Technical University of Crete
School of Electrical and Computer Engineering

Evaluating the Intel HARP (tightly-coupled CPU-FPGA) platform with an ARM many-core accelerator

Georgios Pekridis

Thesis Committee
Professor Dionisios N. Pnevmatikatos (Supervisor)(ECE TUC)
Professor Apostolos Dollas (ECE TUC)
Associate Professor Eftichios Koutroulis (ECE TUC)

Chania, August 2019

Abstract

The purpose of this thesis was to evaluate Intel's platform with scalable Xeon CPU and an integrated Arria 10 FPGA. The communication between CPU and FPGA is done with three physical channels, one QPI (Quick Path Interconnect) coherent channel and two PCIe non-coherent channels. There is also a shared memory between the two sides. The read and write bandwidth of the FPGA to the shared memory is approximately 19 GB/s respectively. The FPGA side consists of a static and a reconfigurable part. The static part implements all the necessary components to establish the communication with the CPU. The reconfigurable part is connected with the static part through the Core Cache Interface protocol (CCI-P) that provides a level of abstraction to the developer for starting developing accelerators. The system consists of software and hardware implementations. The evaluation was done with an ARM many core accelerator. The ARM core has a 3-stage pipeline, it uses a 32-bit architecture and is implements the ARMv4 instruction set. Also it implements a few basic floating point instructions. The RTL for the ARM core was written in Bluespec System Verilog (BSV). The hardware architecture has 16 ARM cores. Each core has a direct-map cache with a variable size. Instruction and data memories of every core can be initialized from software in order to the processors can execute the programs that are defined by the developer. The code and the data for the internal memories of each core are read form binary files. Each core is assigned with buffers with a certain amount of memory space to read and write data from/to it. The hardware can have access to them with the use of physical addresses. For the purpose of measuring the bandwidth of the design STREAM benchmark was used. Plus a matrix multiplication test was made as a way to check how the architecture handles real life applications.

Acknowledgements

First of all I would like to thank my supervisor Prof. Dionisios Pnevmatikatos for his guidance and support throughout this work and for giving me the opportunity to get involved with this very interesting topic. I would also like to express my gratitude to Prof. Apostolos Dollas and Prof. Eftichios Koutroulis of being members of the committee and for evaluating my thesis.In addition I would like to thank Konstantinos Kyriakidis and Vasilis Amourgianos, my studying partners, for their support at the beginning of this project. I am also immensely grateful to Pancretan Endowment Fund for the scholarship that they provided me for my academic endorsements. The tools for this thesis were provided by Intel Academic Compute Environment and I would like to express my gratitude to Intel's stuff for allowing me to use the needed resources. Of course none of this could be possible without the support of my family all the years of my studies in Chania. Lastly I would like to thank one of the most important people in my life, my significant other, Melina for standing by me and supporting me the last six years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Industry vendors are building and developing heterogeneous systems in order to achieve high bandwidth speeds, accelerate application and limit the power consumption. One of the most promising types among the various heterogeneous acceleration platforms is the CPU-FPGA (Field-programmable gate arrays) systems. The reason is that FPGAs provide reconfigurability to accelerate different applications, low power and high energy efficiency. In this platforms, the CPU and the FPGA is tightly coupled with each other either on the same motherboard either on the same SoC (System on Chip). Such CPU-FPGA platforms are the Alpha Data board, the Amazon F1 ,the IBM CAPI, the Microsoft Catapult, the Convey HC-1 and the Intel Xeon+FPGA Platform. The use of these FPGA-base systems for real life application has already started. For example Microsoft Catapult is used into conventional computer clusters to accelerate large-scale production workloads, such as search engines [22] and neural networks [20]. Amazon also has servers equipped with FPGAs (F1 instance) [2]. In addition Intel developed a platform with a scalable Xeon and an integrated Arria 10 FPGA and has predicted that approximately 30% of servers could have FPGAs in 2020 [12]. All the prior mentioned platforms have some differences on the way that the CPU and the FPGA communicate and how the the memory of the system is accessed.

It is also worth noting that at the same time with this diploma, at the Microprocessor and Hardware Laboratory (MHL) of ECE school at TUC, another diploma was prepared by Kostantinos Kyriakidis. His thesis was to develop cache and branch predictor models for use in a full system simulator that it would be running on the Intel platform.

Motivated by the uprising field of CPU and FPGA and all the advantages that these kind of platforms can offer, this thesis aims to make an approach to evaluate the Intel platform with scalable Xeon and integrated Arria 10 FPGA. Xeon connects with the FPGA with three physical channels, one QPI (Quick Path Interconnect) coherent channel and two PCIe non-coherent channels and they share a common memory that is located on the CPU side. The total bandwidth of the three channel is approximately 19 Gb/s for reading and writing respectively. The evaluation was conducted with an ARM many-core accelerator. The ARM core [21] has a 3-stage pipeline, it uses a 32-bit architecture and is implements the ARMv4 instruction set. Also it implements some basic floating point instructions. The RTL for the ARM core was written in Bluespec System Verilog (BSV). The hardware architecture has 16 ARM cores. Each core has a direct-map cache with a variable size. Instruction and data memories of every core can be initialized from software in order to the processors can execute the programs that are defined by the developer. The code and the data for the internal memories of each core are read form binary files. In each core is assigned buffers with a certain amount of memory space to read and write data from/to it. The hardware can have access to them with the use of physical addresses. For the purpose of measuring the bandwidth of the design STREAM benchmark was used. Additionally a matrix multiplication test was made as a way to check how the architecture handles real life applications.

This thesis is structured as follows

- Chapter 2: Analyze the platform and provide some details for the tools that were used.
- Chapter 3: Description of the ARM core and the features that were added
- Chapter 4:Description and implementation of the whole system
- Chapter 5: Experimental results

- Chapter 6: Conclusions and future work.

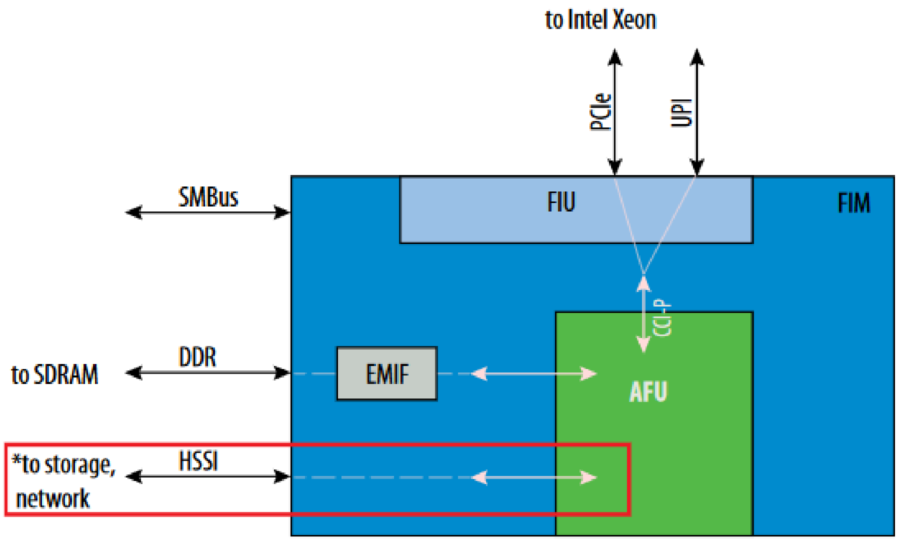
Chapter 2

Intel© Xeon© Scalable Platform with Integrated FPGA

The Intel Xeon Scalable Platform with Integrated FPGA (HARP Platform) is a platform with the Intel Xeon and FPGA in a single package and sharing a coherent view of memory using the Intel Quick Path Interconnect (QPI). The processor is a Broadwell Xeon CPU of the E5-2600v4 family and the FPGA model is an Arria 10 GX1150 FPGA (10AX115U3F45E2SGE3).The communication between the CPU and the FPGA is achieved with the Core Cache Interface (CCI-P). It is a host inteface bus for Accelerator Functional Unit (AFU) and it is intended for connecting the latter to an FPGA Interface Unit(FIU) within an FPGA. The AFU is hardware accelerator which executes a computation task for an application, that came from the CPU. A sowftware app that executes on Xeon can communicate with the FPGA with the Open Programmable Accelerator Engine (OPAE) C API.

2.1 FPGA Interface Manager (FIM)

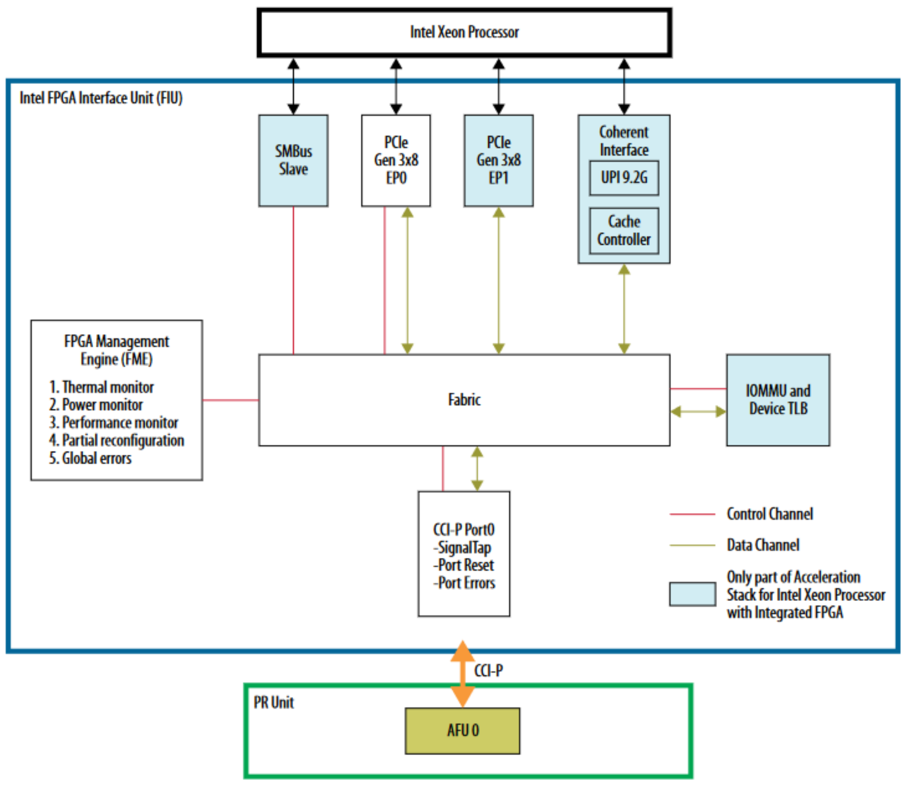
Figure 2.1: FPGA Interface Manage



The whole accelerator package consists of an FIM and AFU. The FIM contains the FIU, an EMMIF for interfacing to external memory and a HSSI for external transceiver interfacing. The FIU has the role of a bridge between AFU and the platform. Also, the FIM owns all hard IPs on FPGA, partial reconfiguration (PR) engine, JTAG atom, IOs, and temperature sensors. The FIM is defined as a static region and the AFU is defined as a partial reconfiguration region.

2.2 FPGA Interface Unit (FIU)

Figure 2.2: FIU for Intel Integrated FPGA Platform Block Diagram



There are three links connecting the FPGA to the processor, one Intel QPI coherent link and two PCIe Gen3x8 links. The FIU maps these links to the CCI-P interface in such a way that the AFU sees one logical communication interface to the host processor with bandwidth equal to the sum bandwidth of the three links. The mapping of the three physical links is done as follows: PCIe0 to VH0, PCIe1 to VH1, QPI to VL0 and all physical links to VA. If an AFU uses VA then it has no information of the physical links and the communication is done through a single logical link that can utilize the total upstream bandwidth available to the FPGA. VA implements a weighted de-multiplexer to route the requests to all of the physical links.

2.3 Memory and Cache Hierarchy

Figure 2.3: Integrated FPGA Platform Memory Hierarchy

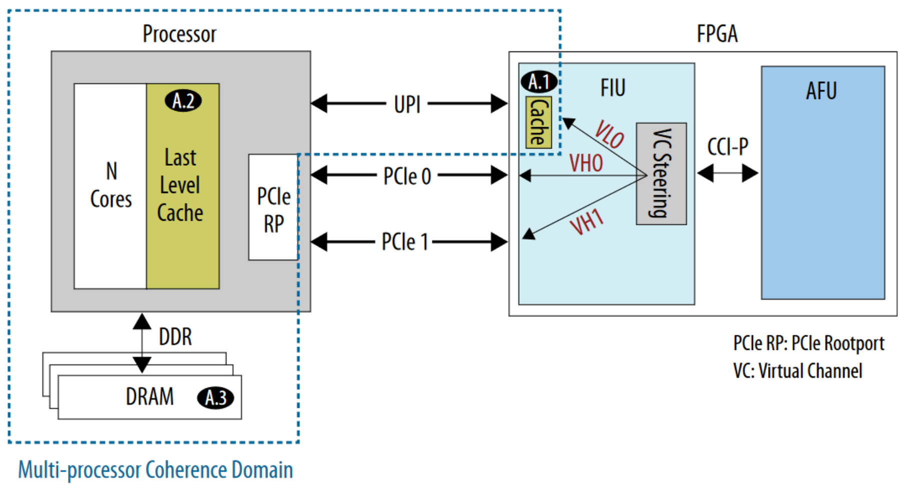
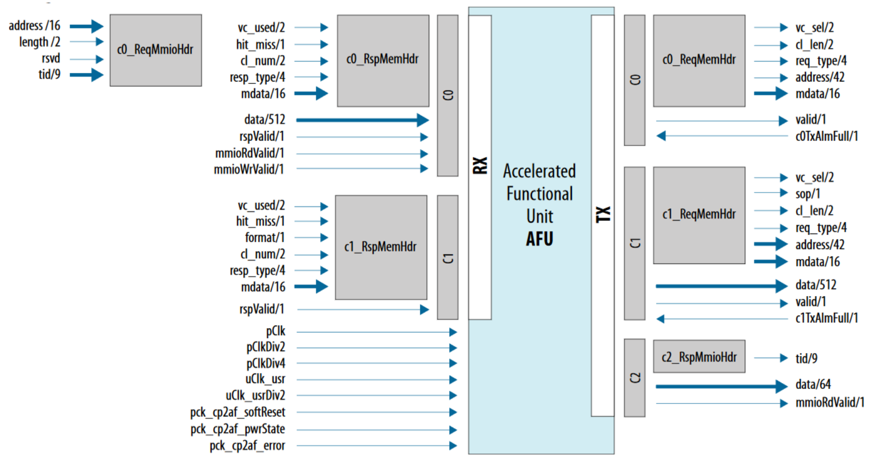


Figure 2.3 shows the three level cache and memory hierarchy seen by an AFU. The platform has one memory node on the processor-side: SDRAM (A.3). The QPI coherent link extends the Xeon processor's coherency domain to the FPGA. A QPI caching agent keeps the FPGA cache in FIU, coherent with the rest of the CPU memory. An AFU request on the CPU memory can be served by the FPGA Cache (A.1), the processor-side cache (A.2) or the processor SDRAM (A.3). As expected the data latency increases from (A.1) to (A.3).In most cases AFUs can achieve maximum memory bandwidth by selecting the VA virtual channel that selects automatically which physical link will serve the request. The choice of the link is made by taking account the physical link latency and efficiency characteristics, physical link utilization and traffic distribution.

2.4 CCI-P Interface

Figure 2.4: CCI-P Signals



The CCI-P implements two memory address spaces, Main Memory and Memory Mapped I/O (MMIO). Main memory is the memory attached to the processor and exposed to the operating system. I/O memory is implemented as CCI-P requests from the host to the AFU. MMIO is typically used as AFU control registers. The AFU developer is responsible for the organization and the implementation of this memory. The CCI-P interface defines a request format to access I/O memory using memory mapped I/O (MMIO) requests. The AFU's MMIO address space is 256 kB in size and is used for exchange of information between software and hardware (accelerator ID,numbers of read/write requests, errors etc). There are three channels that are used to serve read and write requests on main memory and MMIO addres space.The read and write requests for accessing the processors main memory are using physical addresses. In a non-virtualized system, the AFU is expected to drive a host physical address. When in a virtualized system, the AFU is expected to drive a guest physical address. The addressing mode is transparent to the AFU hardware developer. The software application developer must ensure that software provides a physical address to the AFU.

2.4.1 Read Requests

For a read request the AFU sends a memory request over CCI-P *Channel 0 (C0)*, using *pck_af2cp_sTx.c0* and receives the response over *C0*, using *pck_cp2af_sRx.c0*. The response is 512bit long(1 cacheline) The memory read request is sent through the link when the *pck_af2cp_sTx.c0.valid* signal is set. A *header* field is *mdata* that is a user-defined request ID that is returned unmodified with the response. This field is usefull because the AFU developer can tag the requests and their responses and re-order them since the CCI-P can not guarantee the response order. Depending on the *header* field *cl_num* the number of cachelines that is fetched from the memory is determined. That number can be one, two or four and the cachelines are consecutive.

2.4.2 Write Requests

The write requests are sent over CCI-P *Channel 1 (C1)*, using *pck_af2cp_sTx.c1* and receive write completion acknowledgement responses over *C1*, using *pck_cp2af_sRx.c1*. The AFU drives the request with the *data* (512 bit) when the *pck_af2cp_sTx.c1.valid* is set. In this kind of requests again the *cl_num* field of the *header* determines the number of the cachelines will be written.

2.4.3 MMIO Reads

The MMIO read request is sent over *pck_cp2af_sRx.c0* when the *mmioRdValid* is asserted. The response is received via the *pck_af2cp_sTx.c2*. The data lengths that are supported us 4bytes and 8bytes.

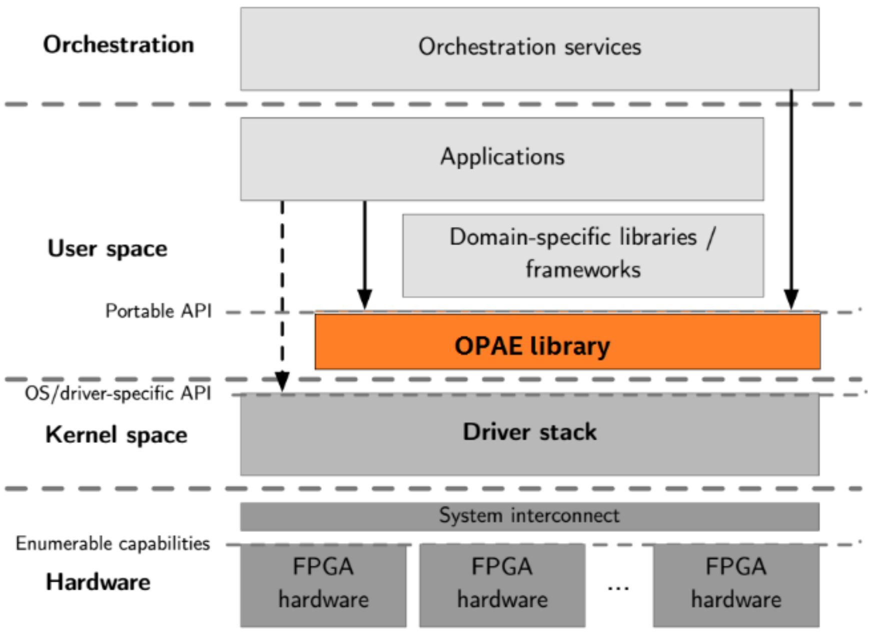
2.4.4 MMIO Writes

The AFU receives an MMIO write request over *pck_cp2af_sRx.c0*. The CCI-P asserts *mmioWr-Valid*. The MMIO write request is posted and no response is expected from the AFU. The data lengths supported are 4 bytes, 8 bytes, and 64 bytes.

2.5 OPAE C API

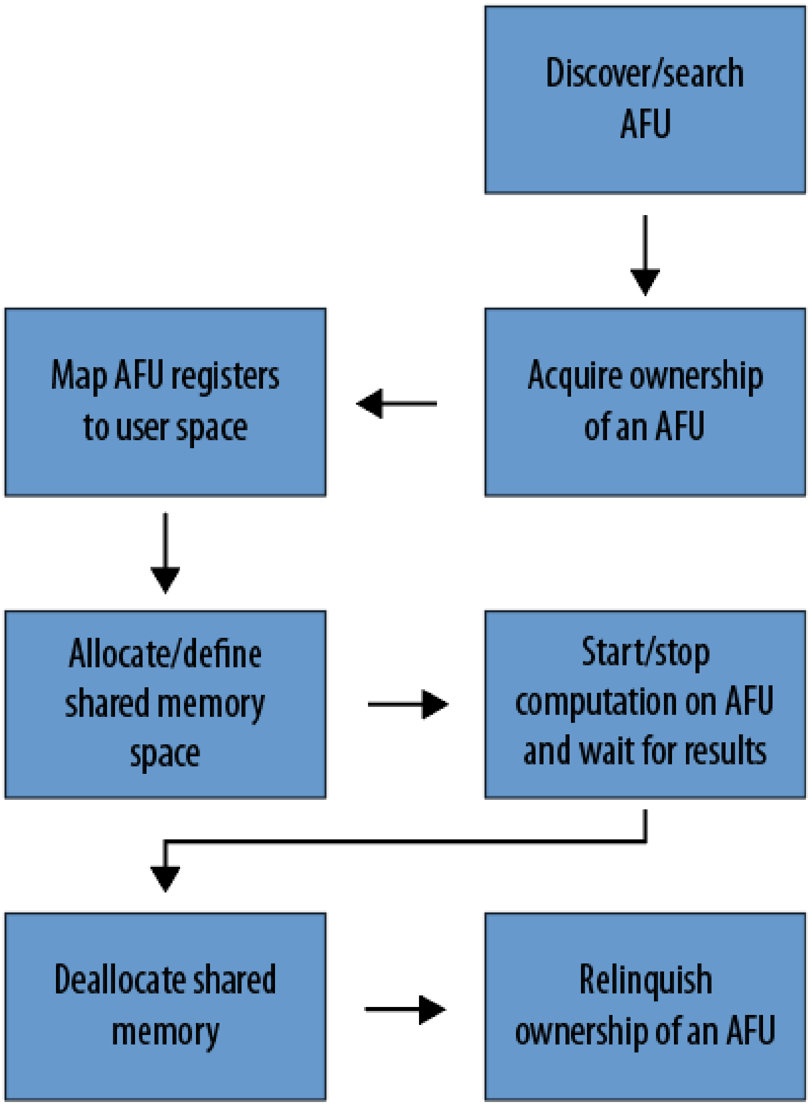
The OPAE[19] C library (libopae-c) is a lightweight user-space library that provides abstractions for FPGA resources in a compute environment. The OPAE C library builds on the driver stack that supports the FPGA device, abstracting hardware- and OS-specific details. It provides access to the underlying FPGA resources as a set of features available to software programs running on the host. These features include the acceleration logic preconfigured on the FPGA and functions to manage and reconfigure the FPGA. The library enables applications to transparently and seamlessly benefit from FPGA-based acceleration.

Figure 2.5: OPAEl



The figure below shows the basic application flow from the viewpoint of a user-process.

Figure 2.6: App Flow


Discover/search AFU
Acquire ownership of an AFU
Map AFU registers to user space
Allocate/define shared memory space
Start/stop computation on AFU and wait for results
Deallocate shared memory
Relinquish ownership of an AFU

2.6 OPAE AFU Simulation Environment (ASE)

ASE [11] aims to provide a consistent transaction level hardware interface and software API that allows users to develop production-quality Accelerator Functional Unit (AFU) RTL and software host application that can run on the real FPGA system without modifications.

Capabilities of ASE

- ASE provides a protocol checker that helps identify protocol correctness. ASE can rule-check if the Accelerator Functional Unit (AFU) complies to CCI-P protocol specifications, and whether OPAE API has been used correctly. It provides methods to identify potential issues early before in-system deployment.

- ASE can help identify certain lock conditions and Configuration/Status Registers (CSR) address mapping and pointer math mistakes.

- ASE presents a fake memory model to the AFU which keeps tabs on memory requested as accelerator workspaces, immediately flagging illegal memory transactions to locations outside of requested memory spaces. This is a good way to identify address violation bugs in simulation.

- ASE does not guarantee synthesizability of the AFU design.

- ASE provides a data hazard checker which can be used to warn users of CCI-P traffic patterns that may cause Write After Write (WAW), Read After Write (RAW), and Write After Read (WAR) hazards. These transactions may be debugged using the waveform viewer or by using a relevant Memory Protocol Factory (MPF) shim.

- ASE does not require administrator privileges needed to run, it is completely user-level. ASE may be run on a plain vanilla user Linux box with all the required tools installed

Limitations of ASE

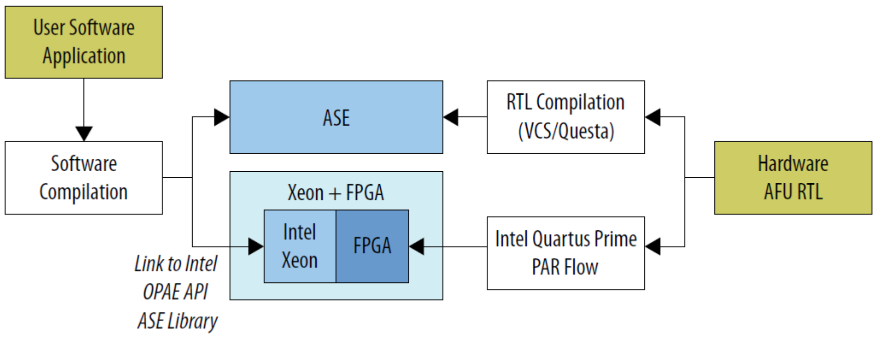
- ASE is a transaction level simulator for a single AFU slot and single application in platform with Intel FPGA IP. ASE does model either Intel UPI or PCIe specific packet structures and protocol layers.

- ASE does not simulate caching activities and is not a cache simulator. It cannot reliably simulate cache collision or capacity issues.

- ASE is designed to take an actual in-system AFU RTL and its corresponding OPAE software application and verify them for correctness. ASE cannot simulate a FPGA programming file.

- Although ASE models some latency parameters, it cannot model real-time system-specific latency behavior. It is also not intended to be a timing simulation of the design or latency/bandwidth profile of the real system, but good for functionally correct development.
- ASE does not simulate multi-AFU or multi-socket configurations.

Figure 2.7: Portability between simulation and Place and Route of Quartus Prime tool

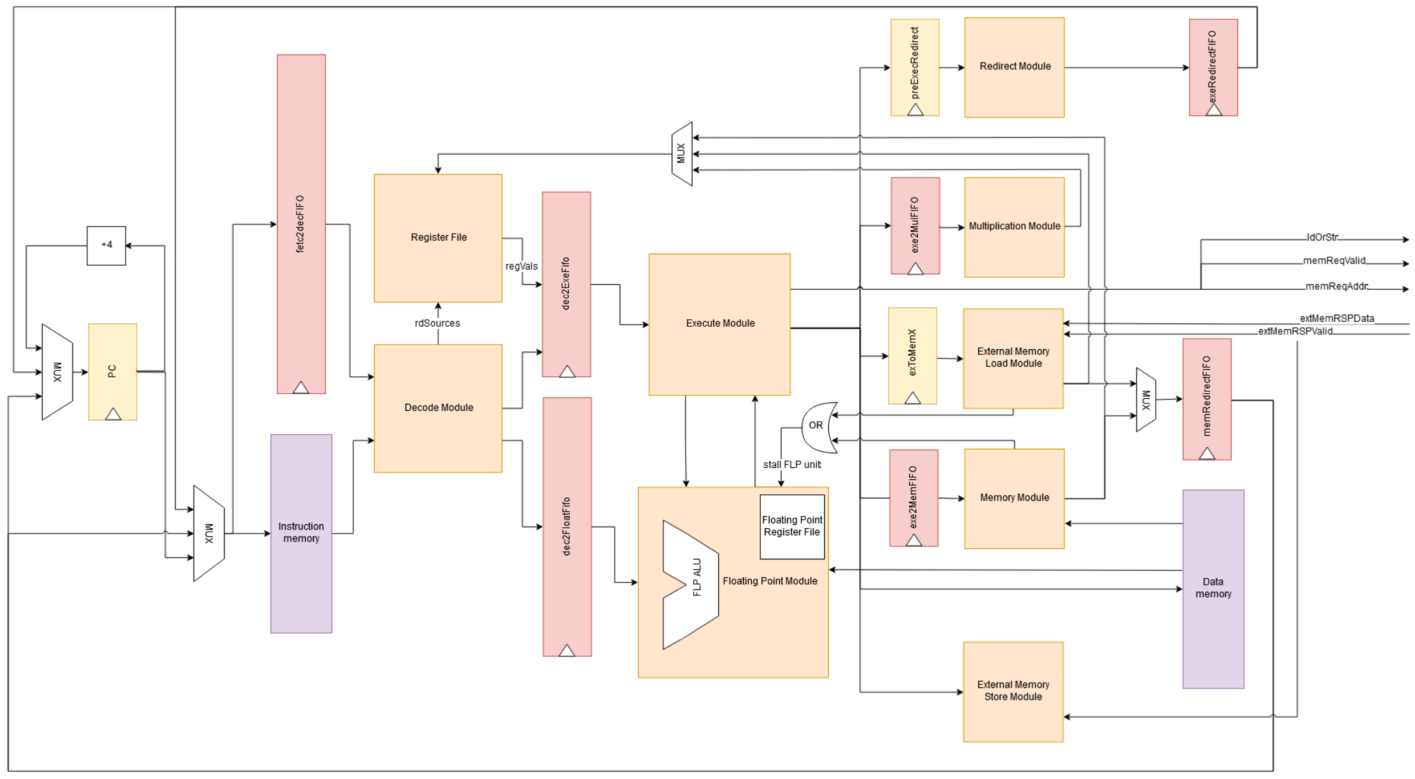

User Software Application
Software Compilation
Link to Intel OPAE API ASE Library
ASE
Xeon + FPGA
Intel Xeon
FPGA
RTL Compilation (VCS/Questa)
Intel Quartus Prime PAR Flow
Hardware AFU RTL

Chapter 3

ARM Core Implementation

The basic functionality of the ARM [3] core that was used in the current thesis is referred on [1]. In this work there were made some changes to improve the design and extend the functionality of the previous architecture. The core has 32-bit architecture, three stage pipeline and implements the majority of the armv4 instruction set. Furthermore the processor supports some basic single floating point (FP) instructions that will be analyzed in the sections below. Except the previous mentioned instructions there were implemented some custom instructions to help with the functionality of the whole system. The design was implemented using Bluespec System Verilog (BSV). In order for the processor to use Intel specific RAMs for the targeted FPGA a BSV wrapper was written that encapsulates altera_syncram IP core.

Figure 3.1: ARM Datapath



3.1 Overview of the ARM Core

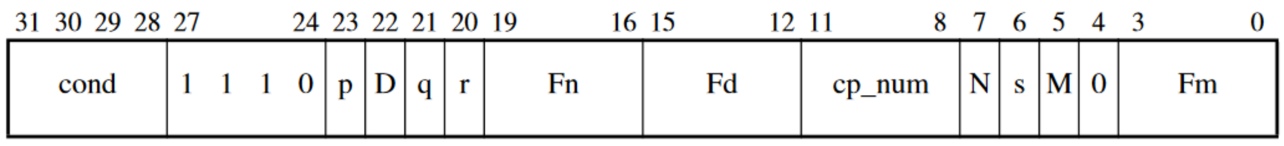
The three stages of the pipeline are Fetch, Decode a Execute. Some instructions ,(long multiplications, load from memory) need four cycles, so an extra pipeline stage. At the Fetch stage, the instruction is fetched from the instructions memory and it is propagated along with the value of the program counter (PC) to the next stage with a pipeline FIFO. The Decode stage, decodes the instructions and determines if the instruction is a normal instruction or a FP one. Also at this stage the values of the proper registers are read from the register file for the normal instructions. If the instruction is normal the values of the registers and the decoded instructions are fed on a pipeline FIFO and move on to the next stage. On the other hand if the instructions is identified as a floating point one, the decoded instruction is moved on a module that it could be characterized as a co-processor to start the execution. The Floating Point Co-processor (FPC) can work in parallel with rest of the pipeline with some limitations. The FPC has a FP register file, 32 registers wide, to store the results of the FP instructions. This register file has four read ports and two write ports. The three read ports and the 1 write port are used from the co-prossesor and the rest of them from the main pipeline. The main core and the co-processor are autonomous to some point. A normal and a floating point instruction can execute in parallel. For example, if a FP instructions is executed at the co-processor the main pipeline can execute normal instructions. Although if the co-processor is busy and the next instruction in line is a FP one, then the core stalls until the co-processor is done with the execution of the previous instruction. Below will be discussed only the instruction that have been implemented and not the whole FP instructions set of the ARM processor.

3.2 Floating point Instruction Set

The FP instructions are sectioned to three major categories: Data-processing instructions, Load and Store instructions and Single register transfer.

3.2.1 Data-processing instructions

Figure 3.2: Data-processing instructions format



cond: The *cond* field indicates if the instruction will be executed based on the Current Program Status Register(CPSR) flags

p,q,r,s: These bits collectively form the instruction's primary opcode. When all of p, q, r and s are 1, the instruction is a two-operand extension instruction, with an extension opcode specified by the Fn and N bits.

Fd and D: These bits specify the destination register of the instruction. Fd holds the top 4 bits of the register number and D holds the bottom bit.

Fn and N: These bits specify the destination register of the instruction. Fn holds the top 4 bits of the register number and N holds the bottom bit.

FM and M: These bits specify the destination register of the instruction. Fm holds the top 4 bits of the register number and M holds the bottom bit.

cp_num: *cp_num* is set to 4'b1010 when the instructions is single precision FP. For double precision the *cp_num* 4'b1011.

Table 3.1: Data-processing instructions

p q r s	**Instruction name**	**Instructions functionality**
0 0 0 0	FMACS	Fd = Fd + (Fn * Fm)
0 0 0 1	FNMACS	Fd = Fd - (Fn * Fm)
0 0 1 0	FMSCS	Fd = -Fd + (Fn * Fm)
0 0 1 1	FNMSCS	Fd = -Fd - (Fn * Fm)
0 1 0 0	FMULS	Fd = Fn * Fm
0 1 0 1	FNMULS	Fd = -(Fn * Fm)
0 1 1 0	FADDS	Fd = Fn + Fm
0 1 1 1	FSUBS	Fd = Fn - Fm
1 0 0 0	FDIVS	Fd = Fn / Fm
1 0 0 1	-	UNDEFINED
1 0 1 0	-	UNDEFINED
1 0 1 1	-	UNDEFINED
1 1 0 0	-	UNDEFINED
1 1 0 1	-	UNDEFINED
1 1 1 0	-	UNDEFINED
1 1 1 1	-	Extension instructions

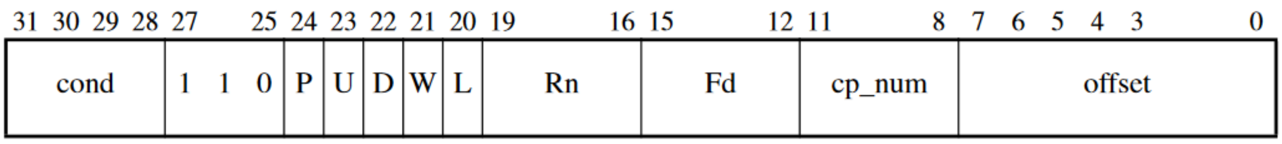
Table 3.2: Extension instructions

FN	N	**Instruction name**	**Instructions functionality**
0000	0	FCPYS	Fd = Fm
0000	1	FABSS	Fd = abs(Fm)
0001	0	FNEGS	Fd = -Fm
0001	1	FSQRTS	Fd = sqrt(Fm)
001x	x	-	UNDEFINED
0100	0	FCMPS	Compare Fd with Fm, no exceptions on quiet NaNs
0100	1	FCMPES	Compare Fd with Fm, with exceptions on quiet NaNs
0101	0	FCMPZS	Compare Fd with 0, no exceptions on quiet NaNs
0101	1	FCMPEZS	Compare Fd with 0, with exceptions on quiet NaNs
011x	x	-	UNDEFINED
1000	0	-	UNDEFINED
1000	1	FSITOS	Signed integer floating-point conversions
1001	x	-	UNDEFINED
101x	x	-	UNDEFINED
1100	x	-	UNDEFINED
1101	0	FTOSIS	Floating-point signed integer conversions
1101	1	FTOSIZS	Floating-point signed integer conversions, RZ mode
111x	x	-	UNDEFINED

The total of Data-Processing Instructions that were implemented are seen in Tables 3.1 and 3.2.

3.2.2 Load and Store instructions

Figure 3.3: Load and store instructions format



The two instructions that were implemented is FSTS and FLDS. FSTS stores one single-precision register to memory and FLDS loads one single-precision register from memory.

P,U,W: These bits specify an addressing mode of the load or store instruction. The addressing can be pre-index, post-index or unindexed. The offset can be added of subtracted from the base register and the memory address can be written to the base register or remain unchanged.

Fd and D: These bits specify the destination floating-point register of a load instruction, or the source floating-point register of a store instruction. Fd holds the top 4 bits of the register number and D holds the bottom bit.

L bit: This bit determines whether the instruction is a load (L == 1) or a store (L == 0).

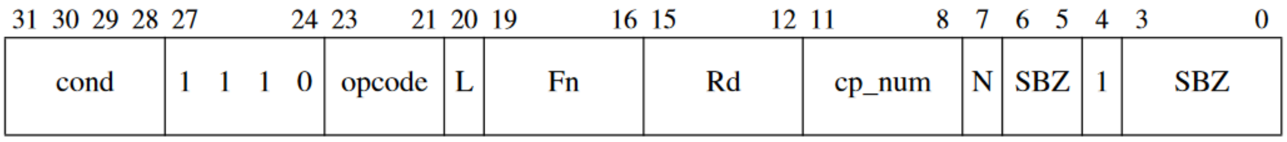
Rn: This specifies the ARM register used as the base register for the address calculation

cp_num: *cp_num* is set to 4'b1010 when the instructions is single precision FP. For double precision the *cp_num* 4'b1011.

offset These bits specify the word offset which is applied to the base register value to obtain the starting memory address for the transfer.

3.2.3 Single register transfer instructions

Figure 3.4: Single register transfer instructions



31 30 29 28	27 24	23 21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	1 1 1 0	opcode	L	Fn	Rd	cp_num	N	SBZ	1	SBZ

The two instructions that were implemented is FMSR and FMRS. FMSR transfers a FP register to an ARM register and FMRS transfers an ARM register to an FP register.

opcode: Is 3'b000 for FMSR and FMRS

L bit: This bit determines the direction of the transfer: L==0 from and ARM register to a FP register (FMRS). L==1 from a FP register to an ARM register (FMSR).

Fn and N: These bits specify the VFP register involved in the transfer. Fn holds the top 4 bits of the register number and N holds the bottom bit.

Rd: This specifies the ARM register involved in the transfer. If Rd is R15, the behaviour is unpredictable

cp_num: *cp_num* is set to 4'b1010 when the instructions is single precision FP. For double precision the *cp_num* 4'b1011.

3.2.4 Custom instruction

There were implemented some custom instructions that have a contribution to the functionality of the whole system.

Load or Store on address greater than 0x400 The loads and stores can be either for ARM of FP registers. If the address of the load or store instruction is greater than 0x400 then the request does not go to the data memory of the ARM but it is served firstly from a cache memory and secondly from the main memory of Xeon. More details for the cache will be discussed in later chapters.

Load or Store on address 0x400 If the address of a load or store instruction is 0x400 then the cache memory is being flashed to the main memory of the Xeon

software interrupt (SVC) When the **SVC** instruction is executed the processor enters a terminal state. That state can change only with a reset to the processor.

3.3 Implementation of data-processing instructions

In order to implement these FP instructions the **Floating Point Hardware 2 Component**(FPH2)[18] package of Intel was used. The use of the package was necessary because of the need of compatibility between FP instructions and the targeted FPGA. **Floating Point Hardware 2** packages all the floating point functions in a single component. This component consists of compinational and multi-cycle FP instructions. This IP core is primarily used by the **Nios II** which is a 32-bit embedded-processor specifically designed for Intel FPGAs.

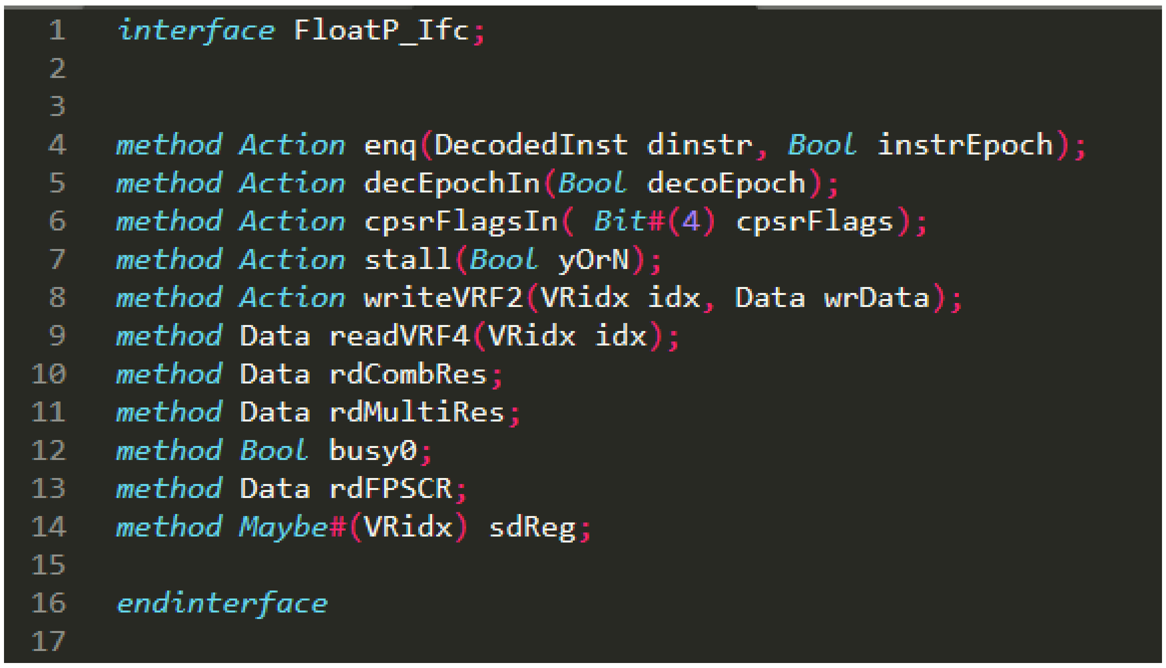
In Table 3.3 are shown the cycles for the execution of every data-processing instruction and the rounding of the results. The cycles are counted from the moment the instruction inserts to the FPC until the result is written to FP register file The **FPH2** component also supports faithful rounding, which is not an IEEE 754-defined rounding mode. Faithful rounding rounds results to either the upper or lower nearest single-precision numbers. Therefore, the result produced is one of two possible values and the choice between the two is not defined. The maximum error of faithful rounding is 1 unit in the last place (ulp). Errors may not be evenly distributed.

In order to use the existing IP core to the processors design, a mechanism of BSV was used. "Import BVI" is a feature of BSV that allows the developer to import an existing Verilog or VHDL block on a BSV design. Technically this feature helps the developer to write a BSV wrapper for the existing block and inform the compiler how to threat it.

Table 3.3: Data processing instructions execution cycles and rounding

Instruction name	Cycles	Rounding
FMACS	9	Faithful
FNMACS	9	Faithful
FMSCS	9	Faithful
FNMSCS	9	Faithful
FMULS	4	Faithful
FNMULS	5	Faithful
FADDS	5	Faithful
FSUBS	5	Faithful
FDIVS	16	Nearest
FCPYS	1	None
FABSS	1	None
FNEGS	1	None
FSQRTS	8	None
FCMPS	1	None
FCMPES	1	None
FCMPZS	1	None
FCMPEZS	1	None
FSITOS	4	Not applicable
FTOSIS	2	Truncation (Round to Zero)
FTOSIZS	2	Truncation (Round to Zero)

Figure 3.5: Floating Point Co-processor interface



```
interface FloatP_Ifc;


method Action enq(DecodedInst dinstr, Bool instrEpoch);
method Action decEpochIn(Bool decoEpoch);
method Action cpsrFlagsIn( Bit#(4) cpsrFlags);
method Action stall(Bool yOrN);
method Action writeVRF2(VRidx idx, Data wrData);
method Data readVRF4(VRidx idx);
method Data rdCombRes;
method Data rdMultiRes;
method Bool busy0;
method Data rdFPSCR;
method Maybe#(VRidx) sdReg;

endinterface

```

In figure 3.5 is presented the interface of **FPC**.

enq(DecodeInst dinstr, Bool instrEpoch)) With this method the decoded instructions and its epoch are inserted into the co-processor. The epoch of the instruction will be compared with the epoch of the pipeline. If those two match then the instruction is able to be executed from the co-processor or else it is discarded.

method Action decEpochIn(Bool decEpoch) The decEpoch is the current epoch of the pipeline. The epoch of the pipeline is changed whenever a taken branch instruction is executed.

cpsrFlagsIn(Bit#(4) cpsrFlags This is the four most significant bits. These bits denote and the condition flags of the instruction denotes if the instructions will be executed.

stall A signal for stalling the co-processor. A stall can occur if a FP instruction is in the co-processor and in the main pipeline is a executed an load instruction that affects a FP register. The load instruction needs one more cycle to write the result back to register when the execution of a FP instruction starts two stages back on the pipeline. So there is a need to stall the co-processor until the value that is being loaded from the memory is written to the FP register file.

writeVRF2(VRidx idx, Data wrData) This is a write port to the FP register file that is accessed from the main pipeline(Execute of Write Back stage)

Data readVRF4(VRidx idx) This is the output of the register file, which returns the a 32bit value.

Data rdCombRes This output has the compinational result of the FPH2.

Data rdMultiRes This output has the multi-cycle result of the FPH2.

Bool busy0 This informs the main pipeline if the co-processor is currently executing another instruction

Data rdFPSCR This output returns the value of the FPSR register

Maybe#(VRidx) sdReg The output has the id number of the destination register of the FP instruction.

3.4 Implementation of Load and Store instructions

The FLDS and FSTS instructions for FP registers are identical to the load and store instructions for the ARM registers. The difference lies between the registers that are taking part to the instructions. These instructions are executed not by the FPC but from the main pipeline. The necessary read and writes to the FP register file is done by the ports of the FPC. If there is another instruction on FPC prior to FLDS or FSTS then the instruction is stalled until the FPC finishes.

3.5 Implementation of single register transfer instructions

The FMSR and FMRS instructions are executed also by the main pipeline and not from the FPC. The necessary read and writes to the FP register file is done by the ports of the FPC. If there is another instruction on FPC prior to FMSR or FMRS then the instruction is stalled until the FPC finishes.

3.6 Implementation of custom instructions

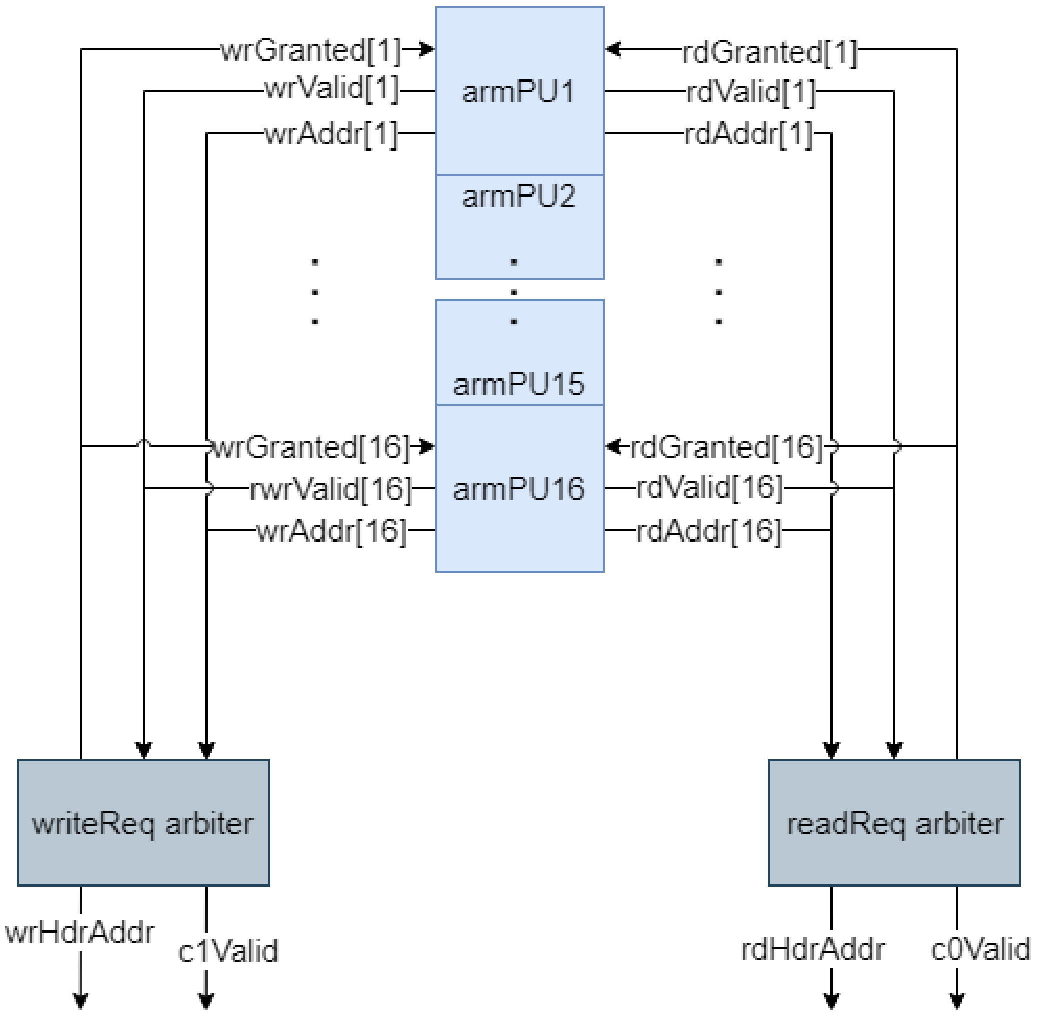
As mentioned before SVC instruction leads the processor to a terminal state and from that state can change only with reset. The implementation for this instruction is plain simple. When the incoming instructions are decoded as SVC then the value of state register is changed to the proper value. Concerning the load and store instruction to the main memory of the platform the procedure is as follows. Firstly, the instructions are decoded and if the request is not for the local data memory, then the processor issues a request with proper address, data and valid signal. Afterwards the processor enters a wait state and the pipeline is stalled until a valid response comes from the cache or the main memory. The response may contain data if the instruction was a load or only a valid signal for store instruction, which suggests that the data were stored on the cache or the main memory.

Chapter 4

Hardware and Software Implementation

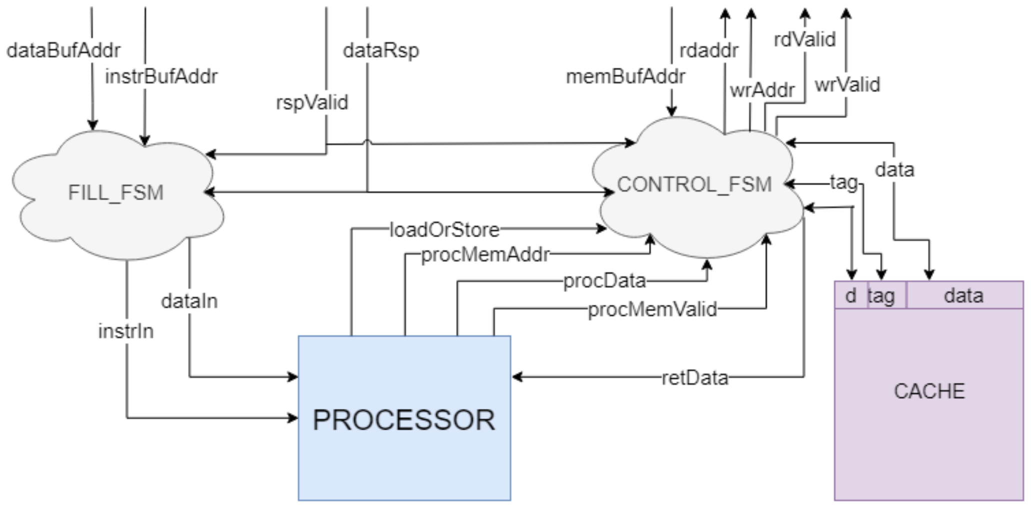
In this section the structural components of the whole system, for the hardware and the software side be analyzed. The system consists of 16 ARM processors and has a direct-map cache memory for every core which can be of any size of power of two. The width of the data of the cache is 512 bit (64B) as that is the size of the cacheline that the platform returns with a single read request. The last two bits of the address that comes from the processor is always zero because the addresses are multiple of four. The next four bits is the block offset and the rest bits are the tag and the index which depend on the size of the cache. There is a control Finite State Machine(FSM) that controls the the read and the writes to and from cache and also issues requests for the Xeon memory with the proper data. Except that there are FSMs that load the instruction and data memory of every core independently. The software(sw) side decide which and how many of the cores will be started or be reset. In addition software can tell if the cores memories will be loaded or cleared or stay as it is with the pre-existing data that have (from synthesis of from a previous run). The necessary information for a run is passed from sw to hw through MMIO requests. With these requests the AFU becomes aware of the addresses of the buffer that will read and write. Except that AFU can know how many instructions and data will be loaded to the internal memories of the cores. Also the cores receive orders from the sw with such requests for example when to start/reset, load or not instructions and data memory. Furthermore, in order to manage the multiple read and write requests from the cores two arbiters were implemented, one that handles the read requests and one that handles the write requests. The read and write requests are independent to each other.

Figure 4.1: Abstract view of system's top module



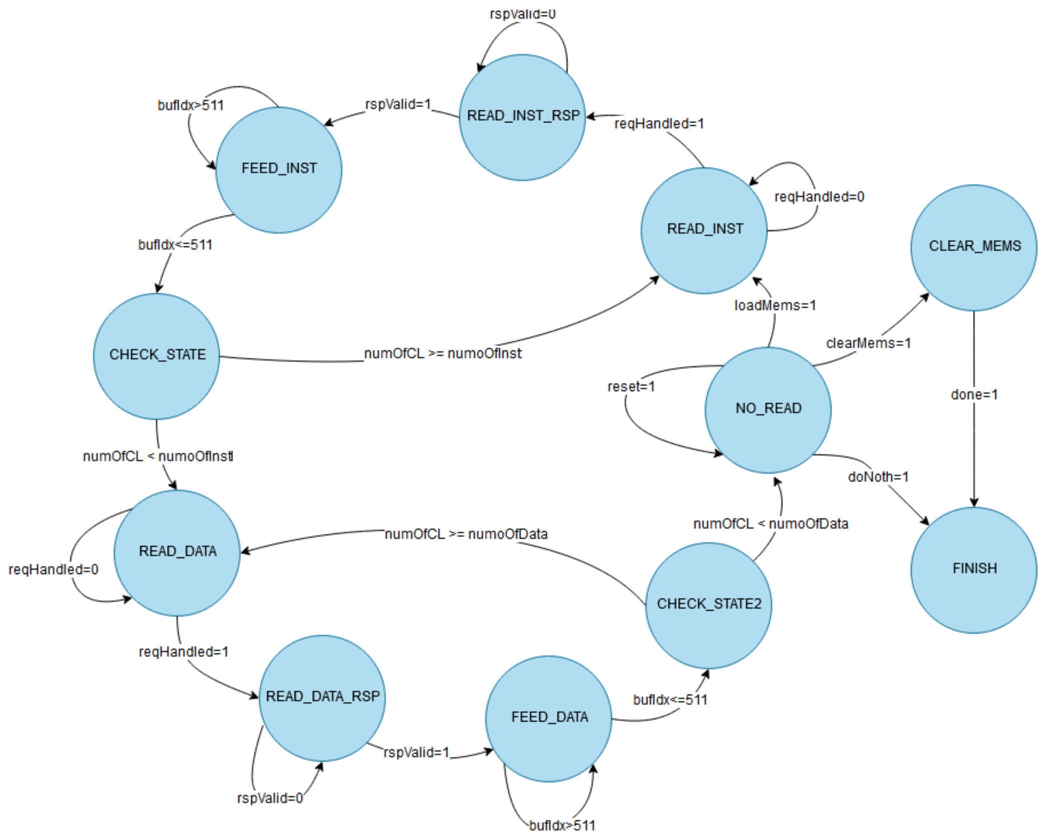
4.1 Hardware Implementation

Figure 4.2: Block diagram of ARM with cache control for read/write requests and control for writing to instructions and data memory



4.1.1 FSM for loading instruction and data memory

Figure 4.3: State diagram FSM for loading instruction and data memory



The states of the FSM in Figure 4.3 will be explained below.

NO_READ: This state is the initial state of the FSM. FSM stays on this state until a start singal comes along with signal that suggests whats is the next state.

READ_INST: If a load memory signal is set then after the NO_READ the state becomes READ_INST. On this state a read request to the proper buffer that keeps the instructions code is prepared.

READ_INST_RSP: The FSM stays on this state until gets a valid response from the memory. Every response data contains sixteen instructions

FEED_INST: After the valid response the FSM feeds the instruction one by one every clock cycle to the instruction memory of the processor. The feeding last sixteen cycles

CHECK_STATE: Then the total number of the instructions is checked. If there are more instructions to be loaded then the FSM goes again to READ_INST to repeat the process and fetch the next instructions.

READ_DATA: When there are no more instructions to be loaded to the memory, FSM issues a request for the buffer on Xeon's memory that keeps the data for the data memory of the processor.

READ_DATA_RSP: Again the FSM stays on this state until it gets a valid response. Every response contains sixteen 32bit data.

FEED_DATA: When FSM gets the response then feeds the data to the data memory of the processor. That lasts sixteen cycles.

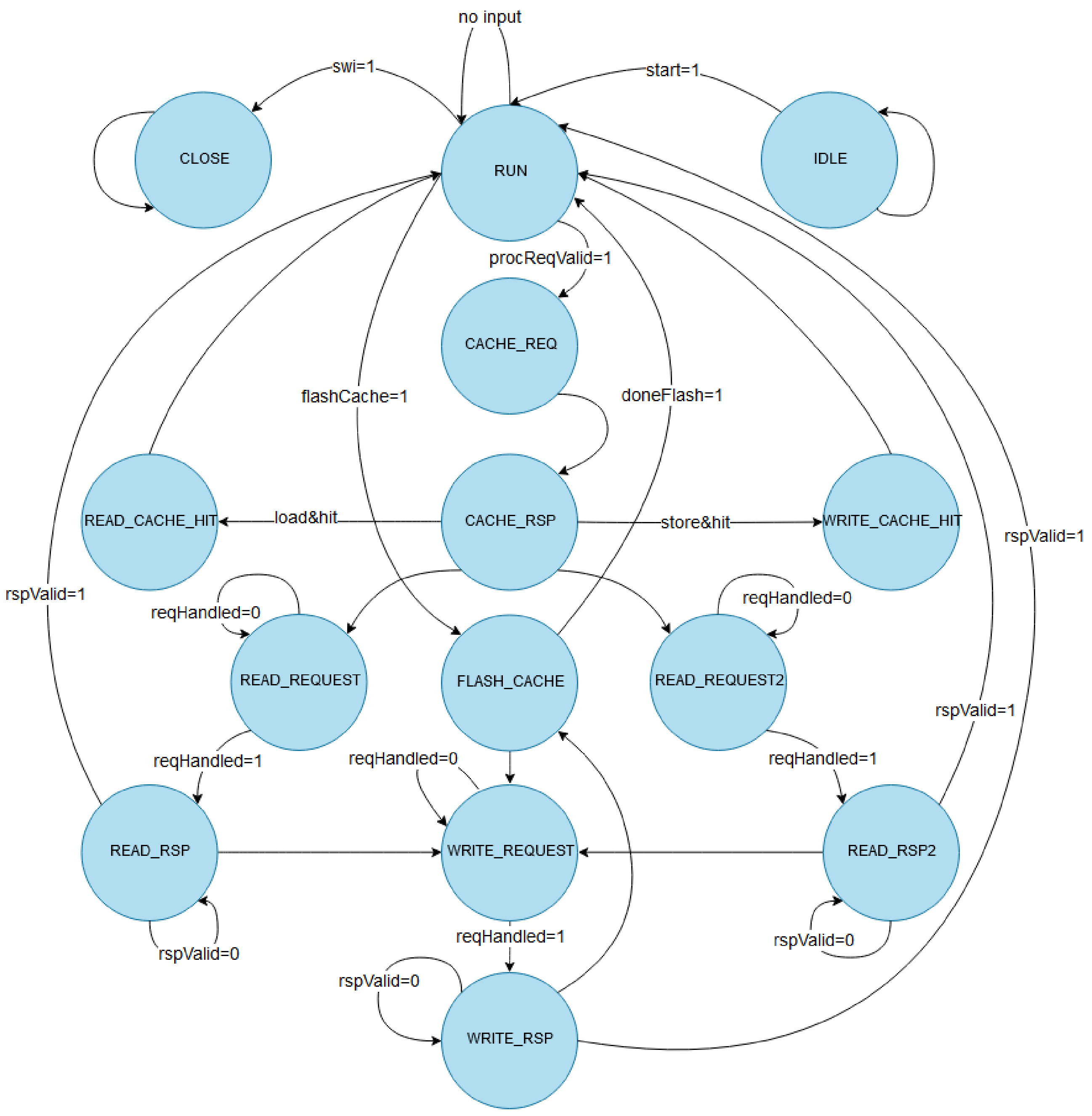
CHECK_STATE2: Afterwards the FSM checks if there are more data to be fetched from the memory. If there are, then the FSM goes to READ_DATA state else it goes to FINISH state.

CLEAR_MEMS: If the FSM enters to this state then zeros will be written in every location.

FINISH: This is the terminal state for this FSM.

4.1.2 FSM for controlling the cache and the read/write requests to the platforms memory

Figure 4.4: State diagram FSM for controlling the cache and issuing read/write requests for external memory



$\boxed{\textbf{IDLE}}$**:** This is the initial state of the FSM. To start this FSM the previous described FSM must be at FINISH state.

$\boxed{\textbf{RUN}}$**:** On this state the FSM waits for a read or write requests from the processor. Depending on the request FSM chooses the next state.

$\boxed{\textbf{CACHE\_REQ}}$**:** If there is a read/write the FSM issues a read request for the cache to get the tag and the data from it.

$\boxed{\textbf{CACHE\_RESP}}$**:** The response from the cache is composed from the data and the tag. The tag is compared with the proper bits of the address from the processor and accordingly the next state is chosen. If the request from the processor is for load and the tag is in the cache then the next state is READ_CACHE_HIT, else the next state is READ_REQUEST. Otherwise if the request is for store data and the tag is in the cache then the next state is WRITE_CACHE_HIT else it is READ_REQUEST2.

$\boxed{\textbf{READ\_CACHE\_HIT}}$**:** Based on the index bits of the address, the proper 32bit of the cache data are selected to return as a response to the processor.

$\boxed{\textbf{READ\_REQUEST}}$**:** On this state the FSM issues a read request to the main memory. The address of the request is the top 26bits of the address that came from the processor. If the request is handled from the arbiter then the FSM goes to READ_RSP state.

$\boxed{\textbf{READ\_RSP}}$**:** The FSM waits for the main memory to respond with the proper data. If the location in which the new data will be written is characterized as dirty then that data must be written back to the main memory, so the next state will be WRITE_REQUEST. If the previous data was not dirty then the next state will be RUN.

$\boxed{\textbf{WRITE\_CACHE\_HIT}}$**:** If the tag of the cache matches the tag bits of the address then the cacheline exists in the cache, therefore the based on the index bits the proper 32bit is altered with the data that came from the processor

$\boxed{\textbf{READ\_REQUEST2}}$**:** In the case that the cacheline does not exist on the cache a read request is made in order for the 64B in which the write will be made. When the read request is served from the arbiter the state is changed to READ_RSP2

$\boxed{\textbf{READ\_RSP2}}$**:** FSM waits the response from the main memory. If the location of the cache in which the new data will be written is dirty the old data must be written back to the main memory. From the 512 new bits the proper 32bit, according to the index bits, is altered to the data that came from the processor.

$\boxed{\textbf{WRITE\_REQUEST}}$**:** In this state a write request is issued for the main memory. When the arbiter handles the request, the state changes to WRITE_RSP.

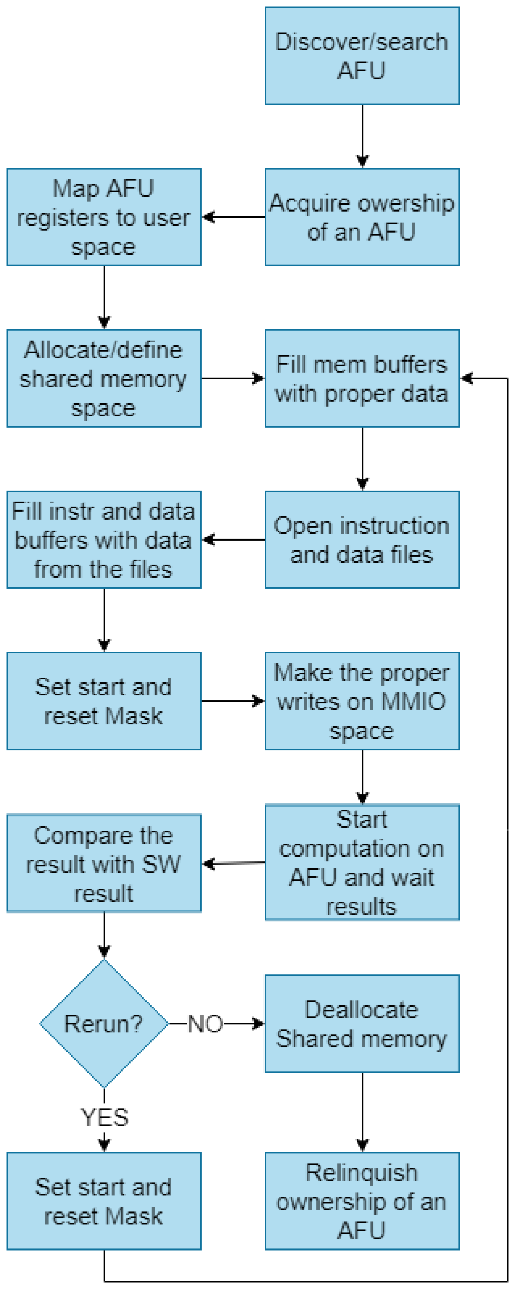
$\boxed{\textbf{WRITE\_RSP}}$**:** FSM waits a valid signal from the main memory. This signal suggests that the request is served.

$\boxed{\textbf{FLASH\_CACHE}}$**:** In this state the FSM prepares the data for the write requests that will flash the whole cache back to main memory. From this state the next state is WRITE_REQUEST this is done as many times as the cache size.

$\boxed{\textbf{CLOSE}}$**:** This is the terminal state of the FSM. The FSM enters to this state if a software interrupt occurred on the processor

4.2 Software implementation

Figure 4.5: Application flow of the software application


Discover/search AFU
Acquire owership of an AFU
Map AFU registers to user space
Allocate/define shared memory space
Fill mem buffers with proper data
Open instruction and data files
Fill instr and data buffers with data from the files
Set start and reset Mask
Make the proper writes on MMIO space
Start computation on AFU and wait results
Compare the result with SW result
Rerun?
NO
YES
Deallocate Shared memory
Set start and reset Mask
Relinquish ownership of an AFU

As seen in Figure 4.5 at first sw is searching for an AFU to connect and if the IDs are matching it will connect to it. The ID is read from the json file of the project for the sw side and it is hardcoded in the AFU. Afterwards the AFU registers is mapped in the user space and memory space for the shared buffers is allocated. This buffers have all the data for the computations, the instructions and proper starting data for the internal memories of the processors. The buffers for the computation are filled with random numbers and the data for the instruction and data buffers are read from binary files. Then the reset and start mask is set by the user. This masks define which cores will be reset and then start to execute. All the necessary information are passed from sw to the AFU with write on the MMIO address space. This MMIO write requests contain the address for the buffers of each processor(memory, instruction and internal data buffers), start and reset mask, the number of instructions and data that will be loaded and and what action will the processors perform to their internal memories. The sw triggers the accelerator to start with a write on a special register and then waits for a write on the same register to continue. The code that the ARM processors is executing in the FPGA is also executed on the Xeon in order to verify that the system is working properly. After the comparison and the evaluation of the results user can restart the system by setting again the reset and start mask and perform a MMIO write to inform the AFU. If all processes are done the sw deallocates the shared memory releases the AFU and the program is terminated.

Below the most important functions of the OPAE library will be discussed.

fpgaGetProperties(fpga_token token, fpga_properties *prop) Initializes the memory pointed at by prop to represent a properties object, and populates it with the properties of the resource referred to by token. Individual properties can then be queried using fpgaPropertiesGet × () accessor functions.

fpgaPropertiesSetObjectType(fpga_properties prop, fpga_objtype objtype) Set the object type of a resource.

fpgaPropertiesSetGUID(fpga_properties prop, fpga_guid guid) Sets the GUID of an FPGA or accelerator object. For an accelerator, the GUID uniquely identifies a specific accelerator context type, i.e. different accelerators will have different GUIDs. For an FPGA, the GUID is used to identify a certain instance of an FPGA, e.g. to determine whether a given bitstream would be compatible.

fpgaEnumerate(const fpga_properties *filters, uint32_t num_filters, fpga_token *tokens, uint32_t max_tokens, uint32_t *num_matches) This call allows the user to query the system for FPGA resources that match a certain set of criteria, e.g. all accelerators that are assigned to a host interface and available, all FPGAs of a specific type, etc.
fpgaEnumerate() will create a number of fpga_tokens to represent the matching resources and populate the array tokens with these tokens. The max_tokens argument can be used to limit the number of tokens allocated/returned by fpgaEnumerate(); i.e., the number of tokens in the returned tokens array will be either max_tokens or num_matches (the number of resources matching the filter), whichever is smaller.

fpgaDestroyProperties(fpga_properties *prop) Destroys an existing fpga_properties object that the caller has previously created using fpgaGetProperties()

fpgaOpen(fpga_token token, fpga_handle *handle, int flags) Acquires ownership of the FPGA resource referred to by token. Most often this will be used to open an accelerator object to directly interact with an accelerator function, or to open an FPGA object to perform management functions.

fpgaDestroyToken(fpga_token *token) This function destroys a token created by fpgaEnumerate() and frees the associated memory.

fpgaPrepareBuffer(fpga_handle handle, uint64_t len, void **buf_addr, uint64_t *wsid, int flags) Prepares a memory buffer for shared access between an accelerator and the calling process. This may either include allocation of physical memory, or preparation of already allocated memory for sharing. This function will ask the driver to pin the indicated memory (make it non-swappable), and program the IOMMU to allow access from the accelerator. If the buffer was not pre-allocated the function will also allocate physical memory of the requested size and map the memory into the callers process virtual address space. It returns in wsid an fpga_buffer object that can be used to program address registers in the accelerator for shared access to the memory.

fpgaGetIOAddress(fpga_handle handle, uint64_t wsid, uint64_t *ioaddr) This function is used to acquire the physical base address (on some platforms called IO Virtual Address or IOVA) for a shared buffer identified by wsid.

fpgaReset(fpga_handle) handle Performs an accelerator reset.

fpgaWriteMMIO64(fpga_handle handle, uint32_t mmio_num, uint64_t offset, uint64_t value) This function will write to MMIO space of the target object at a specified offset.

fpgaReadMMIO64(fpga_handle handle, uint32_t mmio_num, uint64_t offset, uint64_t *value) This function will read from MMIO space of the target object at a specified offset.

fpgaReleaseBuffer(fpga_handle handle, uint64_t wsid) Releases a previously prepared shared buffer. If the buffer was allocated using fpgaPrepareBuffer , this call will deallocate/free that memory. Otherwise, it will only be returned to its previous state (pinned/unpinned, cached/non-cached).
The largest cache size that can be synthesized for an architecture of 16 cores is 2048. The total cache memory will be 2048 * 64B = 128kB

Chapter 5

Testing and Results

The tools that were used for synthesis,simulations verification of the design for purposes of this thesis were provided by Intel. The whole compute environment ACE (Intel Academic Compute Environment) is located on vLabs of Intel.

In this chapter will be discussed the different designs that were synthesized and the results of the various benchmarks that were acquired by evaluating the architecture.

All the synthesized designs have sixteen cores and the difference between them is only the cache size and the RAM type that was used to implement the cache memory . The RAM types are two: LUT RAM (MLAB for Intel FPGAs) and RAM made by dedicate blocks of memory resources (M20K). The max synthesizable cache size for MLAB is 16KB and for M20K is 128KB. The achieved clock is 69Mhz for all the designs. For the purpose of measuring the execution time of the designs a simple hardware counter was implemented that starts counter when the accelerator is triggered and stop when all the computations are finished. Because of that variable size of the cache, the different RAM types and in order to have a global view of the affects of the different cache sizes, there were implemented the following designs as seen on Tables 5.1 5.2.

Table 5.1: Different designs with different cache size and resource usage with MLAB RAMs

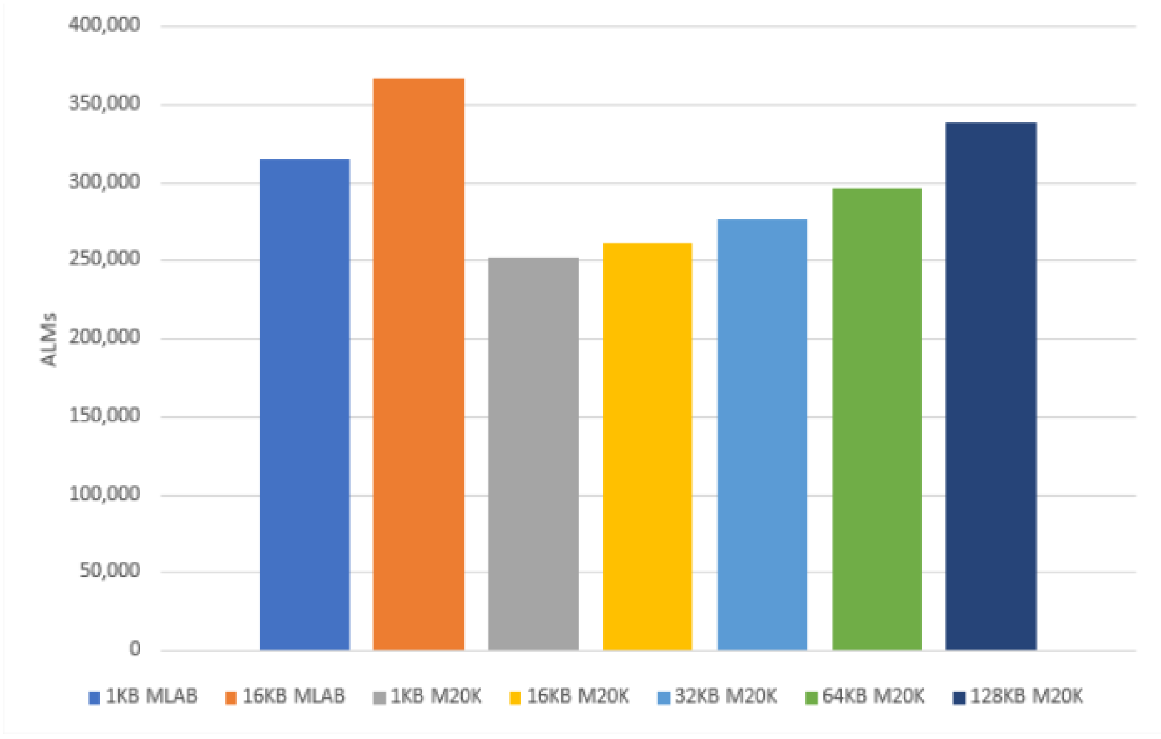
	Cache Size				**Total resources**
	1KB		**16KB**		-
	# used	**% used**	**# used**	**% used**	
ALMs	314,784	74%	366,986	86%	427,200
Registers	278549	-	308869	-	-
Block mem bits	2,265,488	4%	2,387,728	4%	55,562,240
RAM Blocks	409	17%	449	17%	2,713
DSP Blocks	208	14%	208	14%	1,518

Table 5.2: Different designs with different cache size and resource usage with M20K

	Cache Size										**Total resources**
	1KB		**16KB**		**32KB**		**64KB**		**128KB**		-
	# used	**% used**	# used	**% used**	# used	**% used**	# used	**% used**	# used	**% used**	
ALMs	252,074	**59%**	261,757	**61%**	276,734	**65%**	296,306	**69%**	338,417	**79%**	427,200
Registers	181463	**-**	185496	**-**	189641	**-**	197881	**-**	214282	**-**	-
Block mem bits	2,524,944	**5%**	4,590,864	**8%**	6,794,512	**12%**	11,201,808	**20%**	20,016,912	**36%**	55,562,240
RAM Blocks	657	**24%**	657	**24%**	657	**24%**	865	**32%**	1,313	**48%**	2,713
DSP Blocks	208	**14%**	208	**14%**	208	**14%**	208	**14%**	208	**14%**	1,518

As seen on Table 5.2 there after increasing the cache size in all designs there is a 2-4% increase on the ALMS except the transition from the 64KB cache to 128KB cache in which the rate increase is 10%. These numbers are including the static and the PR region of the FPGA. According to the documentation the available ALMs for the Accelerator Functional Unit (AFU) is 391,213 (92% of device total). However, a simpler design as the Hello world example, that Intel provides,uses around 70,000 ALMs. The Hello world design just sends a bit sequence from the hardware to the software side.

Figure 5.1: Diagram of ALM usage per cache size and RAM type



In Figure 5.1 becomes clear that the MLAB RAMs are more expensive in resources than MK20K. For example the design with the 128kB cache and M20K RAMs uses less ALMS than the design with 16 KB cache and MLAB RAMs

5.1 Read/Write Bandwidth of the platform

In this section will be presented the read and write bandwidth of the platform. This bandwidth was measured without the ARM accelerator. Additionally there were used some cache directives in order to check the impact that they have on performance of the system. As referred earlier developer can choose which channel to use for read and write requests independently . VA channel use and the three physical channels and redirects the requests on QPI, PCIe0 or PCIe1 base on the traffic of the channels. The sum of the bandwidth of the three physical channels is the total bandwidth of channel VA. Correspondingly VL0 channel uses QPI, VH0 uses PCIe0 and VH1 uses PCIe1. Therefore the max bandwidth of each virtual channel depends on the bandwidth of the corresponding physical channel.

Read Channel cache directives

- rdline-I: Read Line Invalid. Memory Read Request, with FPGA cache hint set to invalid. The line is not cached in the FPGA, but may cause FPGA cache pollution.
- rdLine-S: Read Line Shared. Memory read request with FPGA cache hint set to shared. An attempt is made to keep it in the FPGA cache in a shared state.

Write Channel cache directives

- wrLine-I: Write Line Invalid. Memory Write Request, with FPGA cache hint set to Invalid. The FIU writes the data with no intention of keeping the data in FPGA cache.
- wrLine-M: Write Line Modified. Memory Write Request, with the FPGA cache hint set to Modified. The FIU writes the data and leaves it in the FPGA cache in a modified state.
- wrpush-I: Write Push Invalid. Memory Write Request, with the FPGA cache hint set to Invalid. The FIU writes the data into the processors Last Level Cache (LLC) with no intention of keeping the data in the FPGA cache. The LLC it writes to is always the LLC associated with the processor where the DRAM address is homed.

Table 5.3: Read bandwidth for the three channels

	Read Bandwidth GB/s			
Channel	**VA**	**VL0**	**VH0**	**VH1**
rdline-I 1CL	17.304	5.963	5.122	5.062
rdline-I 2CL	17.304	5.563	6.015	6.128
rdline-I 4CL	19.806	5.543	6.367	6.362
rdline-S 1CL	16.624	7.474	5.113	5.138
rdline-S 2CL	19.343	7.487	6.159	6.152
rdline-S 4CL	19.927	7.482	6.343	6.352

As seen on Table 5.3 the sum of the bandwidth of the channels VL0, VH0 and VH1 is the maximum bandwidth of the platform can achieve using the VA channel. Cache directive rdline-S leads to significant increase of the bandwidth compare to rdline-I. This cache directives affects only VL0 channel and not VH0 and VH1. With the use of rdline-S the VL0 has more bandwidth than the VHs channels. In addition the more cachelines (CL) the system fetches from memory the more the bandwidth increases.

Table 5.4: Write bandwidth for the three channels

	Write Bandwidth GB/s			
Channel	**VA**	**VL0**	**VH0**	**VH1**
wrline-M 1CL	15.643	5.064	5.227	5.226
wrline-M 2CL	17.755	5.075	6.276	6.276
wrline-M 4CL	19.108	5.071	6.971	6.971
wrline-I 1CL	14.416	4.016	5.227	5.226
wrline-I 2CL	16.511	4.016	6.275	6.276
wrline-I 4CL	17.88	3.946	6.971	6.97
wrpush-I 1CL	14.396	4.017	5.226	5.226
wrpush-I 2CL	16.507	4.02	6.275	6.275
wrpush-I 4CL	17.838	4.018	6.971	6.971

By looking Table 5.4 again the conclusions are almost the same as before. Firstly the cache directives affects only the VLO channel and not the VH channels. The cache directive with the most achieved bandwidth is wrline-M. With the other two directives the results are almost identical. Moreover the more cachelines are being written in the memory the more the bandwidth is increased. Lastly a more general notice is that the read bandwidth is bigger than the write bandwidth of the platform

Figure 5.2: Diagram of the read bandwidth for different cache directives

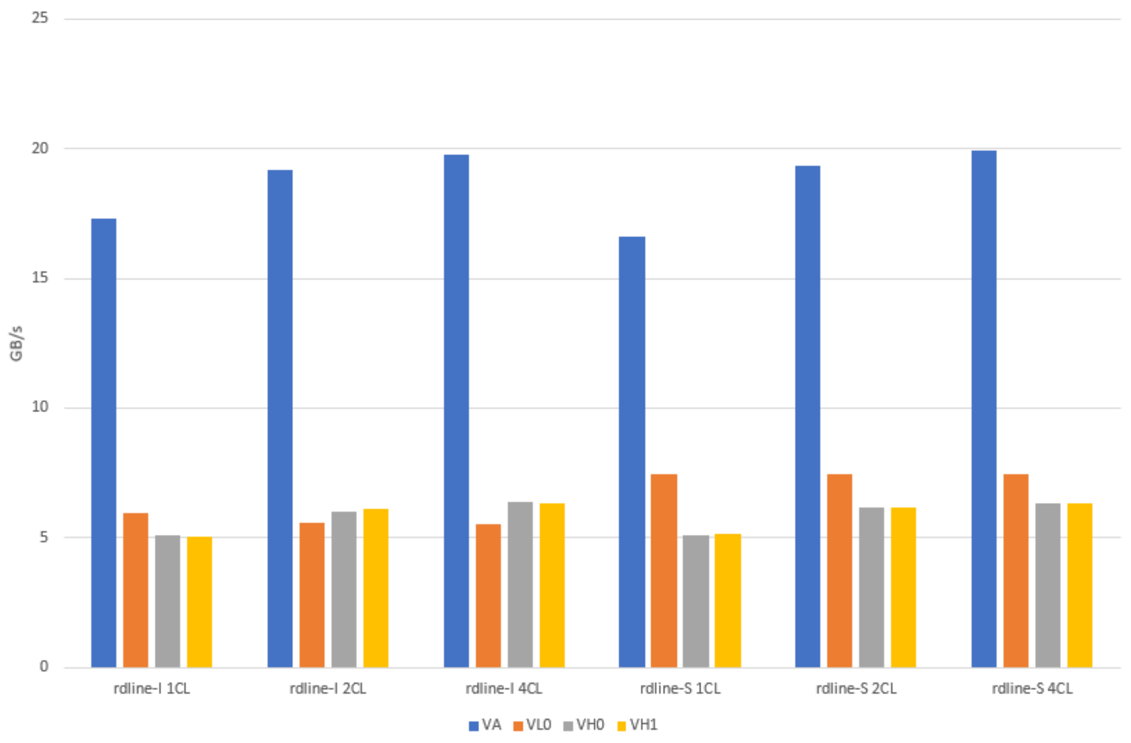
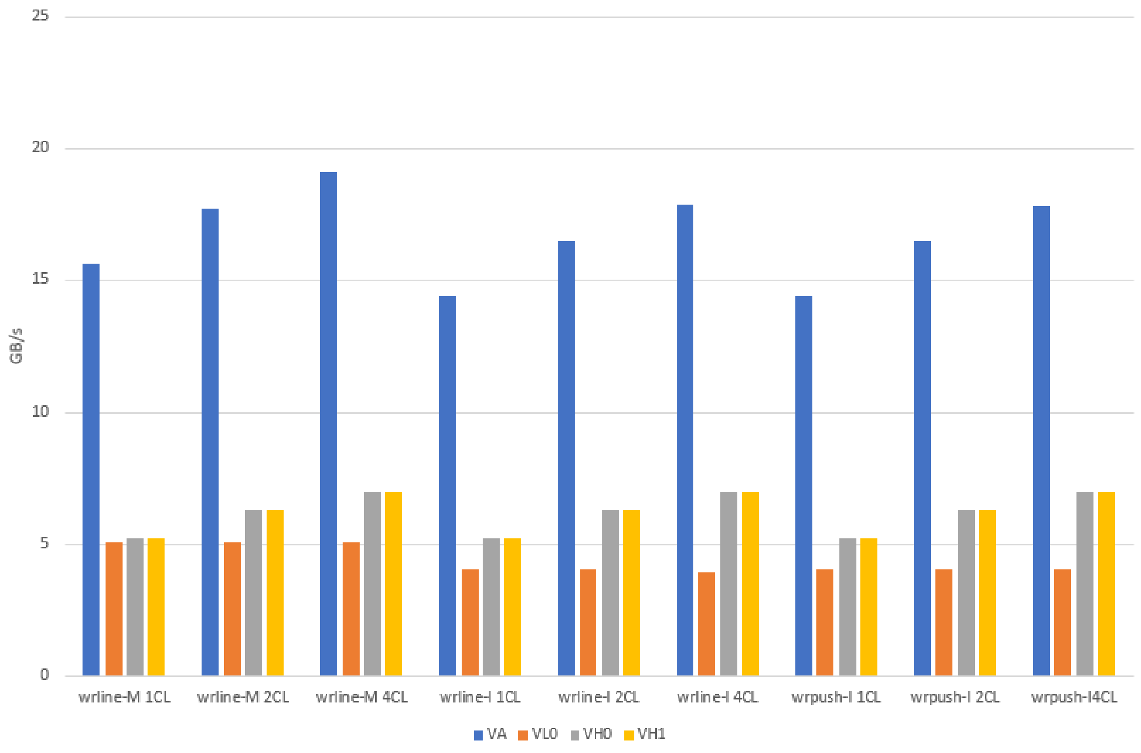


Figure 5.3: Diagram of the write bandwidth for different cache directives



5.2 Stream benchmark results

In order to measure the application bandwidth of the design the STREAM [23] (Sustainable Memory Bandwidth in High Performance Computers) benchmarks were used. Because the STREAM source code could not be used as is, it was examined and then were produced four simpler C codes that each contains one benchmark. The benchmarks are: C = A, B = scalar*C, B = A + C and B = A + scalar*C, where A,B and C are tables. These four C codes, with the use of ARM GNU compiler, were converted to ARM assembly. Afterwards these assembly codes were optimized by hand in order to minimize the loads and stores to the stack of the processor and achieve the maximum possible application bandwidth. For the floating point benchmarks the ARM instructions that are processing integers they were changed to floating point instructions. At the end of each program there were placed two standard instructions **str r0, [r3, #1024]** and **svc #0**. The **str** instructions tells the processor to flash its cache back to the main memory and **svc** informs the core that the program has finished and stops operating.

5.2.1 Integer results

The read requests on benchmarks C=A and B=Scalar*C are two times more than the write requests because to make a write on a certain cacheline first the old data must be read in order to save them back unaltered. On the other two benchmarks the read requests are three times more than the write requests because the benchmarks involve three tables.

Table 5.5: Benchmark C = A with MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	67497273	67496447	67347419	67556708	67776859
Execution Time(s)	0.98	0.98	0.98	0.98	0.98
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bdwth MB/s	20.44	40.89	81.96	163.42	325.77
Write bdwth MB/s	10.22	20.45	40.98	81.71	162.89

Table 5.6: Benchmark C = A with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	80983627	81000182	81006195	81066037	81021838
Execution Time(s)	1.17	1.17	1.17	1.17	1.17
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bandwidth MB/s	17.04	34.07	68.14	136.18	272.51
Write bandwidth MB/s	8.52	17.04	34.07	68.09	136.26

On Tables 5.5 and 5.6 are seen the results of the Benchmark C=A . The benchmark copies a memory region to another.

Table 5.7: Benchmark B = Scalar*C with MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	69833878	69923342	69840406	70035879	70581877
Execution Time(s)	1.01	1.01	1.01	1.02	1.02
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bdwth MB/s	19.76	39.47	79.04	157.63	312.82
Write bdwth MB/s	9.88	19.74	39.52	78.82	156.41

Table 5.8: Benchmark B = Scalar*C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	83332400	83641376	83648912	83595523	83675686
Execution Time(s)	1.21	1.21	1.21	1.21	1.21
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bandwidth MB/s	16.56	33.00	65.99	132.06	263.87
Write bandwidth MB/s	8.28	16.50	33.00	66.03	131.94

The benchmark B = Scalar*C (Tables 5.7, 5.8) is moving a part of the memory to another region and simultaneously alters it by a multiplication with a constant number.

Table 5.9: Benchmark B=A+C with MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	60582345	60623515	60711476	60851192	61504998
Execution Time(s)	0.88	0.88	0.88	0.88	0.89
# of elements in Table/core	174752	174752	174752	174752	174752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bdwth MB/s	22.78	45.52	90.91	181.41	358.97
Write bdwth MB/s	7.59	15.17	30.31	60.47	119.66

Table 5.10: Benchmark B=A+C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	73632703	73658017	73656416	73671021	73696264
Execution Time(s)	1.07	1.07	1.07	1.07	1.07
# of elements in Table/core	174752	174752	174752	174752	174752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bandwidth MB/s	18.74	37.47	74.94	149.84	299.58
Write bandwidth MB/s	6.25	12.49	24.98	49.95	99.86

Benchmarks on Tables 5.9 and 5.10 adds to tables and moves the result to a third one.

Table 5.11: Benchmark A=B+scalar*C with MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	62392900	62412403	62494948	62365775	63175507
Execution Time(s)	0.90	0.90	0.91	0.90	0.92
# of elements in Table/core	174.752	174.752	174.752	174.752	174.752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bdwth MB/s	22.12	44.22	88.32	177.01	349.47
Write bdwth MB/s	7.37	14.74	29.44	59.00	116.49

Table 5.12: Benchmark A =B +scalar*C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	75579391	75386692	75367797	75395413	75452664
Execution Time(s)	1.10	1.09	1.09	1.09	1.09
# of elements in Table/core	174752	174752	174752	174752	174752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bandwidth MB/s	18.26	36.61	73.24	146.42	292.61
Write bandwidth MB/s	6.09	12.20	24.41	48.81	97.54

The last STREAM benchmark multiplies elements of C table with a constant number and adds the elements of B and stores the result to table A.

The observation that occurs from the above Tables is that the bandwidth is analogous to the number of the cores and the difference between the read and write bandwidth depends on the number of tables are being read and written. In addition designs implemented with MLAB RAMs are executed faster than M20K designs because MLAB RAMs allow asynchronous read requests while M20K has one cycle delay. Furthermore MLAB benchmarks has more application bandwidth because for the same amount of requests the execution time is shorter.

In the figures below there are presented the read and write bandwidth for each benchmark for the different RAM implementations for better visualization.

Figure 5.4: Diagram of the read bandwidth for the integer benchmarks with MLAB RAM

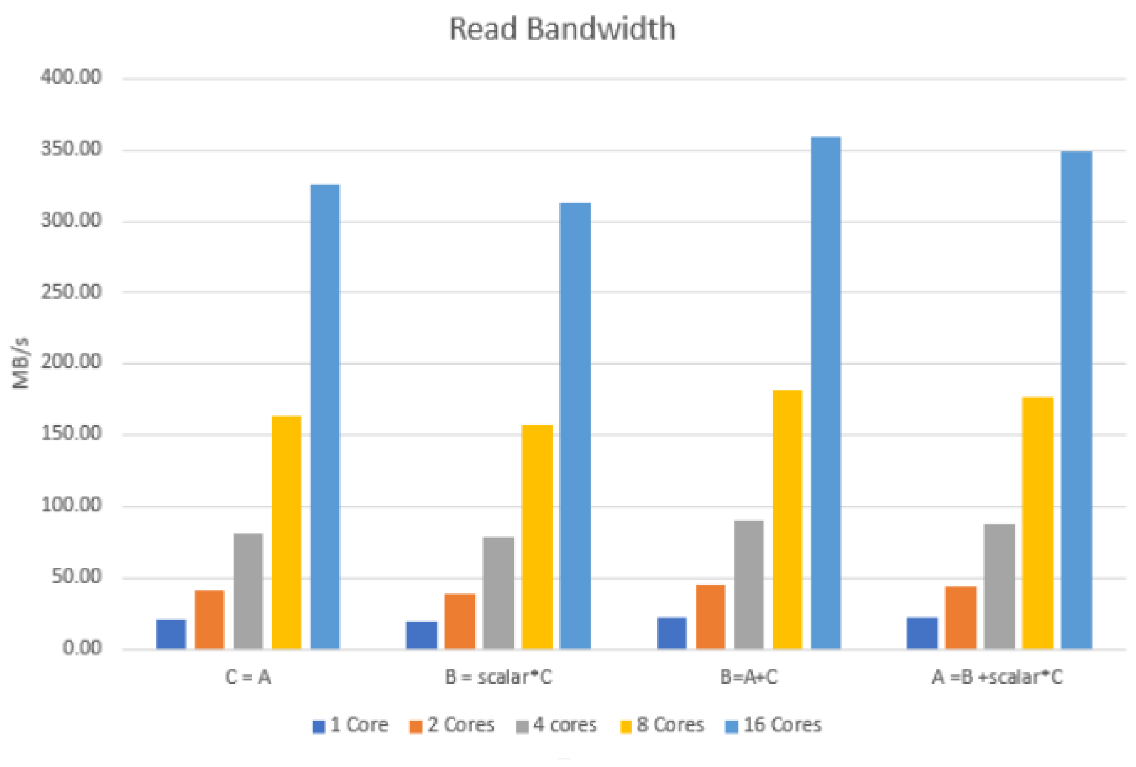


Figure 5.5: Diagram of the write bandwidth for the integer benchmarks with MLAB RAM

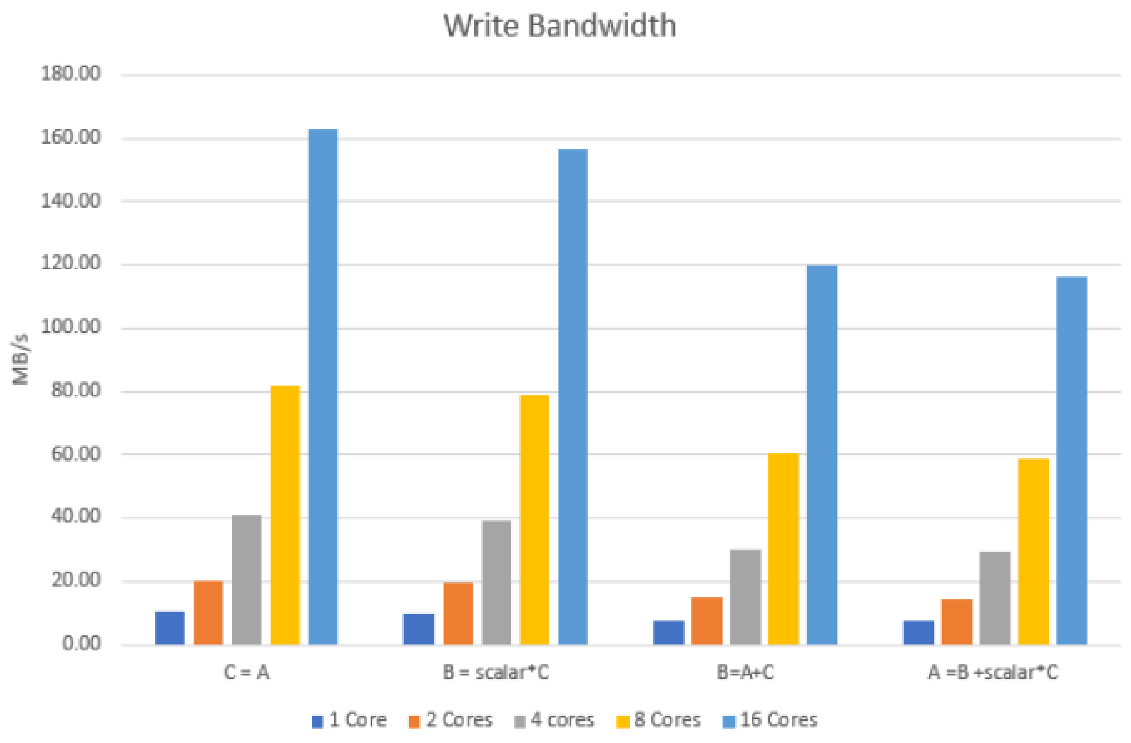


Figure 5.6: Diagram of the read bandwidth for the integer benchmarks with M20K RAM

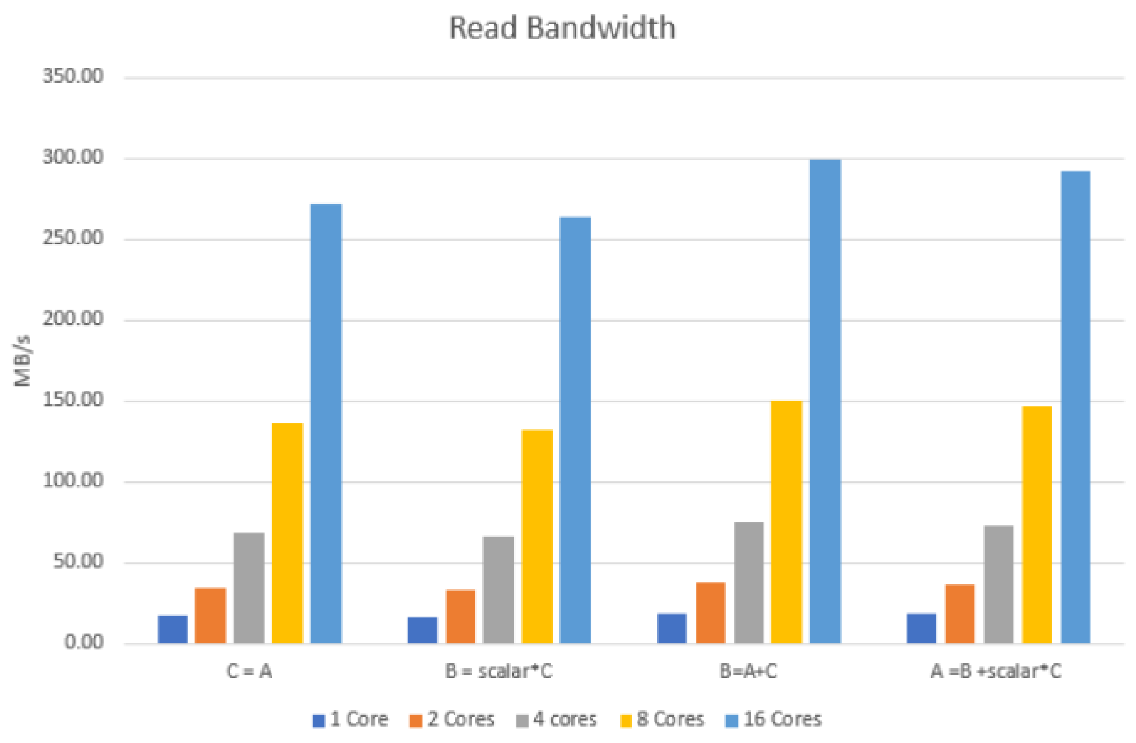
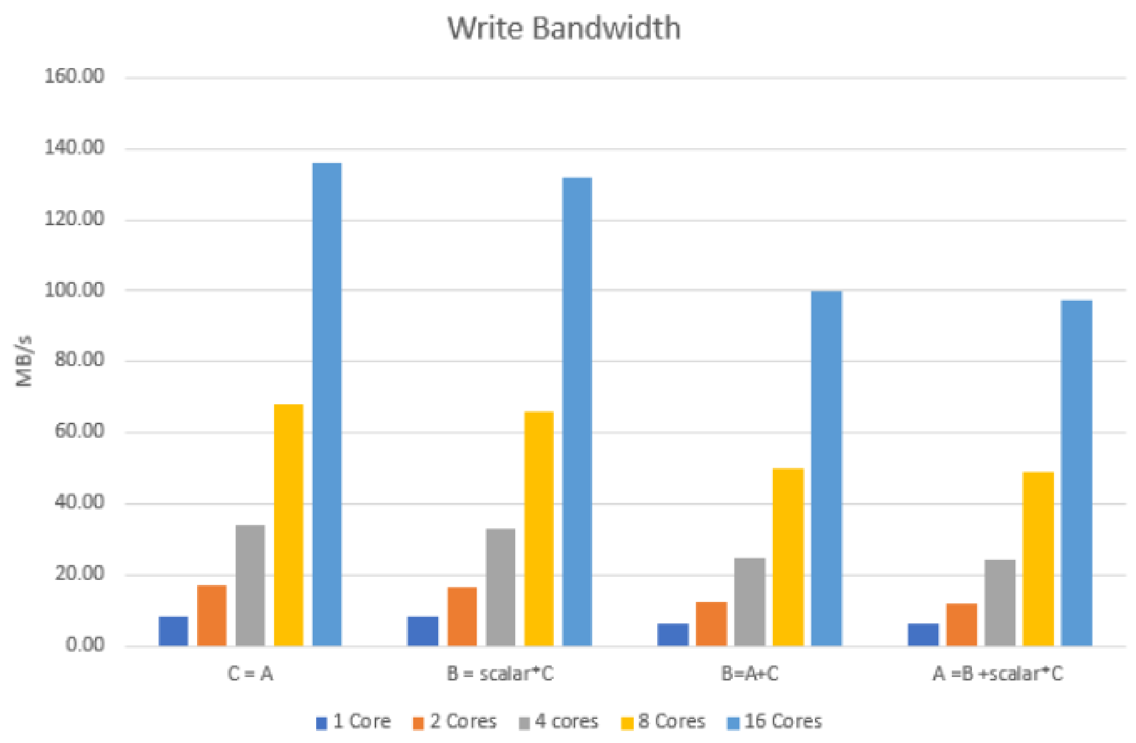


Figure 5.7: Diagram of the write bandwidth for the integer benchmarks with M20K RAM



5.2.2 Floating point results

In the following tables are presented the STREAM benchmarks that use floating point numbers. The above observations apply to the following floating point results. One more note is that the floating point benchmarks requires more time to execute because floating point instructions need more cycles to produce results. s a consequence the read and write bandwidth decreases because for the same read and write request the execution time has been increased.

Table 5.13: Floating point benchmark C = A with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	80972393	80998299	81006094	81015377	81028297
Execution Time(s)	1.17	1.17	1.17	1.17	1.17
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bandwidth MB/s	17.04	34.07	68.14	136.27	272.49
Write bandwidth MB/s	8.52	17.04	34.07	68.14	136.25

Table 5.14: Floating point benchmark B = Scalar*C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	88638697	88704084	88727067	88713138	88778209
Execution Time(s)	1.28	1.29	1.29	1.29	1.29
# of elements in Table/core	262144	262144	262144	262144	262144
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	163840	327680	655360	1310720	2621440
Read bandwidth MB/s	15.57	31.11	62.21	124.44	248.71
Write bandwidth MB/s	7.78	15.56	31.11	62.22	124.35

Table 5.15: Floating point benchmark B=A+C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	78791044	78789298	78799787	78820816	78821295
Execution Time(s)	1.14	1.14	1.14	1.14	1.14
# of elements in Table/core	174752	174752	174752	174752	174752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bandwidth MB/s	17.51	35.03	70.05	140.05	280.11
Write bandwidth MB/s	5.84	11.68	23.35	46.69	93.37

Table 5.16: Floating point benchmark A =B +scalar*C with M20K RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	84065472	84069619	84045993	84079352	84259279
Execution Time(s)	1.22	1.22	1.22	1.22	1.22
# of elements in Table/core	174752	174752	174752	174752	174752
# Read Requests	327674	655348	1310696	2621392	5242784
# Write Requests	109220	218440	436880	873760	1747520
Read bandwidth MB/s	16.41	32.83	65.67	131.29	262.03
Write bandwidth MB/s	5.47	10.94	21.89	43.77	87.34

Figure 5.8: Diagram of the write bandwidth for the floating point benchmarks

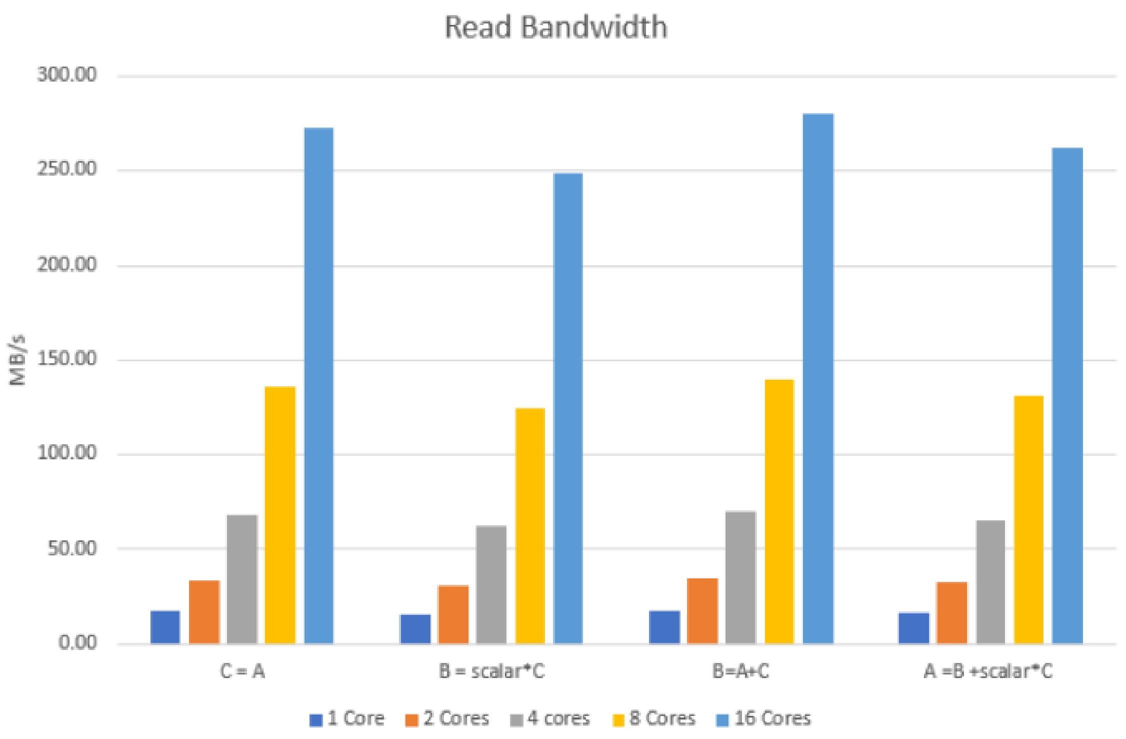
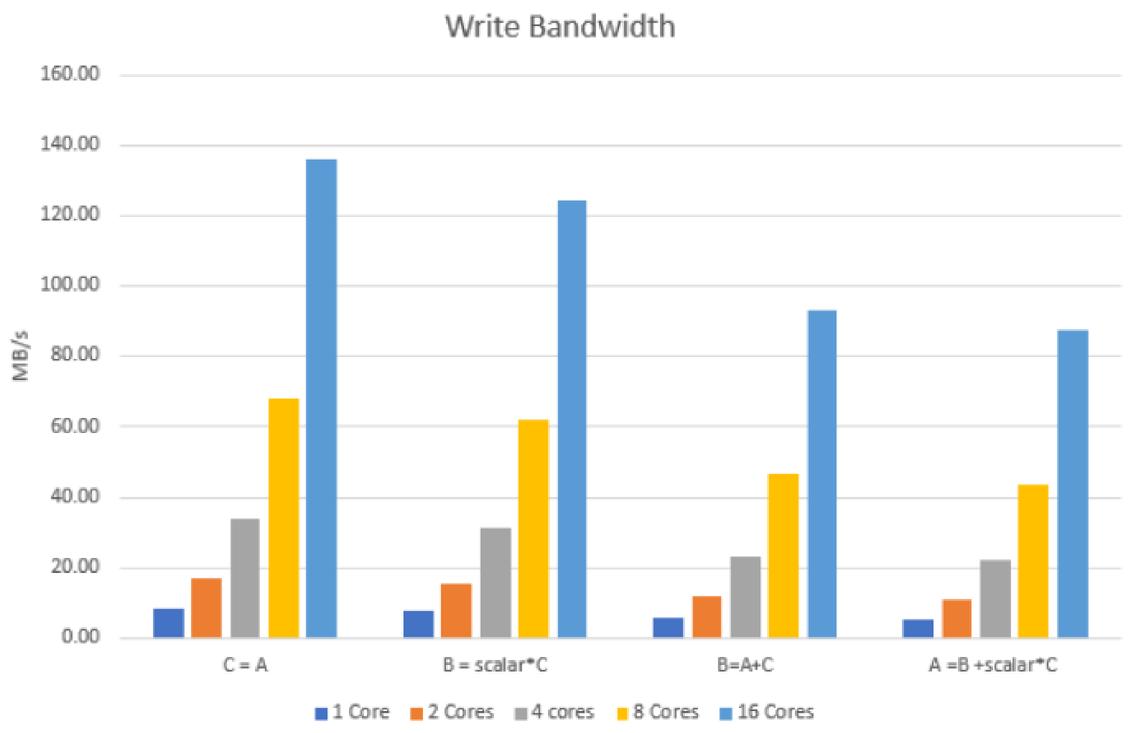


Figure 5.9: Diagram of the write bandwidth for the floating point benchmarks



5.3 Matrix Multiplication

The same process was followed to produce the assembly code for the matrix multiplication program. At first a code for matrix multiplication was written in C. Afterwards this C code was converted to assembly with ARM GNU compiler. The produced assembly code was optimized by hand for the purpose of achieving the highest possible application bandwidth. For the floating point matrix multiplication the instructions that are processing integers were altered to floating point instructions.

5.3.1 Integer results

Table 5.17: Benchmark matrix multiplication C = A x B - cache size = 1KB MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3829425172	4038496166	4037733909	4065472896	3903235027
Execution Time(s)	55.50	58.53	58.52	58.92	56.57
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	26599877	53199754	106399508	212799016	425598032
# Write Requests	9089600	18179200	36358400	72716800	145433600
Read bdwth MB/s	29.25	55.48	110.98	220.44	459.20
Write bdwth MB/s	10.00	18.96	37.92	75.33	156.92

Table 5.18: Benchmark matrix multiplication C = A x B - cache size = 16KB MLAB RAM

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	2784450582	2800326614	2784823752	2787313662	2800822924
Execution Time(s)	40.95	41.18	40.95	40.99	41.19
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	5521750	11043500	22087000	44174000	88348000
# Write Requests	292016	584032	1168064	2336128	4672256
Read bandwidth MB/s	8.23	16.37	32.92	65.78	130.92
Write bandwidth MB/s	0.44	0.87	1.74	3.48	6.92

Table 5.19: Benchmark matrix multiplication C = A x B - cache size = 1KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3922716875	4659297857	4369969573	4168544614	4296298707
Execution Time(s)	56.85	67.53	63.33	60.41	62.27
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	26599877	53199754	106399508	212799016	425598032
# Write Requests	9089600	18179200	36358400	72716800	145433600
Read bandwidth MB/s	28.56	48.09	102.54	214.99	417.19
Write bandwidth MB/s	9.76	16.43	35.04	73.46	142.56

Table 5.20: Benchmark Matrix MULL C = A x B - cache size = 16KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3361090801	3362092055	3361562960	3362696263	3377215132
Execution Time(s)	48.71	48.73	48.72	48.73	48.95
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	5521750	11043500	22087000	44174000	88348000
# Write Requests	292016	584032	1168064	2336128	4672256
Read bdwth MB/s	6.92	13.83	27.67	55.32	110.17
Write bdwth MB/s	0.37	0.73	1.46	2.93	5.83

Table 5.21: Benchmark matrix multiplication C = A x B - cache size = 32KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3350368070	3358740516	3358362688	3358875024	3349574061
Execution Time(s)	48.56	48.68	48.67	48.68	48.54
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	5021463	10042926	20085852	40171704	80343408
# Write Requests	151424	302848	605696	1211392	2422784
Read bandwidth MB/s	6.31	12.59	25.19	50.37	101.02
Write bandwidth MB/s	0.19	0.38	0.76	1.52	3.05

Table 5.22: Benchmark matrix multiplication C = A x B - cache size = 64KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3344425460	3346195969	3346301653	3346313763	3340995166
Execution Time(s)	48.47	48.50	48.50	48.50	48.42
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	4771032	9542064	19084128	38168256	76336512
# Write Requests	81040	162080	324160	648320	1296640
Read bandwidth MB/s	6.01	12.01	24.02	48.04	96.22
Write bandwidth MB/s	0.10	0.20	0.41	0.82	1.63

Table 5.23: Benchmark matrix multiplication C = A x B - cache size = 128KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3338726602	3339409822	3339598779	3339691036	3334267761
Execution Time(s)	48.39	48.40	48.40	48.40	48.32
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	4645819	9291638	18583276	37166552	74333104
# Write Requests	45856	91712	183424	366848	733696
Read bandwidth MB/s	5.86	11.72	23.43	46.87	93.89
Write bandwidth MB/s	0.06	0.12	0.23	0.46	0.93

Figure 5.10: Diagram of the read bandwidth for integer matrix multiplication for different cache size/type

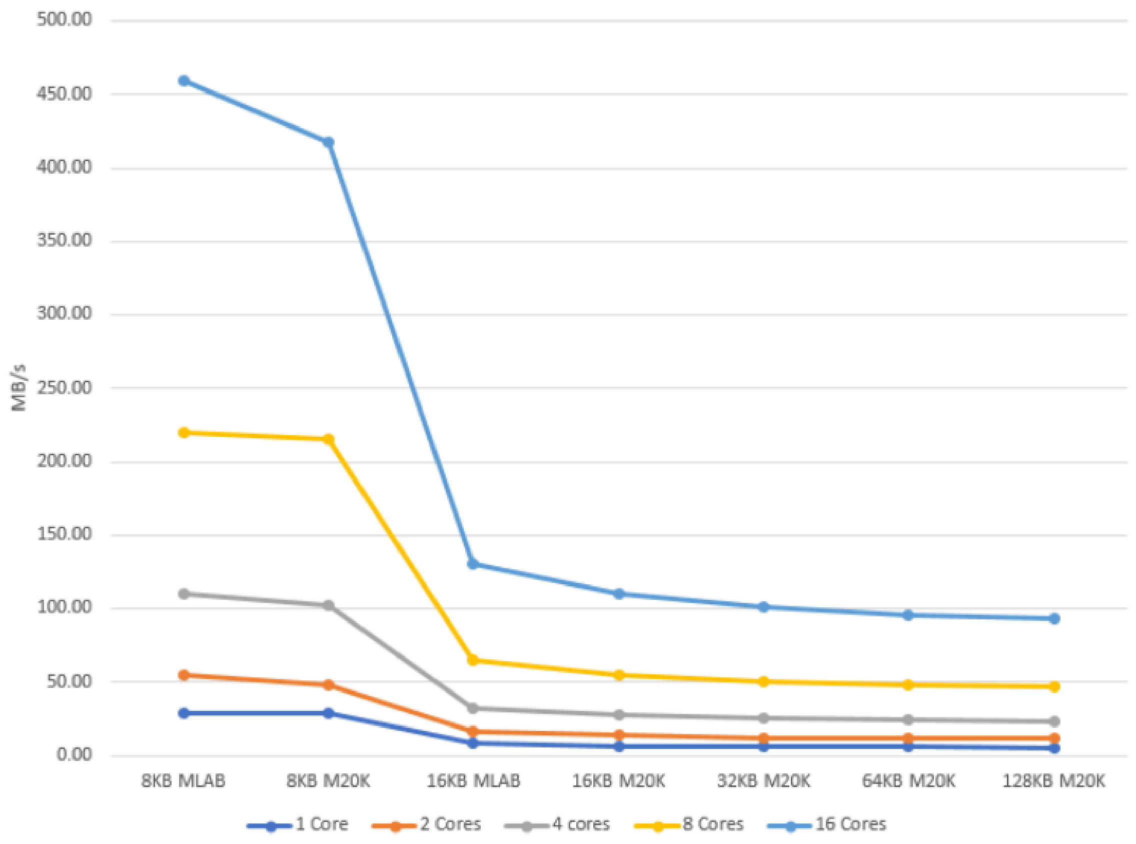
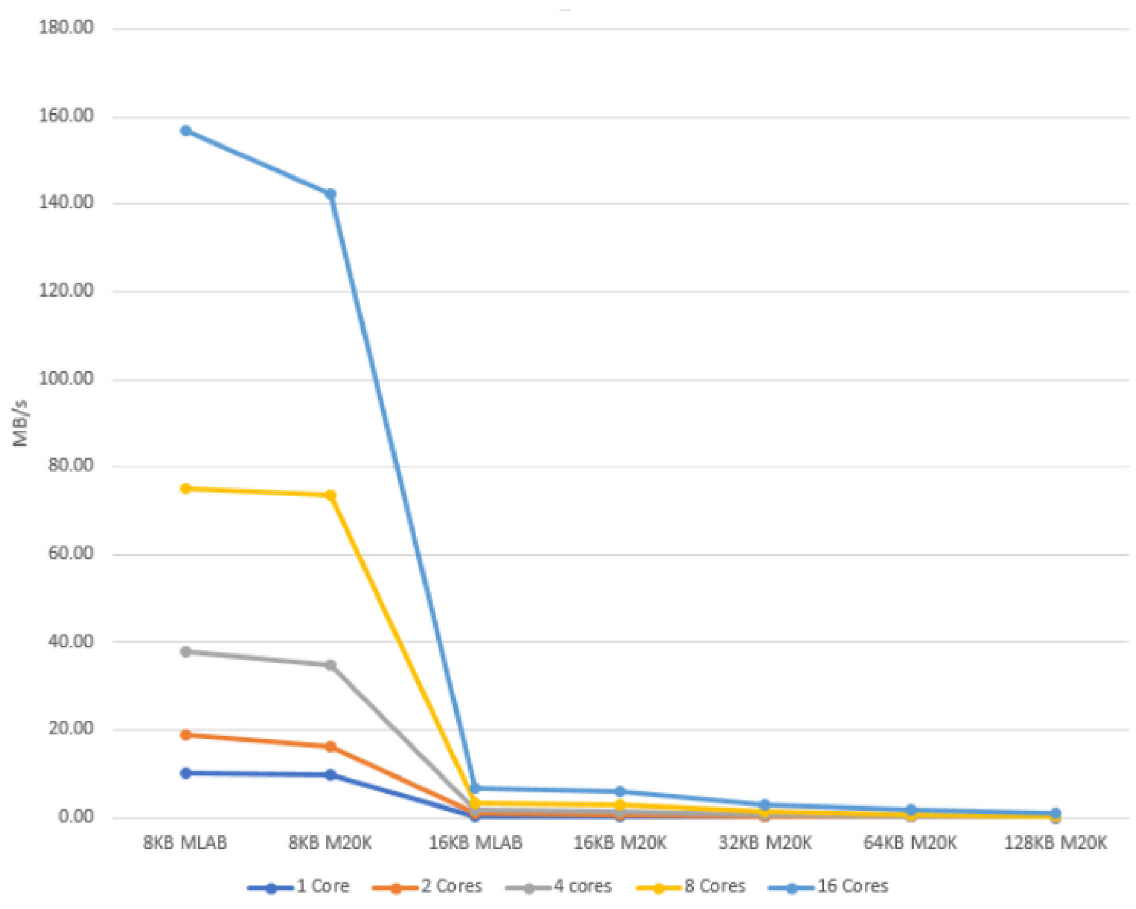
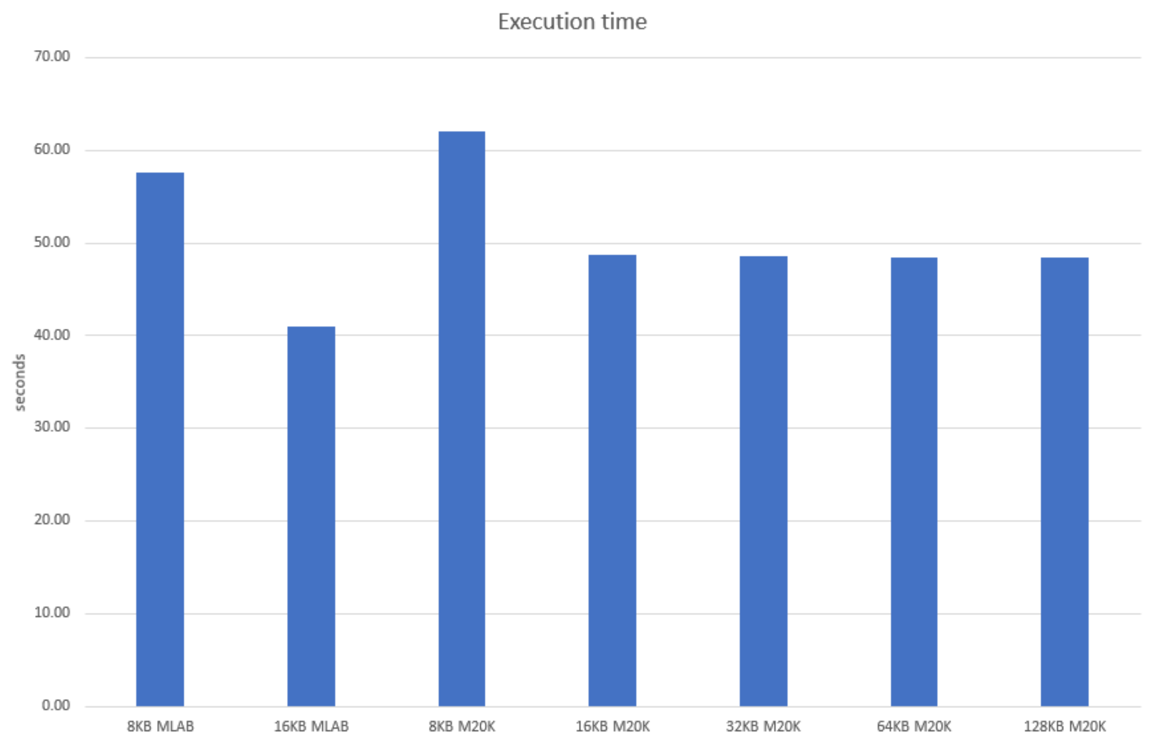


Figure 5.11: Diagram of the write bandwidth for integer matrix multiplication for different cache size/type



Likewise to the STREAM benchmarks the MLAB designs are faster than the M20K designs for the same cache size. The implementations with 1KB cache has more bandwidth because they need to perform more requests to transfer all the data into the FPGA. Accordingly, as the cache size gets larger,the requests towards memory, the bandwidth and runtime decrease.

Figure 5.12: Diagram of the average execution for integer matrix multiplication time compared with the cache size/type



Looking at Figure 5.12 The increase of cache size from 1KB to 16 KB has significant impact to performance, as the execution time is decreased by 10 seconds. From there and after the increase of cache size only decreases the runtime by 0.1 seconds. So the best choice for cache size is 16 KB.

5.3.2 Floating Point results

In the below Tables and Figures are presented the results of matrix multiplication with floating point numbers for different cache sizes.

Table 5.24: Floating point benchmark matrix multiplication C = A x B - cache size = 16KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3793640911	3814184195	3814643568	3814267970	3801890906
Execution Time(s)	54.98	55.28	55.28	55.28	55.10
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	5521750	11043500	22087000	44174000	88348000
# Write Requests	292016	584032	1168064	2336128	4672256
Read bandwidth MB/s	6.13	12.19	24.38	48.77	97.86
Write bandwidth MB/s	0.32	0.64	1.29	2.58	5.18

Table 5.25: Floating point benchmark matrix multiplication C = A x B - cache size = 32KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3781604333	3789728696	3791575483	3789617846	3789892861
Execution Time(s)	54.81	54.92	54.95	54.92	54.93
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	5021463	10042926	20085852	40171704	80343408
# Write Requests	151424	302848	605696	1211392	2422784
Read bandwidth MB/s	5.59	11.16	22.31	44.64	89.28
Write bandwidth MB/s	0.17	0.34	0.67	1.35	5.19

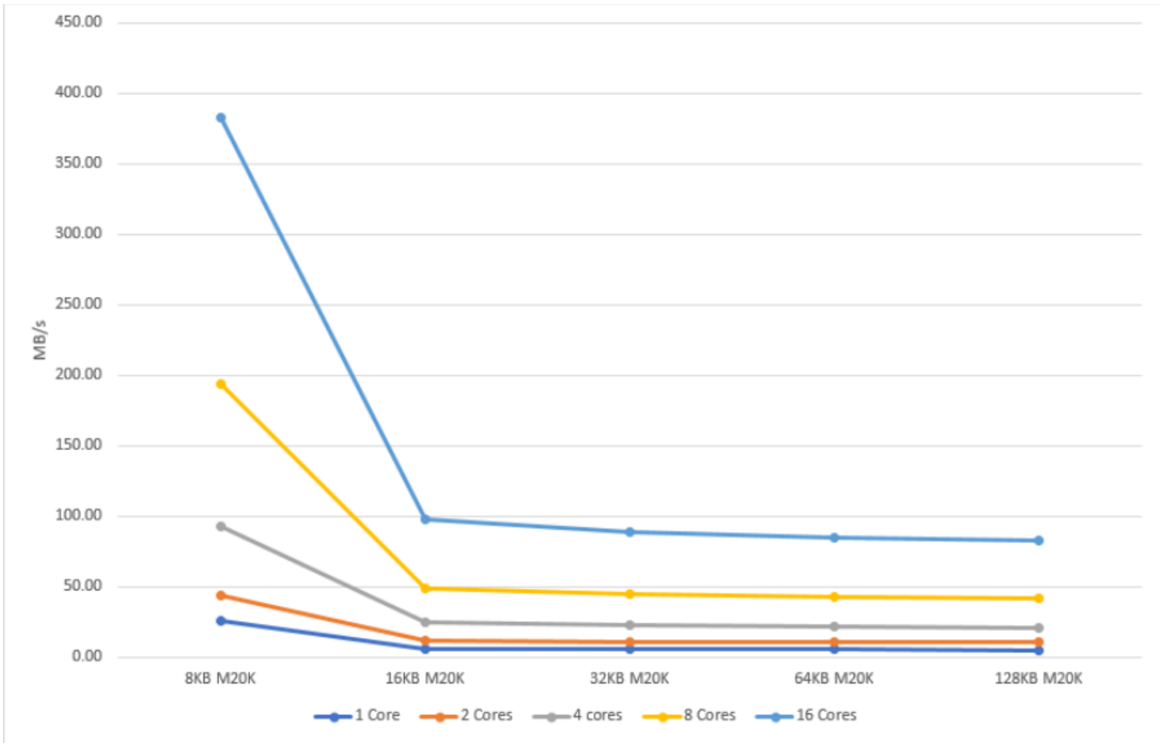
Table 5.26: Floating point benchmark matrix multiplication C = A x B - cache size = 64KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3775620861	3777130793	3777191841	3777187956	3777323784
Execution Time(s)	54.72	54.74	54.74	54.74	54.74
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	4771032	9542064	19084128	38168256	76336512
# Write Requests	81040	162080	324160	648320	1296640
Read bandwidth MB/s	5.32	10.64	21.28	42.56	85.11
Write bandwidth MB/s	0.09	0.18	0.36	0.72	1.45

Table 5.27: Floating point benchmark matrix multiplication C = A x B - cache size = 128KB

	1 core	**2 cores**	**4 cores**	**8 cores**	**16 cores**
Cycles	3769917836	3773694524	3770575355	3770788846	3770725381
Execution Time(s)	54.64	54.69	54.65	54.65	54.65
# of elements in Table/core	173056	173056	173056	173056	173056
# Read Requests	4645819	9291638	18583276	37166552	74333104
# Write Requests	45856	91712	183424	366848	733696
Read bandwidth MB/s	5.19	10.37	20.76	41.51	83.02
Write bandwidth MB/s	0.05	0.10	0.20	0.41	0.82

Figure 5.13: Diagram of the read bandwidth for floating point matrix multiplication for different cache sizes



For the case of matrix multiplication of the floating point numbers an increase of the execution time, compared to the integer, is observed again as in the STREAM benchmarks. The reason again is that the floating point instructions need more cycles to execute on the ARM processor. Once more the bandwidth and the execution time is decreased while the cache size increases.

Figure 5.14: Diagram of the write bandwidth for floating point matrix multiplication for different cache sizes

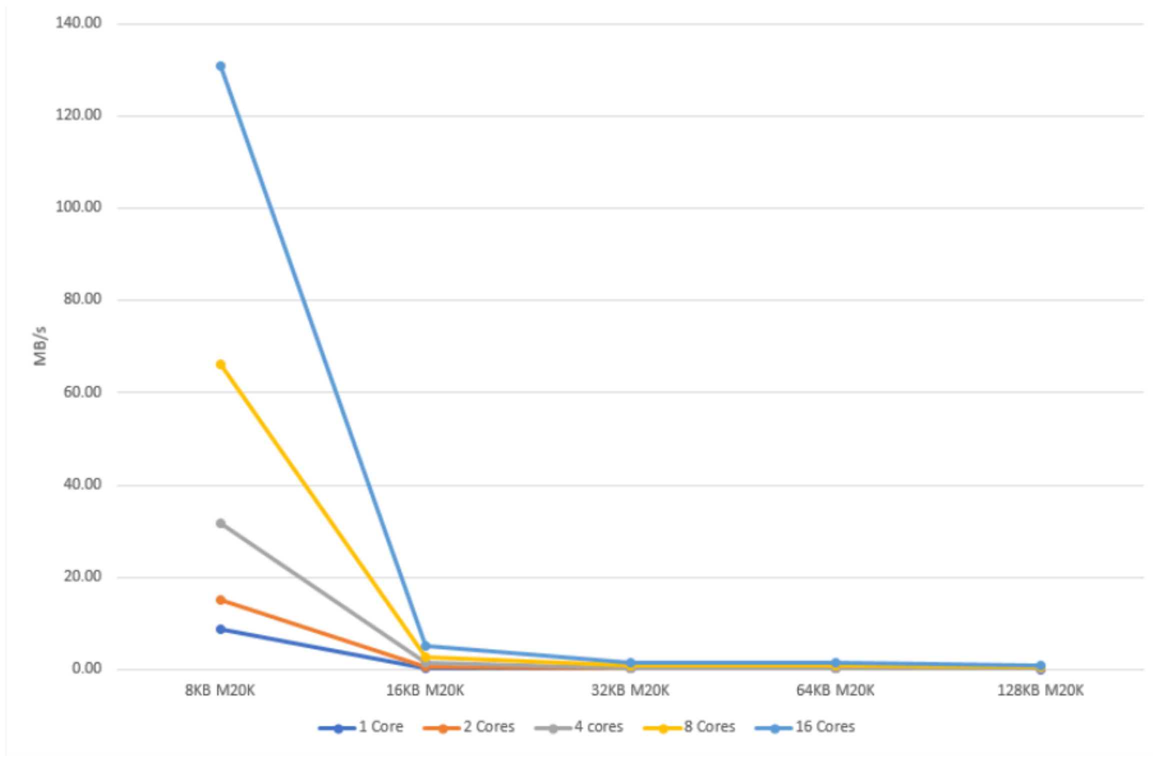
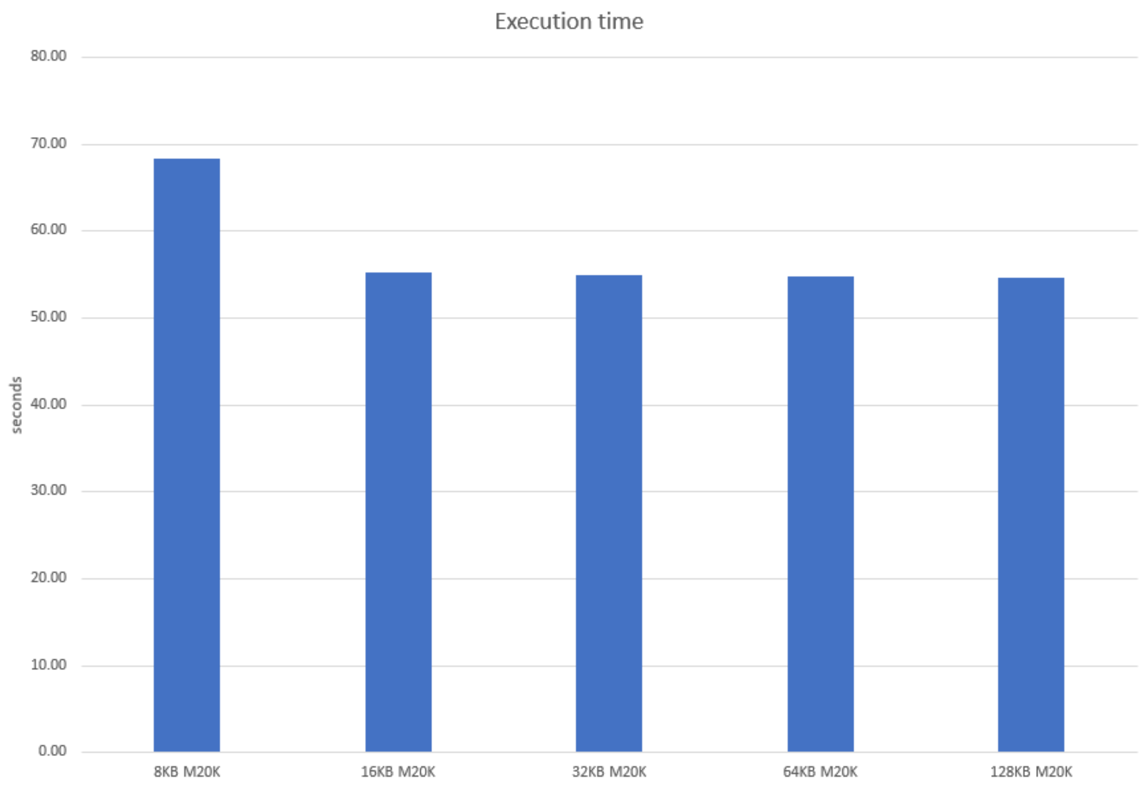


Figure 5.15: Diagram of the average execution time for floating point matrix multiplication time and cache size



Observing the Figure 5.15 the average execution time of the implementation with 1KB cache size is more than 10 seconds shorter than the 16KB design. After that point the decrease of the execution time is almost zero. In conclude the best choice for the cache size is 16 KB as in the integer matrix multiplication example.

Chapter 6

Conclusions and Future work

6.1 Conclusions of Thesis

This master thesis was an attempt to evaluate the Intel scalable Xeon with integrated FPGA and investigate what kind of applications can be accelerated in this platform. Depending on the measurements of the read and write bandwidth of the system and the results of the benchmarks with the ARM accelerator the conclusion that comes is that streaming applications would be preferable for this platform. The accelerator that have been implemented for this thesis with the 16 ARM cores utilizes only the 2% of the platform's theoretical bandwidth, that means that there have to be some improvements in order to achieve higher bandwidth. In general this platform is very user friendly because Intel provides all the necessary tools and guides in order to the developer can start right away to design accelerators.

6.2 Future Work

- Optimize the code of the 3-stage pipeline processor and see if this has as a result to get higher clock frequency and fewer ALMs on the FPGA.
- Design a processor with 5-stage pipeline and data forwarding.
- Expand the instruction set to another version(ARMv4T, ARMv5T etc).
- Include instruction and data caches.

- Implement a branch prediction unit
- Study the tool to find an optimal way to design processors in Bluespec
- Implement some peripherals for the processor(e.g Debug and Support Unit)
- Implement a multicore design with L2 L3 cache
- Resource management of the design in order to fit 32 cores
- Improve architecture to achieve higher bandwidths
- Power management of the design

Bibliography

[1] *Accelerator Functional Unit (AFU) Developer's Guide for Intel FPGA Programmable Acceleration Card (Intel FPGA PAC)*. URL: https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html.

[2] *Amazon EC2 F1 Instance*. 2017. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[3] *ARM ArchitectureReference Manual*. URL: https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf.

[4] *ARM7TDMI-S Data Sheet*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf.

[5] *Bluespec System Verilog Wiki*. URL: http://wiki.bluespec.com/.

[6] *Bluespec TM SystemVerilog Reference Guide*. URL: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf.

[7] Young-Kyu Choi et al. "In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.1 (2019), p. 4.

[8] Nirav Hemant Dave et al. "Designing a processor in Bluespec". PhD thesis. Massachusetts Institute of Technology, 2005.

[9] Joel S Emer and Murali Vijayaraghavan. "Computer Architecture: A Constructive Approach". In: ().

[10] *Infocenter ARM*. URL: http://infocenter.arm.com.

[11] *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) User Guide.* URL: https://opae.github.io/1.1.2/docs/ase_userguide/ase_userguide.html.

[12] *Intel to Start Shipping Xeons With FPGAs in Early 2016.* 2016. URL: http://www.eweek.com/servers/%20intel-to-start-shipping-xeons-with-fpgas-in-early-2016.html.

[13] *Intel Wiki for FPGA accelerators.* URL: https://wiki.intel-research.net/FPGA.html.

[14] Kyriakidis Konstantinos. "Full system architectural simulation on the HARP integrated CPU-FPGA platform". School of Electrical and Computer Engineering, Technical University of Crete, 2019. URL: https://dias.library.tuc.gr/view/82851.

[15] Daniel Mattsson and Marcus Christensson. *Evaluation of synthesizable CPU cores.* Chalmers tekniska högskola, 2004.

[16] Rishiyur S Nikhil and Kathy R Czeck. "BSV by Example". In: *CreateSpace, Dec* (2010).

[17] Rishiyur S Nikhil, Daniel L Rosenband, Nirav Dave, et al. "High-level synthesis: an essential ingredient for designing complex ASICs". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* IEEE. 2004, pp. 775–782.

[18] *Nios II Custom Instruction UserGuide.* URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf.

[19] *OPAE C API Programming Guide.* URL: https://opae.github.io/1.1.2/docs/fpga_api/prog_guide/readme.html.

[20] K. Ovtcharov et al. "Toward accelerating deep learning at scale using specialized hardware in the datacenter". In: *2015 IEEE Hot Chips 27 Symposium (HCS).* Aug. 2015, pp. 1–38. DOI: 10.1109/HOTCHIPS.2015.7477459.

[21] Georgios Pekridis. "Implementation of ARM processor by using Bluespec language". School of Electrical and Computer Engineering, Technical University of Crete, 2018. URL: https://dias.library.tuc.gr/view/71171.

[22] Andrew Putnam et al. “A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services”. In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 13–24. ISSN: 0163-5964. DOI: 10.1145/2678373.2665678. URL: http://doi.acm.org/10.1145/2678373.2665678.

[23] *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* URL: https://www.cs.virginia.edu/stream/.