Out32(NEKO_SGRP_ADDR, index);
      inst_scalar_data[index]=XIo_In32(NEKO_SGRP_QUAD_0);
      inst_scalar_data[index+1]=XIo_In32(NEKO_SGRP_QUAD_1);
      inst_scalar_data[index+2]=XIo_In32(NEKO_SGRP_QUAD_2);
      inst_scalar_data[index+3]=XIo_In32(NEKO_SGRP_QUAD_3);
    }

    //Retrieve the data in the vector registers (results)
    for(vgpr=0;vgpr<num_vect_data;vgpr++){
      XIo_Out32(VGPR_ADDR, vgpr);
      inst_vect_data[vgpr] = vgpr_data_pointer[0];
    }
  }
  return(succeeded);
```

```
}

int32_t test_vector_inst_creation(){
  //Verifies the instruction creation by comparing against manually
      generated
//instructions

  xil_printf("Instruction␣generation\n\r");
  if(create_vop2(37, 3, 2, 5) != 0x4A060405){//v_add_i32 v3, vcc, s5,v2-4
      A060405
    xil_printf("VOP2:␣Failed\n\r");
    return 0;
  }
  else
    xil_printf("VOP2:␣Ok\n\r");

  if(create_vop1(1, 5, 0x106) != 0x7E0A0306){//v_mov_b32 v5, v6 - 7E0A0306
    xil_printf("VOP1:␣Failed\n\r");
    return 0;
  }
  else
    xil_printf("VOP1:␣Ok\n\r");

  if(create_vopc(0x80,4, 2, 0x102) != 0x7D080502){//v_cmp_gt_i32  vcc, v2,
      v2
    xil_printf("VOPC:␣Failed\n\r");
    return 0;
  }
  else
    xil_printf("VOPC:␣Ok\n\r");

  int32_t inst_3a[2];
  create_vop3a(362, 0, 5,0x108, 3,0x102,0, 0, 0, 0, inst_3a);
  //v_mul_hi_u32  v5, v2, s3 - D2D40005 04200702
  if(inst_3a[0] != 0xD2D40005 || inst_3a[1] != 0x04200702){
    xil_printf("VOP3a:␣Failed\n\r");
    return 0;
  }
  else
    xil_printf("VOP3a:␣Ok\n\r");

  int32_t inst_3b[2];
  create_vop3b(293, 1, 2,0, 0x100, 0x101, 0, 0, inst_3b);
  //v_add v2, v0, v1 ; s1 = carry_out
  if(inst_3b[0] != 0xD24A0102 || inst_3b[1] != 0x00020101){
    xil_printf("VOP3b:␣Failed\n\r");
    return 0;
  }
  else
    xil_printf("VOP3b:␣Ok\n\r");

  return 1;
}

int32_t analyze_vop2(int32_t opcode, int32_t s1, int32_t s0, int32_t d,
                     int32_t res,int32_t vcc_lo, int32_t vcc_hi){
  /* Checks the first result of the operation (opcode) with vector operands
       s1
   * and s0 and results in res
   *
```

```
   * Some instructions write to the vcc register, and some use the extra
       value
   * 'd' (which can be the initial value of the register or some constant)
   *
   * TODO: IMPLEMENT COMPLETE VECTOR VERIFICATION (for all 64 words)
   *
   * IMPORTANT: This function assumes that the exec mask is all ones
   * (EXEC_hi = EXEC_lo = 0xFFFFFFFF)
   *
   */

  int32_t correct = 0;
  //Convertion of operands and result to unsigned int and float formats
      which
//are used by some of the operations
  uint32_t u_s1=(uint32_t)s1,
       u_s0=(uint32_t)s0,
       u_res=(uint32_t)res;


  union ufloat f_s1, f_s0, f_res, f_d, f_aux;
  f_s1.u=s1, f_s0.u=s0, f_res.u=res, f_d.u = d;
xil_printf("s1=%08x\n\rs0=%08x\n\rres=%08x\n\r",f_s1.u,f_s0.u,f_res.u);
  switch(opcode){
  case 0://v_cndmask_b32
    correct = (vcc_lo & 0x01) ? (u_res == u_s1) : (u_res == u_s0);

    /*if((vcc_lo & 0x01)!=0){
      correct = (u_res == u_s1);
    }
    else{
      correct = (u_res == u_s0);
    }*/

    break;
  case 3://v_add_f32
    f_aux.f = f_s1.f+f_s0.f;
    correct = (f_res.u==f_aux.u);
    break;
  case 4://v_sub_f32
    correct = (f_res.f==(f_s0.f-f_s1.f));
    break;
  case 5://v_subrev_f32
    correct = (f_res.f==(f_s1.f-f_s0.f));
    break;
  case 8://v_mul_f32
    correct = (f_res.f==(f_s0.f*f_s1.f));
    break;
  case 9://v_mul_i32_i24
    //only the 24 lsb's are multiplied
    correct = (res==(s0&0x00FFFFFF)*(s1&0x00FFFFFF));
    break;
  case 15://v_min_f32
    correct = (f_res.f == (f_s0.f < f_s1.f ? f_s0.f : f_s1.f));
    break;
  case 16://v_max_f32
    correct = (f_res.f == (f_s0.f > f_s1.f ? f_s0.f : f_s1.f));
    break;
  case 18://v_max_i32
    correct = (res == (s0 > s1 ? s0 : s1));
```

```
    break;
  case 19://v_min_u32
    correct = (u_res == (u_s0 < u_s1 ? u_s0 : u_s1));
    break;
  case 20://v_max_u32
    correct = (u_res == (u_s0 > u_s1 ? u_s0 : u_s1));
    break;
  case 22://s1.u >> s0[4:0] v_lshrrev_b32
    correct = (u_res == (u_s1 >> (u_s0 & 0x0F)));
    break;
  case 24://s1.i >> s0[4:0] v_ashrrev_i32
    correct = (res == (s1 >> (s0 & 0x0F)));
    break;
  case 26://s1.i << s0[4:0] v_lshlrev_b32
    correct = (u_res == (u_s1 << (u_s0 & 0x0F)));
    break;
  case 27://v_and_b32
    correct = (u_res == (u_s1 & u_s0 ));
    break;
  case 28://v_or_b32
    correct = (u_res == (u_s1 | u_s0 ));
    break;
  case 31://v_mac_f32 D = S0*S1+D
    correct = (f_res.f == (f_s0.f*f_s1.f+f_d.f));
    break;
  case 32://v_madmk_f32 D = S0*constant+S1
    correct = (f_res.f == (f_s0.f*f_d.f+f_s1.f));
    break;
  case 37://v_add_i32
    correct = (u_res == (u_s1 + u_s0 ));
    break;
  case 38://v_sub_i32
    correct = (u_res == (u_s0 - u_s1 ));
    break;
  case 39://v_subrev_i32
    correct = (u_res == (u_s1 - u_s0 ));
    break;
  case 40://v_addc_u32 d = s0 + s1 + vcc
    correct = (u_res == (u_s1 + u_s0 + vcc_lo));
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_vop1(int32_t opcode, int32_t s0, int32_t res,int32_t vcc_lo
    ,
                     int32_t vcc_hi){
  /* Checks the first result of the operation (opcode) with vector operand
      s0
   * and results in res
   *
   * Some instructions write to the vcc register
   *
   * IMPORTANT: This function assumes that the exec mask is all ones
   * (EXEC_hi = EXEC_lo = 0xFFFFFFFF)
   *
   */

  /*
```

```
   * TODO:
   * IMPLEMENT COMPLETE VECTOR VERIFICATION (for all 64 words)
   * conversions
   * op 35 -> round nearest integer (right now only checks if is equal to 2
   * (since we give 1.5f as our s0))
   * op 37/53/54 -> pow/sin/cos busts the memory instruction available,
       need to
   * check result in a better way
   *
   *
   */
  int32_t correct = 0;

  //Convertion of operands and result to unsigned int and to float formats
      which
// are used by some of the operations
  uint32_t u_s0=(uint32_t)s0,
       u_res=(uint32_t)res;

  union ufloat  f_s0, f_res;
  f_s0.u=s0, f_res.u=res;

  switch(opcode){
  case 1://v_mov_b32
    correct = (u_res == u_s0);
    break;
  case 5://v_cvt_f32_i32
    correct = (f_res.f == (float)s0);
  case 6://v_cvt_f32_u32
    correct = (f_res.f == (float)u_s0);
  case 7://v_cvt_u32_f32
    correct = (u_res == (uint32_t)f_s0.f);
  case 8://v_cvt_i32_f32
      correct = (res == (int32_t)f_s0.f);

  case 32://v_fract_f32
    correct = (f_res.f == (f_s0.f-floor(f_s0.f)));
    break;
  case 33://v_trunc_f32
    correct = (f_res.f == floor(f_s0.f));
    break;
  case 34://v_ceil_f32
    correct = (f_res.f == ceil(f_s0.f));
    break;
  case 35://v_rndne_f32
    correct = (f_res.f == 2.0f);
    break;
  case 36://v_floor_f32
    correct = (f_res.f == floor(f_s0.f));
    break;
  case 37://v_exp_f32
    correct = (f_res.u == 0x403504f3);//pow(2.0f,f_s0.f))
    break;
  case 38://v_log_clamp_f32
  case 39://v_log_f32
    correct = (f_res.f == log2(f_s0.f));
    break;
  case 40://v_rcp_clamp_f32
  case 42://v_rcp_f32
    correct = (f_res.f == (1.0f/f_s0.f));
```

```
    break;
  case 44://v_rsq_clamp_f32
  case 46://v_rsq_f32
    correct = (f_res.f == (1.0f/sqrt(f_s0.f)));
    break;
  case 51://v_sqrt_f32
    correct = (f_res.f == sqrt(f_s0.f));
    break;
  case 53://v_sin_f32
    correct = (f_res.f == 0x3f7f5bd5);//sin(f_s0.f))
    break;
  case 54://v_cos_f32
    correct = (f_res.f == 0x3d90deab);//cos(f_s0.f))
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_vopc(int32_t opcode, int32_t s0, int32_t s1,int32_t vcc_lo,
                     int32_t vcc_hi){
  /* Checks the first result of the comparison with vector operands s1 and
      s0
   * and results in the vcc register
   * returns 1 if the result is correct; 0 if incorrect and -1 if there isn
       't a
   * test available
   *
   * TODO: IMPLEMENT COMPLETE VECTOR VERIFICATION (for all 64 words)
   *
   * IMPORTANT: This function assumes that the exec mask is all ones
   * (EXEC_hi = EXEC_lo = 0xFFFFFFFF)
   */

  int32_t correct = 0;
  //Convertion of operands and result to unsigned int and to float formats
      which
// are used by some of the operations
  uint32_t u_s1=(uint32_t)s1,
       u_s0=(uint32_t)s0;
  union ufloat f_s1, f_s0;
    f_s1.u=s1, f_s0.u=s0;

  switch(opcode){
  case 0://v_cmp_F_f32
  case 0x10://v_cmpx_F_f32
  case 0x40://v_cmps_F_f32
  case 0x50://v_cmpsx_F_f32
    correct = (vcc_lo == 0 && vcc_hi == 0);
    break;
  case 1://v_cmp_LT_f32
  case 0x11://v_cmpx_LT_f32
  case 0x41://v_cmps_LT_f32
  case 0x51://v_cmpsx_LT_f32
    correct = ((vcc_lo & 0x01) == (f_s0.f < f_s1.f));
    break;
  case 2://v_cmp_EQ_f32
  case 0x12://v_cmpx_EQ_f32
  case 0x42://v_cmps_EQ_f32
  case 0x52://v_cmpsx_EQ_f32
```

```
  correct = ((vcc_lo & 0x01) == (f_s0.f == f_s1.f));
  break;
case 3://v_cmp_LE_f32
case 0x13://v_cmpx_LE_f32
case 0x43://v_cmps_LE_f32
case 0x53://v_cmpsx_LE_f32
  correct = ((vcc_lo & 0x01) == (f_s0.f <= f_s1.f));
  break;
case 4://v_cmp_GT_f32
case 0x14://v_cmpx_GT_f32
case 0x44://v_cmps_GT_f32
case 0x54://v_cmpsx_GT_f32
  correct = ((vcc_lo & 0x01) == (f_s0.f > f_s1.f));
  break;
case 5://v_cmp_LG_f32
case 0x15://v_cmpx_LG_f32
case 0x45://v_cmps_LG_f32
case 0x55://v_cmpsx_LG_f32
  correct = ((vcc_lo & 0x01) == (f_s0.f != f_s1.f));
  break;
case 6://v_cmp_GE_f32
case 0x16://v_cmpx_GE_f32
case 0x46://v_cmps_GE_f32
case 0x56://v_cmpsx_GE_f32
  correct = ((vcc_lo & 0x01) == (f_s0.f >= f_s1.f));
  break;
case 7://v_cmp_O_f32
case 0x17://v_cmpx_O_f32
case 0x47://v_cmps_O_f32
case 0x57://v_cmpsx_O_f32
  //float32 NaN -> Exponent = all ones (8bits) (infinity)
  //          Mantissa = at least one bit different than 0
  if(!(((f_s0.u & 0x7f800000) == 0x7f800000) && ((f_s0.u & 0x007fffff)
      !=0))
     //!(s0 is nan)
      &&
     !(((f_s1.u & 0x7f800000) == 0x7f800000) && ((f_s1.u & 0x007fffff)
        !=0))
     //!(s1 is nan)
  ){
    correct = ((vcc_lo & 0x01) == 1) ? 1 : 0;
  }
  else
    correct = ((vcc_lo & 0x01) == 0) ? 1 : 0;
  break;
case 8://v_cmp_U_f32
case 0x18://v_cmpx_U_f32
case 0x48://v_cmps_U_f32
case 0x58://v_cmpsx_U_f32
  if(!(( (f_s0.u & 0x7f800000) == 0x7f800000) && (f_s0.u & ~0x007fffff))
      //!(s0 is nan)
      ||
      !(((f_s1.u & 0x7f800000) == 0x7f800000) && (f_s1.u & ~0x007fffff))
      //!(s1 is nan)
  )
    correct = ((vcc_lo & 0x01) == 1) ? 1 : 0;
  else
    correct = ((vcc_lo & 0x01) == 0) ? 1 : 0;
  break;
case 9://v_cmp_NGE_f32
```

```
case 0x19://v_cmpx_NGE_f32
case 0x49://v_cmps_NGE_f32
case 0x59://v_cmpsx_NGE_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f >= f_s1.f));
  break;
case 10://v_cmp_NLG_f32
case 0x1A://v_cmpx_NLG_f32
case 0x4A://v_cmps_NLG_f32
case 0x5A://v_cmpsx_NLG_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f != f_s1.f));
  break;
case 11://v_cmp_NGT_f32
case 0x1B://v_cmpx_NGT_f32
case 0x4B://v_cmps_NGT_f32
case 0x5B://v_cmpsx_NGT_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f > f_s1.f));
  break;
case 12://v_cmp_NLE_f32
case 0x1C://v_cmpx_NLE_f32
case 0x4C://v_cmps_NLE_f32
case 0x5C://v_cmpsx_NLE_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f <= f_s1.f));
  break;
case 13://v_cmp_NEQ_f32
case 0x1D://v_cmpx_NEQ_f32
case 0x4D://v_cmps_NEQ_f32
case 0x5D://v_cmpsx_NEQ_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f == f_s1.f));
  break;
case 14://v_cmp_NLT_f32
case 0x1E://v_cmpx_NLT_f32
case 0x4E://v_cmps_NLT_f32
case 0x5E://v_cmpsx_NLT_f32
  correct = ((vcc_lo & 0x01) == !(f_s0.f < f_s1.f));
  break;
case 15://v_cmp_TRU_f32
case 0x1F://v_cmpx_TRU_f32
case 0x4F://v_cmps_TRU_f32
case 0x5F://v_cmpsx_TRU_f32
  correct = (vcc_lo == 0xFFFFFFFF && vcc_hi == 0xFFFFFFFF);
  break;

case 0x80://v_cmp_F_i32
  correct = (vcc_lo == 0 && vcc_hi == 0);
  break;
case 0x81://v_cmp_LT_i32
  correct = ((vcc_lo & 0x01) == (s0 < s1));
  break;
case 0x82://v_cmp_EQ_i32
  correct = ((vcc_lo & 0x01) == (s0 == s1));
  break;
case 0x83://v_cmp_LE_i32
  correct = ((vcc_lo & 0x01) == (s0 <= s1));
  break;
case 0x84://v_cmp_GT_i32
  correct = ((vcc_lo & 0x01) == (s0 > s1));
  break;
case 0x85://v_cmp_LG_i32
  correct = ((vcc_lo & 0x01) == (s0 != s1));
  break;
```

```
  case 0x86://v_cmp_GE_i32
    correct = ((vcc_lo & 0x01) == (s0 >= s1));
    break;
  case 0x87://v_cmp_TRU_i32
    correct = (vcc_lo == 0xFFFFFFFF && vcc_hi == 0xFFFFFFFF);
    break;

  case 0xC0://v_cmp_F_u32
    correct = (vcc_lo == 0 && vcc_hi == 0);
    break;
  case 0xC1://v_cmp_LT_u32
    correct = ((vcc_lo & 0x01) == (u_s0 < u_s1));
    break;
  case 0xC2://v_cmp_EQ_u32
    correct = ((vcc_lo & 0x01) == (u_s0 == u_s1));
    break;
  case 0xC3://v_cmp_LE_u32
    correct = ((vcc_lo & 0x01) == (u_s0 <= u_s1));
    break;
  case 0xC4://v_cmp_GT_u32
    correct = ((vcc_lo & 0x01) == (u_s0 > u_s1));
    break;
  case 0xC5://v_cmp_LG_u32
    correct = ((vcc_lo & 0x01) == (u_s0 != u_s1));
    break;
  case 0xC6://v_cmp_GE_u32
    correct = ((vcc_lo & 0x01) == (u_s0 >= u_s1));
    break;
  case 0xC7://v_cmp_TRU_u32
    correct = (vcc_lo == 0xFFFFFFFF && vcc_hi == 0xFFFFFFFF);
    break;
  default:
    correct = -1;
  }
  return correct;
}
int32_t analyze_vop3a(int32_t opcode, int32_t s2, int32_t s1, int32_t s0,
                      int32_t d, int32_t res,int32_t vcc_lo, int32_t vcc_hi
                         ){
  /* Checks the first result of the operation with vector operands s1 and
      s0
   * (and, in some cases s2) and results in res
     *
     * TODO: IMPLEMENT COMPLETE VECTOR VERIFICATION (for all 64 words)
     *       64bit instructions - opcodes: 332,356,357,358,359
     *
     * IMPORTANT: This function assumes that the exec mask is all ones
     * (EXEC_hi = EXEC_lo = 0xFFFFFFFF)
     */

  int32_t correct = 0;

  //Convertion of operands and result to unsigned int, float and unsigned
      int
//64bit formats which are used by some of the operations
  uint32_t u_s2=(uint32_t)s2,
      u_s1=(uint32_t)s1,
       u_s0=(uint32_t)s0,
      u_res=(uint32_t)res;
```

```
union ufloat f_s2,f_s1, f_s0, f_res, f_aux;
f_s2.u=s2,f_s1.u=s1, f_s0.u=s0, f_res.u=res;

uint64_t u64_s1 = (uint64_t)s1,
     u64_s0 = (uint64_t)s0;

switch(opcode){
case 321://v_mad_f32 s0*s1+s2
  correct = (f_res.f == (f_s0.f*f_s1.f+f_s2.f));
  break;
case 328://v_bfe_u32 (s0>>s1[4:0]) & ( (1<<s2[4:0]) -1 )
  correct = (u_res == ((u_s0 >> (u_s1 & 0x0f)) & ((1<<(u_s2 & 0x0f))-1)))
      ;
  break;
case 329://v_bfe_i32
  correct = (res == ((s0 >> (s1 & 0x0f)) & ((1<<(s2 & 0x0f))-1)));
  break;
case 330://v_bfi_b32 (s0 & s1) | (~s0 & s2)
  correct = (u_res == ((u_s0 & u_s1) | (~u_s0 & u_s2)));
  break;
case 331://v_fma_f32 s0*s1+s2
  correct = (f_res.f == (f_s0.f*f_s1.f+f_s2.f));
  break;
case 337://v_min3_f32
    f_aux.f = f_s0.f < f_s1.f ? f_s0.f : f_s1.f;
    correct = (f_res.f == (f_aux.f < f_s2.f ? f_aux.f : f_s2.f));
    break;
case 340://v_max3_f32
  f_aux.f = f_s0.f > f_s1.f ? f_s0.f : f_s1.f;
  correct = (f_res.f == (f_aux.f > f_s2.f ? f_aux.f : f_s2.f));
  break;
case 343://v_med3_f32
  //med -> a, b, c -- b>=a && b=<c
  if(f_s0.f >= f_s1.f){
    if(f_s1.f >= f_s2.f)
      correct = (f_res.f == f_s1.f);
    else if(f_s0.f >= f_s2.f)
      correct = (f_res.f == f_s2.f);
    else
      correct = (f_res.f == f_s0.f);
  }
  else{
    if(f_s0.f >= f_s2.f)
      correct = (f_res.f == f_s0.f);
    else if(f_s1.f >= f_s2.f)
      correct = (f_res.f == f_s2.f);
    else
      correct = (f_res.f == f_s1.f);
  }
  break;
case 361://v_mul_lo_u32
  correct = (u_res == u_s0 * u_s1);
  break;
case 362://v_mul_hi_u32
  correct = (u_res == (uint32_t)((u64_s0 * u64_s1)>>32));
  break;
case 363://v_mul_lo_i32
  correct = (res == s0 * s1);
  break;
default:
```

```
    correct = -1;
  }
  return correct;
}

int32_t test_vop2(){
  /*Tests are done by issuing the following instructions and reading the
      results
   * exec_lo = s2
   * exec_hi = s2
   * s0 = vcc_lo
   * s1 = vcc_hi
   * v0 = v2 op v3
   * s3 = vcc_hi (this instruction can be replaced by constant when needed)
   * s2 = vcc_lo
   * s3 = vcc_hi
   */
  int32_t insts[9];

  insts[0] = 0xBEFE0302;//exec_lo = s2 = 0xFFFFFFFF
  insts[1] = 0xBEFF0302;//exec_hi = s2 = 0xFFFFFFFF
  insts[2] = create_sop1(3, 0, 106); //s0 = vcc_lo -> sop1 inst
  insts[3] = create_sop1(3, 1, 107); //s1 = vcc_hi -> sop1 inst

  insts[5] = create_sop1(3, 3, 107); //Repeated instruction is a place
      holder
//for a constant
  insts[6] = create_sop1(3, 2, 106); //s2 = vcc_lo -> sop1 inst
  insts[7] = create_sop1(3, 3, 107); //s3 = vcc_hi -> sop1 inst
  insts[8] = END_PRGRM;

  int32_t inc_counter, data_counter, vect_data_counter;
  int32_t res, check_res;
  int32_t inst_data[12],inst_vect_data[10];//,ori_vect_data[10][64];
  int32_t running_insts_counter = 0;

  for(inc_counter = 0; inc_counter<49; inc_counter++){
    //Generate data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter;
      if(data_counter == 2)
        inst_data[data_counter] = 0xFFFFFFFF;
    }
    //Generate data for the vector registers (which will be replicated for
        all
//64 words of the register)
    for(vect_data_counter = 0; vect_data_counter<10; vect_data_counter++){
      inst_vect_data[vect_data_counter] = vect_data_counter;
    }

    //Generate the VOP2 instruction to be tested
    insts[4] =  create_vop2(inc_counter, 0, 3, 0x102);

    //Set the constant for the op's that need it (v_madmk_f32 and
        v_madak_f32)
    if(inc_counter == 32 || inc_counter == 33){
      insts[5] = 0x0;
    }
    else
      insts[5] = create_sop1(3, 3, 107);
```

```
    //Test the instruction
    res = run_vop_program_neko(insts,9, inst_data, 12,inst_vect_data,
        10,15000);

    if(res){
      //If the program completed then the result is tested
      check_res = analyze_vop2(inc_counter, 3,2,0, inst_vect_data[0],
                               inst_data[0], inst_data[1]);
      if(check_res == 1){
        xil_printf("VOP2: OPCODE %d OK\n\r",inc_counter);
        running_insts_counter++;
      }
      else if(check_res == 0)
        xil_printf("VOP2: OPCODE %d wrong result\n\r",inc_counter);
      else
        xil_printf("VOP2: OPCODE %d No test available\n\r",inc_counter);
    }
    else
      xil_printf("VOP2: OPCODE %d did not finish running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_vop1(){
  /*Tests are done by issuing the following instructions and reading the
      results
   * exec_lo = s2
   * exec_hi = s2
   * v0 = op v4
   * s0 = vcc_lo
   * s1 = vcc_hi
   */

  int32_t insts[6];

  insts[0] = 0xBEFE0302;//exec_lo = s2 = 0xFFFFFFFF
  insts[1] = 0xBEFF0302;//exec_hi = s2 = 0xFFFFFFFF

  insts[3] = create_sop1(3, 0, 106); //s0 = vcc_lo -> sop1 inst
  insts[4] = create_sop1(3, 1, 107); //s1 = vcc_hi -> sop1 inst
  insts[5] = END_PRGRM;

  int32_t inc_counter, data_counter, running_insts_counter = 0;
  int32_t res, check_res;
  int32_t inst_data[12],inst_vect_data[10];

  for(inc_counter = 0; inc_counter<68; inc_counter++){
    if(inc_counter >= 23 && inc_counter <= 31)//reserved values
      continue;

    //Generate data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter;
      if(data_counter == 2)
        inst_data[data_counter] = 0xFFFFFFFF;
    }
    //Generate data for the vector registers (which will be replicated for
        all
//64 words of the register)
    for(data_counter = 0; data_counter<10; data_counter++){
```

```
      inst_vect_data[data_counter] = data_counter;
      if(data_counter == 4)
        inst_vect_data[data_counter] = 0x3fc00000;//v[4] = 1.5f
    }

    //Generate the VOP1 instruction to be tested
    insts[2] =  create_vop1(inc_counter, 0, 0x104);

    //Test the instruction
    res = run_vop_program_neko(insts,6, inst_data, 12,inst_vect_data,
        10,15000);

    if(res){
      //If the program completed then the result is verified
      check_res = analyze_vop1(inc_counter, 0x3fc00000, inst_vect_data[0],
                               inst_data[0], inst_data[1]);
      if(check_res == 1){
        running_insts_counter++;
        xil_printf("VOP1: OPCODE %d OK\n\r",inc_counter);
      }
      else if(check_res == 0)
        xil_printf("VOP1: OPCODE %d wrong result\n\r",inc_counter);
      else
        xil_printf("VOP1: OPCODE %d No test available\n\r",inc_counter);

    }
    else
      xil_printf("VOP1: OPCODE %d did not finish running\n\r",inc_counter);
  }
  return(running_insts_counter);
}
int32_t test_vopc(){
  /*Tests are done by issuing the following instructions and reading the
      results
   * exec_lo = s2
   * exec_hi = s2
   * vcc = v2 comp v3
   * s0 = vcc_lo
   * s1 = vcc_hi
   *
   * The instruction are only considered correct if the whole family
   * (opcode base) is running correctly
   * This rule has an exception when the base opcode corresponds to an OP16
       and
   *  the offset is either 7 or 8 (checking if there is a NaN).
   */

  int32_t insts[6];

  insts[0] = 0xBEFE0302;//exec_lo = s2 = 0xFFFFFFFF
  insts[1] = 0xBEFF0302;//exec_hi = s2 = 0xFFFFFFFF

  insts[3] = create_sop1(3, 0, 106); //s0 = vcc_lo -> sop1 inst
  insts[4] = create_sop1(3, 1, 107); //s1 = vcc_hi -> sop1 inst
  insts[5] = END_PRGRM;

  int32_t inc_counter, data_counter, running_insts_counter = 0,
          group_inst_counter;
  int32_t res, check_res;
  int32_t inst_data[12],inst_vect_data[10];
```

```
  int32_t max_offset, offset, opcode;

  for(inc_counter = 0; inc_counter<16; inc_counter++){
    //Set if it's a cmp_{op16} or a cmp_{op8}
    if(inc_counter<8)
      max_offset = 15;
    else
      max_offset = 7;

    //Set the OPcode Base
    opcode = inc_counter << 4;

    //Run through every OPcode offset
    group_inst_counter = 0;
    for(offset = 0; offset <= max_offset; offset++){

      //Generate data for the scalar registers
      for(data_counter = 0; data_counter<12; data_counter++){
        inst_data[data_counter] = data_counter;
        if(data_counter == 2)
          inst_data[data_counter] = 0xFFFFFFFF;
      }

      //Generate data for the vector registers (which will be replicated
          for all
// 64 words of the register)
      for(data_counter = 0; data_counter<10; data_counter++)
        inst_vect_data[data_counter] = data_counter;

      //Generate the VOPC operation to be tested
      insts[2] =  create_vopc(opcode,offset, 3, 0x102);

      //Test the OP
      res = run_vop_program_neko(insts,6, inst_data, 12,inst_vect_data,
          10,15000);

      if(res){
        //If the program finished running then the result is verified
        check_res = analyze_vopc(opcode+offset, 2,3,inst_data[0],inst_data
            [1]);
        if(check_res == 1){
          group_inst_counter++;
          xil_printf("VOPC: OPCODE %02x OK\n\r",opcode+offset);
        }
        else if(check_res == 0){
          xil_printf("VOPC: OPCODE %02x wrong result\n\r",opcode+offset);
          if(!((opcode==0x0 || opcode==0x10 ||opcode==0x40||opcode==0x50)
              &&
                (offset == 7 || offset == 8))){
            group_inst_counter=0;
            break;
          }
        }
        else
          xil_printf("VOPC: OPCODE %02x No test available\n\r",opcode+
              offset);
      }
      else{
        xil_printf("VOPC: OPCODE %02x did not finish running\n\r",opcode+
            offset);
```

```
        if(!((opcode==0x0 || opcode==0x10 ||opcode==0x40||opcode==0x50) &&
             (offset == 7 || offset == 8))){
          group_inst_counter=0;
          break;
        }
      }
    }
    running_insts_counter+=group_inst_counter;
  }
  return(running_insts_counter);
}
int32_t test_vop3a(){
  /*Tests are done by issuing the following instructions and reading the
     results
   * exec_lo = s0
   * exec_hi = s0
   * In the comparison (VOPC equivalent) instructions:
   *   s0 = v3 comp v2
   * In the VOP2 equivalent:
   *  v0 = v2 op v1
   *some of the vop2 have an extra constant which is added after the
      instruction
   * In the VOP3a only:
   *  v0 = v2 op v1 op v3
   * In the VOP1 equivalent:
   *  v0 = op v4
   * s2 = vcc_lo
   * s3 = vcc_hi
   */

  int32_t insts[8];

  insts[0] = create_sop1(3, 126, 0);//0xBEFE0302;//exec_lo = s0 = 0
     xFFFFFFFF
  insts[1] = create_sop1(3, 127, 0);//0xBEFF0302;//exec_hi = s0 = 0
     xFFFFFFFF

  insts[4] = create_sop1(3, 2, 106); //s2 = vcc_lo -> sop1 inst
  insts[5] = create_sop1(3, 3, 107); //s3 = vcc_hi -> sop1 inst
  insts[6] = END_PRGRM;

  int32_t inc_counter, data_counter,running_insts_counter = 0;
  int32_t max_offset, offset, opcode, group_inst_counter;
  int32_t res, check_res;
  int32_t inst_data[12], inst_vect_data[10];
  int32_t d_constant = 0;

  /*Separate the instructions between the comparison ones, the vop2 ones,
     the
   * vop3 and the vop1 ones*/

  //Testing the VOPC equivalent
  for(inc_counter = 0; inc_counter<16; inc_counter++){
    //Set if it's a cmp_{op16} or a cmp_{op8}
    if(inc_counter<8)
      max_offset = 15;
    else
      max_offset = 7;

    group_inst_counter = 0;
```

```
    //Set the OPcode Base
    opcode = inc_counter << 4;

    //Run through every opcode offset
    for(offset = 0; offset <= max_offset; offset++){
      //Generate data for the scalar registers
      for(data_counter = 0; data_counter<12; data_counter++){
        inst_data[data_counter] = data_counter;
        if(data_counter == 0)
          inst_data[data_counter] = 0xFFFFFFFF;
      }
      //Generate data for the vector registers (which will be replicated
          for all
// 64 words of the register)
      for(data_counter = 0; data_counter<10; data_counter++)
        inst_vect_data[data_counter] = data_counter;

      //Generate the VOP3a operation to be tested
      create_vop3a(opcode,offset, 0,0x101, 0x102, 0x103,0,0,0,0, &insts[2])
          ;

      //Test the program
      res = run_vop_program_neko(insts,7, inst_data, 12,
                                         inst_vect_data, 10,15000);

      if(res){//If the program finishes test the result (tests if the whole
//family is running correctly)
       //These first instructions can be directly analyzed in the "
           analyze_vopc"
        check_res = analyze_vopc(opcode+offset, 3,2,inst_data[0],inst_data
            [1]);
        if(check_res == 1){
          group_inst_counter++;
        }
        else if(check_res == 0){
          xil_printf("VOP3a: OPCODE %02x wrong result\n\r",opcode+offset);
          if(!((opcode==0x0 || opcode==0x10 ||opcode==0x40||opcode==0x50)
            && (offset == 7 || offset == 8))){
            group_inst_counter=0;
            break;
          }
        }
        else
          xil_printf("VOP3a: OPCODE %02x No test available\n\r",opcode+
              offset);
      }
      else{
        xil_printf("VOP3a: OPCODE %02x did not finish running\n\r",
                   opcode+offset);
        if(!((opcode==0x0 || opcode==0x10 ||opcode==0x40||opcode==0x50)
          && (offset == 7 || offset == 8))){
          group_inst_counter=0;
          break;
        }
      }
    }
    if(group_inst_counter!=0)//If the whole family ran that opcode base is
        OK!
```

```
      xil_printf("VOP3a: OPCODE %02x OK - %d instructions running\n\r",
          opcode,
                                                                group_inst_counter
                                                                    );
    else
      xil_printf("VOP3a: OPCODE %02x NOT OK\n\r",opcode);
    running_insts_counter+=group_inst_counter;
  }
  //Testing the VOP2 instructions
  for(inc_counter = 256; inc_counter<306; inc_counter++){
    if(inc_counter>=293 && inc_counter<=298)//vop3b instructions
      continue;

    //Generate data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter+1;
      if(data_counter == 0)
        inst_data[data_counter] = 0xFFFFFFFF;
    }
    //Generate data for the vector registers (which will be replicated for
        all
//64 words of the register)
    for(data_counter = 0; data_counter<10; data_counter++)
      inst_vect_data[data_counter] = (int32_t)data_counter;

    d_constant = 0;

    //Sets v1 and v2 to more computation "friendly" f32 values except for
        the
//v_mul_i32_i24 instruction
    if(inc_counter != 265){
      inst_vect_data[1] = 0x3fc00000;//v[1] = 1.5f
      inst_vect_data[2] = 0x40000000;//v[2] = 2.0f
    }

    //Adds a constant for the instructions that need it
    //(v_madmk_f32 and v_madak_f32)
    if(inc_counter==288 || inc_counter==289){

      d_constant = 0x3f800000;//1.0
      //Sets the constant after the instruction
      insts[4] = d_constant; //inline constant == 1.0f
      //Moves the remaining instructions one slot down
      insts[5] = create_sop1(3, 2, 106); //s2 = vcc_lo -> sop1 inst
      insts[6] = create_sop1(3, 3, 107); //s3 = vcc_hi -> sop1 inst
      insts[7] = END_PRGRM;
    }

    //Generates the instruction to be tested
    create_vop3a(inc_counter, 0, 0,0x103, 0x101, 0x102,0,0,0,0, &insts[2]);

    //Tests the program
    res = run_vop_program_neko(insts,8, inst_data, 12,inst_vect_data,
        10,15000);

    if(res){
      //If the program has finished tests the result with the VOP2
          verification
      check_res = analyze_vop2(inc_counter-256, inst_vect_data[1],
```

```
                                inst_vect_data[2], d_constant,
                                    inst_vect_data[0],
                                inst_data[2],inst_data[3]);
      if(check_res == 1){
        running_insts_counter++;
        xil_printf("VOP3a: OPCODE %d OK\n\r",inc_counter);
      }
      else if(check_res == 0)
        xil_printf("VOP3a: OPCODE %d wrong result\n\r",inc_counter);
      else
        xil_printf("VOP3a: OPCODE %d No test available\n\r",inc_counter);
    }
    else
      xil_printf("VOP3a: OPCODE %d did not finish running\n\r",inc_counter)
          ;

    if(inc_counter==288 || inc_counter==289){
      //Restores the instruction to their rightful place
      insts[4] = create_sop1(3, 2, 106); //s2 = vcc_lo -> sop1 inst
      insts[5] = create_sop1(3, 3, 107); //s3 = vcc_hi -> sop1 inst
      insts[6] = END_PRGRM;
    }
  }
  //Testing the VOP3 instructions
  for(inc_counter = 320; inc_counter<373; inc_counter++){
    if(inc_counter== 365|| inc_counter == 366)//vop3b
      continue;
    //Generate data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter+1;
      if(data_counter == 0)
        inst_data[data_counter] = 0xFFFFFFFF;
    }
    //Generate data for the vector registers (which will be replicated for
        all
//64 words of the register)
    for(data_counter = 0; data_counter<10; data_counter++)
      inst_vect_data[data_counter] = (int32_t)data_counter;

    //Generates the instruction to be tested
    create_vop3a(inc_counter, 0, 0,0x103, 0x101, 0x102,0,0,0,0, &insts[2]);

    //Tests the program
    res = run_vop_program_neko(insts,7, inst_data, 12,inst_vect_data,
        10,15000);

    if(res){
      //If the program has finished tests the result with the VOP3a
          verification
      check_res = analyze_vop3a(inc_counter, inst_vect_data[3],
                                inst_vect_data[1],inst_vect_data[2],
                                    d_constant,
                                inst_vect_data[0],inst_data[2],inst_data
                                    [3]);
      if(check_res == 1){
        running_insts_counter++;
        xil_printf("VOP3a: OPCODE %d OK\n\r",inc_counter);
      }
      else if(check_res == 0)
        xil_printf("VOP3a: OPCODE %d wrong result\n\r",inc_counter);
```

```
      else
        xil_printf("VOP3a: OPCODE %d No test available\n\r",inc_counter);

    }
    else
      xil_printf("VOP3a: OPCODE %d did not finish running\n\r",inc_counter)
          ;
  }

  //Testing the VOP1 equivalent instructions
  for(inc_counter = 384; inc_counter<453; inc_counter++){
    if(inc_counter>=407 && inc_counter<=415)//Reserved
      continue;
    //Generate data for the scalar registers
    for(data_counter = 0; data_counter<12; data_counter++){
      inst_data[data_counter] = data_counter+1;
      if(data_counter == 0)
        inst_data[data_counter] = 0xFFFFFFFF;
    }
    //Generate data for the vector registers (which will be replicated for
        all
//64 words of the register)
    for(data_counter = 0; data_counter<10; data_counter++){
      inst_vect_data[data_counter] = (int32_t)data_counter;
      if(data_counter == 4)
        inst_vect_data[data_counter] = 0x3fc00000;//1.5f
    }
    //Generates the instruction to be tested
    create_vop3a(inc_counter, 0, 0,0x100, 0x100, 0x104,0,0,0,0, &insts[2]);

    //Runs the program
    res = run_vop_program_neko(insts,7, inst_data, 12,inst_vect_data,
        10,15000);
    if(res){
      //If the program has finished tests the result with the VOP1
          verification
      check_res = analyze_vop1(inc_counter-384, 0x3fc00000, inst_vect_data
          [0],
                               inst_data[2], inst_data[3]);
      if(check_res == 1){
        running_insts_counter++;
        xil_printf("VOP3a: OPCODE %d OK\n\r",inc_counter);
      }
      else if(check_res == 0)
        xil_printf("VOP3a: OPCODE %d wrong result\n\r",inc_counter);
      else
        xil_printf("VOP3a: OPCODE %d No test available\n\r",inc_counter);
    }
    else
      xil_printf("VOP3a: OPCODE %d did not finish running\n\r",inc_counter)
          ;
  }
  return(running_insts_counter);
}
int32_t test_vop3b(){
  /*Tests are done by issuing the following instructions and reading the
      results
   * exec_lo = s0
   * exec_hi = s0
   * v0 = v3 op v2 ; s0 = carry_out
```

```
   */

  int32_t insts[5];

  insts[0] = create_sop1(3, 126, 0);//0xBEFE0302;//exec_lo = s0 = 0
      xFFFFFFFF
  insts[1] = create_sop1(3, 127, 0);//0xBEFF0302;//exec_hi = s0 = 0
      xFFFFFFFF

  insts[4] = END_PRGRM;
  int32_t inc_counter, data_counter, running_insts_counter = 0;
  int32_t res;
  int32_t inst_data[12], inst_vect_data[10];

  for(inc_counter = 293; inc_counter<299; inc_counter++){//missing 365 and
      366
    //Generate data for the scalar registers
      for(data_counter = 0; data_counter<12; data_counter++){
        inst_data[data_counter] = data_counter+1;
        if(data_counter == 0)
          inst_data[data_counter] = 0xFFFFFFFF;
      }
      //Generate data for the vector registers
      for(data_counter = 0; data_counter<10; data_counter++)
        inst_vect_data[data_counter] = (int32_t)data_counter;

      //Generate the VOP3b operation to be tested
      create_vop3b(inc_counter, 0, 0,0x101, 0x102, 0x103, 0, 0, &insts[2]);

      //Run the program
      res = run_vop_program_neko(insts,5, inst_data, 12,
                                          inst_vect_data, 10,15000);

      if(res){//Check if the program finishes
        running_insts_counter++;
        xil_printf("VOP3b: OPCODE %d is running\n\r",inc_counter);
        }
      else
        xil_printf("VOP3b: OPCODE %d did not finish running\n\r",
            inc_counter);
  }
  return(running_insts_counter);
}

void test_vector_instructions(){
  uint32_t total = 0, type_count = 0;
  type_count = test_vop2();
  total += type_count;
  xil_printf("VOP2 - %d instructions running\n\r",type_count);

  type_count = test_vop1();
  total += type_count;
  xil_printf("VOP1 - %d instructions running\n\r",type_count);

  type_count = test_vopc();
  total += type_count;
  xil_printf("VOPC - %d instructions running\n\r",type_count);

  type_count = test_vop3a();
  total += type_count;
```

```
  xil_printf("VOP3a␣-␣%d␣instructions␣running\n\r",type_count);

  type_count = test_vop3b();
  total += type_count;
  xil_printf("VOP3b␣-␣%d␣instructions␣running\n\r",type_count);

  xil_printf("Total␣number␣of␣vector␣instructions␣running␣correctly:␣%d\n\r
      ",
                                                                          total
                                                                              )
                                                                              ;

}

int main()
{
    init_platform();

    XIo_Out32(NEKO_RESET,0);
    XIo_Out32(NEKO_RESET,1);
    XIo_Out32(NEKO_RESET,0);

    if(!test_vector_inst_creation())
          return 1;

    test_vector_instructions();

    XIo_Out32(VGPR_ADDR, 0);
    cleanup_platform();
    return 0;
}
```

E

Memory Instruction Testing

```
#include <stdio.h>
#include <math.h>
#include "platform.h"
#include "xio.h"
#include "xparameters.h"

#define NEKO_CMD_ADDR XPAR_AXI_SLAVE_0_S00_AXI_BASEADDR
#define NEKO_BASE_LDS (NEKO_CMD_ADDR + 16)
#define NEKO_INSTR_ADDR (NEKO_CMD_ADDR + 28)
#define NEKO_INSTR_VALUE (NEKO_CMD_ADDR + 32)
#define NEKO_GPR_CMD (NEKO_CMD_ADDR + 40)
#define NEKO_SGRP_ADDR (NEKO_CMD_ADDR + 44)
#define NEKO_SGRP_QUAD_0 (NEKO_CMD_ADDR + 48)
#define NEKO_SGRP_QUAD_1 (NEKO_CMD_ADDR + 52)
#define NEKO_SGRP_QUAD_2 (NEKO_CMD_ADDR + 56)
#define NEKO_SGRP_QUAD_3 (NEKO_CMD_ADDR + 60)
#define NEKO_MEM_OP (NEKO_CMD_ADDR + 128)
#define NEKO_MEM_RD_DATA (NEKO_CMD_ADDR + 132) // Address for data to be
    read
//from MIAOW and written to memory
#define NEKO_MEM_ADDR (NEKO_CMD_ADDR + 136)
#define NEKO_MEM_WR_DATA (NEKO_CMD_ADDR + 192)//Addr. for writing data to
    MIAOW
#define NEKO_MEM_WR_EN (NEKO_CMD_ADDR + 196)
#define NEKO_MEM_ACK (NEKO_CMD_ADDR + 200)
#define NEKO_MEM_DONE (NEKO_CMD_ADDR + 204)
#define NEKO_CYCLE_COUNTER (NEKO_CMD_ADDR + 192)
#define NEKO_PC (NEKO_CMD_ADDR + 196)
#define NEKO_RESET (NEKO_CMD_ADDR + 36)
#define MEM_WR_ACK_WAIT 1
#define MEM_WR_RDY_WAIT 2
#define MEM_WR_LSU_WAIT 3
#define MEM_RD_ACK_WAIT 4
#define MEM_RD_RDY_WAIT 5
#define MEM_RD_LSU_WAIT 6
#define VGPR_DATA (NEKO_CMD_ADDR + 0x0D4)
#define VGPR_ADDR (NEKO_CMD_ADDR + 0x0D0)
#define VGPR_WR_CMD (NEKO_CMD_ADDR + 0x01D4)
#define VGPR_WR_CLEAN (NEKO_CMD_ADDR + 0x01D8)
#define VGPR_WR_MASK_LO (NEKO_CMD_ADDR + 0x01DC)
#define VGPR_WR_MASK_HI (NEKO_CMD_ADDR + 0x01E0)
#define END_PRGRM 0xBF810000

union ufloat{
     float f;
     uint32_t u;
  } x;
union ufloat64{
       double f;
       uint64_t u;
    };
int32_t create_sop1(int32_t op, int32_t sdst, int32_t s0){
  /* SOP1 instruction format:
   * MSB -> LSB
   * | ENC(9) = 9'b101111101 | SDST(7) | OP(8) | SSRC0(8) |
   */
  int32_t inst = 0xBE800000;
  op = op << 8;
  sdst = sdst << 16;
  inst = inst | sdst | op | s0;
```

```
  return(inst);
}

void create_mtbuf(int32_t op, int32_t offset,int32_t offen,int32_t idxen,
                  int32_t glc,int32_t addr64,int32_t dfmt, int32_t nfmt,
                  int32_t vaddr,int32_t vdata,int32_t srsrc,int32_t slc,
                  int32_t tfe, int32_t soffset,int32_t *inst){
  /* MTBUF instruction format:
   * MSB -> LSB
   * 1st 32 bit word:
   * | ENC(6) = 6'b111010 | NFMT(3) | DFMT(4) | OP(3) | ADDR64(1) | GLC(1)
       |
       IDXEN(1) | OFFEN(1) | OFFSET(12) |
   * 2nd 32bit word
   *|SOFFSET(8) | TFE(1) | SLC(1) | RESERVED(1) | SRSRC(5) | VDATA(8) |
      VADDR(8)|
   */
  inst[0] = 0xE8000000;
  inst[1] = 0x00000000;
  offen = offen << 12;
  idxen = idxen << 13;
  glc = glc << 14;
  addr64 = addr64 << 15;
  op = op << 16;
  dfmt = dfmt << 19;
  nfmt = nfmt << 23;
  inst[0] = inst[0] | nfmt | dfmt | op | addr64 | glc | idxen | offen |
     offset;

  vdata = vdata << 8;
  srsrc = srsrc << 16;
  slc = slc << 22;
  tfe = tfe << 23;
  soffset = soffset << 24;
  inst[1] = inst[1] | soffset | tfe | slc | srsrc | vdata | vaddr;
}


int32_t run_vop_program_neko(int32_t insts[],int32_t num_insts,
                             int32_t inst_scalar_data[], int32_t
                                 num_scalar_data
                            ,int32_t inst_vect_data[], int32_t
                               num_vect_data,
                             int32_t max_clocks){
  /*
   * Execution Flow:
   * Resets Neko
   * Populates NEKO's instruction buffer, the scalar registers and the
      vector
   * registers (all 64 words of a register are initialized with the same
      value)
   * Send "start execution" command, responds to memory requests and waits
      for
   * program completion or until the timeout is reached
   * If the program reaches the end of execution before the timeout the
      data in
   * the registers is read and success(1) is returned
   * Otherwise returns 0 (unsuccessful)
   */
  uint32_t pc;
```

```
  int32_t index, cycle_counter = 0, succeeded = 1,address,data;
  int32_t vgpr, vgpr_word;
  int32_t * vgpr_data_pointer = (int32_t*)VGPR_DATA;

  //NEKO's reset pulse
  XIo_Out32(NEKO_RESET,0);
  XIo_Out32(NEKO_RESET,1);
  XIo_Out32(NEKO_RESET,0);

  XIo_Out32(NEKO_BASE_LDS, XPAR_MIG_7SERIES_0_BASEADDR);

  //Load scalar registers with data
  for(index = 0; index < num_scalar_data; index+=4){
    XIo_Out32(NEKO_SGRP_ADDR, index);
    XIo_Out32(NEKO_SGRP_QUAD_0, inst_scalar_data[index]);
    XIo_Out32(NEKO_SGRP_QUAD_1, inst_scalar_data[index+1]);
    XIo_Out32(NEKO_SGRP_QUAD_2, inst_scalar_data[index+2]);
    XIo_Out32(NEKO_SGRP_QUAD_3, inst_scalar_data[index+3]);
    XIo_Out32(NEKO_GPR_CMD, 1);
  }

  //Load vector registers with data (replicating the data for every word of
       the
//register)
  for(vgpr=0;vgpr<num_vect_data;vgpr++){
    XIo_Out32(VGPR_ADDR, vgpr);
    XIo_Out32(VGPR_WR_CLEAN, 1);
    XIo_Out32(VGPR_WR_CMD, 1);
    for(vgpr_word=0;vgpr_word<64;vgpr_word++){
      vgpr_data_pointer[vgpr_word] = inst_vect_data[vgpr];
    }
    vgpr_data_pointer[60] = 1024;
    XIo_Out32(VGPR_WR_CMD, 1);
  }

  //Load the instruction buffer
  for(index = 0; index < num_insts; index++){
    XIo_Out32(NEKO_INSTR_ADDR, index);
    XIo_Out32(NEKO_INSTR_VALUE, insts[index]);
  }


  //Start execution
  XIo_Out32(NEKO_CMD_ADDR, 1);

  //Wait for the end of execution
  while(XIo_In32(NEKO_CMD_ADDR) != 1){
    //Verify the timeout
    cycle_counter = XIo_In32(NEKO_CYCLE_COUNTER);
    if(cycle_counter>max_clocks){
      succeeded=0;
      break;
    }
    data = XIo_In32(NEKO_MEM_OP);
    if(data != 0)
    {
      //checks if the instruction performs a read or a write
      int nextValue = MEM_RD_RDY_WAIT;
      if(data == MEM_RD_ACK_WAIT)
      {
```

```
      nextValue = MEM_RD_RDY_WAIT;
    }
    else if(data == MEM_WR_ACK_WAIT)
    {
      nextValue = MEM_WR_RDY_WAIT;
    }
    else if(data == MEM_WR_LSU_WAIT || data == MEM_RD_LSU_WAIT )
      continue;//last instruction is not finished yet

    //acknowledges that the request was received
    XIo_Out32(NEKO_MEM_ACK, 0);
    XIo_Out32(NEKO_MEM_ACK, 1);

    do {
      data = XIo_In32(NEKO_MEM_OP);
    } while(data != nextValue);

    //reads the requested address
    address = XIo_In32(NEKO_MEM_ADDR);

    //performs the memory access
    if(nextValue == MEM_RD_RDY_WAIT)
    {
      data = XIo_In32(address);
      XIo_Out32(NEKO_MEM_WR_DATA, 0x012345678);
      nextValue = MEM_RD_LSU_WAIT;
    }
    else
    {
      data = XIo_In32(NEKO_MEM_RD_DATA);
      XIo_Out32(address, data);
      nextValue = MEM_WR_LSU_WAIT;
    }

    //signals request completion
    XIo_Out32(NEKO_MEM_DONE, 0);
    XIo_Out32(NEKO_MEM_DONE, 1);

    do {
      data = XIo_In32(NEKO_MEM_OP);
    } while(data != 0 && data != nextValue && data != MEM_RD_ACK_WAIT
                      && data != MEM_WR_ACK_WAIT);
  }
}

//NEKO's reset pulse
XIo_Out32(NEKO_RESET,0);
XIo_Out32(NEKO_RESET,1);
XIo_Out32(NEKO_RESET,0);

if(succeeded){
  //Retrieve the data in the scalar registers (results)
  for(index = 0; index < num_scalar_data; index+=4){
    XIo_Out32(NEKO_SGRP_ADDR, index);
    inst_scalar_data[index]=XIo_In32(NEKO_SGRP_QUAD_0);
    inst_scalar_data[index+1]=XIo_In32(NEKO_SGRP_QUAD_1);
    inst_scalar_data[index+2]=XIo_In32(NEKO_SGRP_QUAD_2);
    inst_scalar_data[index+3]=XIo_In32(NEKO_SGRP_QUAD_3);
  }
```

```
    //Retrieve the data in the vector registers (results)
    for(vgpr=0;vgpr<num_vect_data;vgpr++){
      XIo_Out32(VGPR_ADDR, vgpr);
      inst_vect_data[vgpr] = vgpr_data_pointer[0];
    }
  }
  return(succeeded);
}
int32_t test_mtbuf(){
  //Tests are done by issuing the following instructions and reading the
      results
  //exec_lo = 0xffffffff
  //exec_hi = 0xffffffff
  //v0 is used to perform a memory access (either read or write)
  int32_t insts[5];

  insts[0] = create_sop1(3, 126, 2);//0xBEFE0302;//exec_lo = s2 = 0
      xFFFFFFFF
  insts[1] = create_sop1(3, 127, 2);//0xBEFF0302;//exec_hi = s2 = 0
      xFFFFFFFF

  insts[4] = END_PRGRM;
  int32_t inc_counter;
  int32_t data_counter;
  int32_t res;

  int32_t inst_vect_data[10];
  int32_t running_insts_counter = 0;

  int32_t dmft;
  for(inc_counter = 0; inc_counter<1; inc_counter++){
          int32_t inst_data[] = {0,0,0xffffffff,0,4,5,0x019,9,8,9,10,11};

      for(data_counter = 0; data_counter<10; data_counter++)
        inst_vect_data[data_counter] = (int32_t)data_counter+3;
      switch(inc_counter){
          case '0':
          case '4':
            dmft = 4;
            break;
          case '1':
          case '5':
          dmft = 11;
          break;
          case '2':
          case '6':
          dmft = 13;
          break;
          case '3':
          case '7':
          dmft = 14;
          break;
          default:
          dmft = 4;
      }
      create_mtbuf(inc_counter, 29,1,0,0,0,dmft,0,0,0,0,0,0, 6,&insts[2]);
      //v0 = mem_access

      res = run_vop_program_neko(insts,5, inst_data, 12,
                                          inst_vect_data, 10,2000000000);
```

```
      if(res){
        running_insts_counter++;
        //if the instruction ran prints the result for us to evaluate it
        xil_printf("MTBUF: OPCODE %d OK\n\r",inc_counter);
        xil_printf("v0_0 = 0x%x       ",inst_vect_data[0]);
        xil_printf("s0 = 0x%x\n\r",inst_data[0]);
        }
      else
        xil_printf("MTBUF: OPCODE %d did not finish running\n\r",
            inc_counter);
  }
  return(running_insts_counter);
}
int main(){
  int32_t vgpr, count;
  int32_t * vgpr_data_pointer = (int32_t*)VGPR_DATA;
  test_mtbuf();
  //dumps the contents of the first 3 vgprs
  for(vgpr=0;vgpr<3;vgpr++){
    XIo_Out32(VGPR_ADDR, vgpr);
    for(count = 0; count<64;count++)
      xil_printf("%08x | ",vgpr_data_pointer[count]);
    xil_printf("\n\r");
  }
  return 0;
}
```