SAPIENZA
UNIVERSITÀ DI ROMA

Energy Efficient Digital Electronic Systems Design for Edge-Computing Applications, through Innovative RISC-V Compliant Processors.

By

Abdallah Cheikh
عبدالله نبيل الشيخ

A Thesis Submitted to the Department of Information Electronics and Telecommunication Engineering (DIET)
La Sapienza Università di Roma

DOCTOR OF PHILOSOPHY
February 2020

Prof Mauro Olivieri
(Thesis Supervisor)

Acknowledgements

These three years of doing research at the LSD lab at Sapienza have passed like the wind. Throughout my journey on this PhD career, I slowly transformed from being a High-Power Electrical Engineer, into a Low-Power Computer Architect for IoT devices and Embedded Systems. The only thing I regret is the amount of strain I placed on my eyes, caused by staring daily at the computer screen for over 12 hours. However, I am much more thankful than regretful for the things I have experienced in this wonderful journey.

First and foremost, I am thankful to all mighty God الله for giving me everything I have asked for, and more. He facilitated my means to travel to Italy, He surrounded me with kind hearted, and supportive people that helped me in a foreign country in which I barely understood the language, and He was always my guide in the good and the bad moments in life. I never had to beg anyone for anything neither did I feel at any point in time the struggle to sustain myself. God is great, and no matter how many times I thank Him, I feel that it is not enough and that I should be ever more thankful.

Needless to mention, but nonetheless I am thankful for both my parents that had never batted an eye when I asked them for help. They covered any financial shortcomings I had throughout my PhD career. They provided me with every kind of support in order help me succeed in my career. My parents never once have abandoned me, and they always prayed for my success. As they grow into their older days, I wish to support them by giving back at least a fraction of what they'd been giving me my entire life.

Second of all, I would then like to thank my professor and thesis supervisor Mauro Olivieri, as I feel eternally grateful for his support throughout my PhD career. I contacted Mauro in early July 2016 for a chance to pursue a PhD career at Sapienza, and Mauro quickly responded to my request, helping me every step of the way in the application process until I finally got admitted to the PhD program. Mauro continued his support and guidance throughout the years providing me with opportunities to pursue conferences in different parts of Italy and Europe. In May 2018 he again provided me with the unforgettable and wonderful opportunity to move to Barcelona and collaborate on the European Processing Initiative (EPI) project. In Barcelona, I met amazing people and gained a lot of experience in the field of computer architecture. Mauro till this day continues to be a great support, as he constantly provides me with wonderful opportunities at every turn, and for that I am always very grateful, and have very much respect for all what he has done for me.

Furthermore, I am grateful to my amazing friend and mentor Antonio Mastrandrea. Antonio from day one in Italy was there for me. The reason I managed to stay standing on my feet in Italy, without getting lost or stranded was Antonio himself. He helped me literally in anything I asked for. Antonio was basically my guide for everything in Italy. Not to mention throughout my PhD he continuously provided a lot of support in various areas I lacked experience in. Antonio is a great friend, and a great support, and I really enjoyed his company throughout my PhD career. Thank you, Antonio!

I have met a great deal of people in the past three years, among them is my great friend Simone Ponzio. Simone volunteered to help me with my work continuously for more than 8 months. Simone helped, me perform the earlies parts of verification of my work in the second year and he also was a great friend and a very fun guy to be around. Then came along my colleague and my dear friend Stefano Sordillo. Without Stefano's amazing hard work and his collaboration on common areas of interests in our researcher, I would not have had my work results flourish as they had today, Stefano was the software developer that made the complex tests which benchmarked my work. Stefano was a great help at all times, even on the

weekends. For all the people that I have met, I want to say you were all amazing, and thank you all for giving me the pleasure of meeting you.

As a final note I would like to express my gratitude to the Italian government, and their vision to provide a career opportunity for a both foreign and national students equally by allowing them to pursue a PhD career all under their expenses, and without any bias in the selection process be it race, nationality, gender, or religion. Italy in that sense I consider to be a model country, and I owe my thanks to all the Italians for their kindness and hospitality towards me, and other foreign researchers as well.

Table of Contents

Contents

List of Figures

List of Tables

Abstract

The number of IoT devices has greatly increased over the years, so that they have invaded the electronic market. IoT describe a device-to-device communication without human interface. A large class of these devices are battery powered, and the energy consumption inside them is considered critical.

Today's embedded IoT systems interface multiple peripherals such as sensors that perform continuous monitoring of the environment around it, and actuators that are controlled by the embedded systems. Also, they interface wireless devices for data transmissions. A part of their job includes some basic pre-processing of the data before transmitting it over those wireless networks. Such pre-processing "on the edge of the network" minimizes the data to be transmitted over the wireless channels, and only transmits the desired outputs.

In front of the increase demand to support pre-processing, such as computer vision and voice recognition, on small embedded systems on the edge of the network, they cannot completely satisfy those demands due to their little performance

In this study we demonstrate the performance and energy efficiency of interleaved multithreaded architectures, which can be used in an embedded system on the edge of the IoT interfacing multiple sensors and peripherals, each serviced by a different hardware thread. We show the optimal pipeline organization to use in such architectures, and we finally demonstrate how these architectures can be exploited to easily improve instruction level parallelism by integrating a convolutional neural networking accelerator that can perform very fast vector arithmetic operations, and finally benchmarking this accelerator by running a custom implementation of the VGG16 convolutional neural network.

The microprocessors presented are a part of a family of processing cores called *Klessydra*. The Klessydra microprocessors were written such that they have a pinout that are 100 percent identical with Riscy cores from PULPino SoC. The subset of the Klessydra cores presented in this thesis is called the *Klessydra-T*. The letter 'T' indicating that the cores are multithreaded, the Klessydra-T subset has two main implementations used throughout this thesis, they are *Klessydra-T03* and *Klessydra-T13*. *T03* and *T13* for short.

The processor cores have been tested with the Modelsim / Questasim simulators. The cores have been synthesized on the 7-series FPGAs from Xilinx with the Vivado Synthesis tool. Synthesis and Post-synthesis simulations have been made. Dynamic Power estimations were calculated by Vivado from the power report generated by Modelsim after having simulated a post-synthesis Vivado netlist. FPGA synthesis was chosen as our target implementation, as they provide high reconfigurability, which allows the user to easily customize their own accelerator and make it adapt accordingly to their specific applications.

In our assessment throughout this thesis we nominated the T03 interleaved multithreaded processor as our optimal and most balanced pipeline organization. The T03 core had many advantages over other architectures, however it was only suitable to be used in control applications. T13 solves this problem by implementing superscalar hardware accelerators. A hybrid implementation of the hardware accelerator targeting thread level parallelism and slight data level parallelism was the approach yielding the highest performance and still maintaining a relatively low energy consumption for energy critical environments.

Organization of the Dissertation:

Chapter 1 This dissertation starts with the preface that provides a brief literature review of IoT devices, and the convergence between cloud computing and embedded systems.

Chapter 2 The second chapter gives an overview of the RISC-V ISA focusing on the implemented instruction sets in Klessydra-T, and the custom instructions appended to the native RISC-V ISA.

Chapter 3 The third chapter provides an overview of the PULPino SoC, and describes the modifications made to the Pulpino environment that made it possible for Klessydra to be integrated.

Chapter 4 The fourth chapter introduces the Klessydra-T0. In this chapter we investigate the optimal pipeline organization to adopt through a series of experimental and analytical studies. Then the building blocks of the Klessydra-T0 will be illustrated, and then we show some basic libraries written to compliment the hardware side with some software code.

Chapter 5 The fifth chapter introduces the Klessydra-T1, and shows the hardware accelerator added to the T1 core. Then the accelerator is benchmarked when implemented in three different approaches, and we deduce which approach is the most ideal to use. The accelerator is benchmarked with VGG16 DCNN test, and it is shown how it was benchmarked

Chapter 6 The sixth chapter just shows the software suite of the tests that were used to benchmark the accelerator in chapter 5. They demonstrate how the convolutions were implemented on the accelerator, and a brief display of how the different structures in the VGG16 test were written.

Conclusion We conclude by summarizing the results presented in chapters four and five.

Appendix A Contains the Klessydra technical manual detailing the implementation, the ISA support, the architecture, and the CSR instructions in the Klessydra-T cores.

Appendix B This RTL of the Klessydra-T is here. The T1 and the T0 implementations can be generated from the PKG file, as well as all the configurations detailed in chapter 5.

Chapter 1 Preface

This chapter is a preface to the work being detailed in this study. In the first section we provide a brief introduction on IoT devices and their growth in the current electronic market. In the next section we discuss the artificial neural networks, focusing on the sector of computer vision and convolutional neural networks. In the last section we show the convergence of AI applications from cloud computing to embedded low power IoT devices. Then we discuss the energy efficient digital system developed in this study that target the IoT market, and facilitates the execution of CNNs that are being steadily embedded in IoT devices.

1.1. Internet of things

The MOSFET was the main driver for the rise of the Internet of things. The scaling of the MOSFET down to the nanoscale was also followed by the scaling down of the power consumption as well. As of 2019, the smallest MOSFETs in production are 5nm FinFETs manufactured by Samsung and TSMC [1][2]. Gordon Moore observed the shrinking of the transistor and predicted that the number of transistors on an integrated circuit would approximately double every two years (figure 1.1) with the speed doubling every 18 months without increasing the power [3].

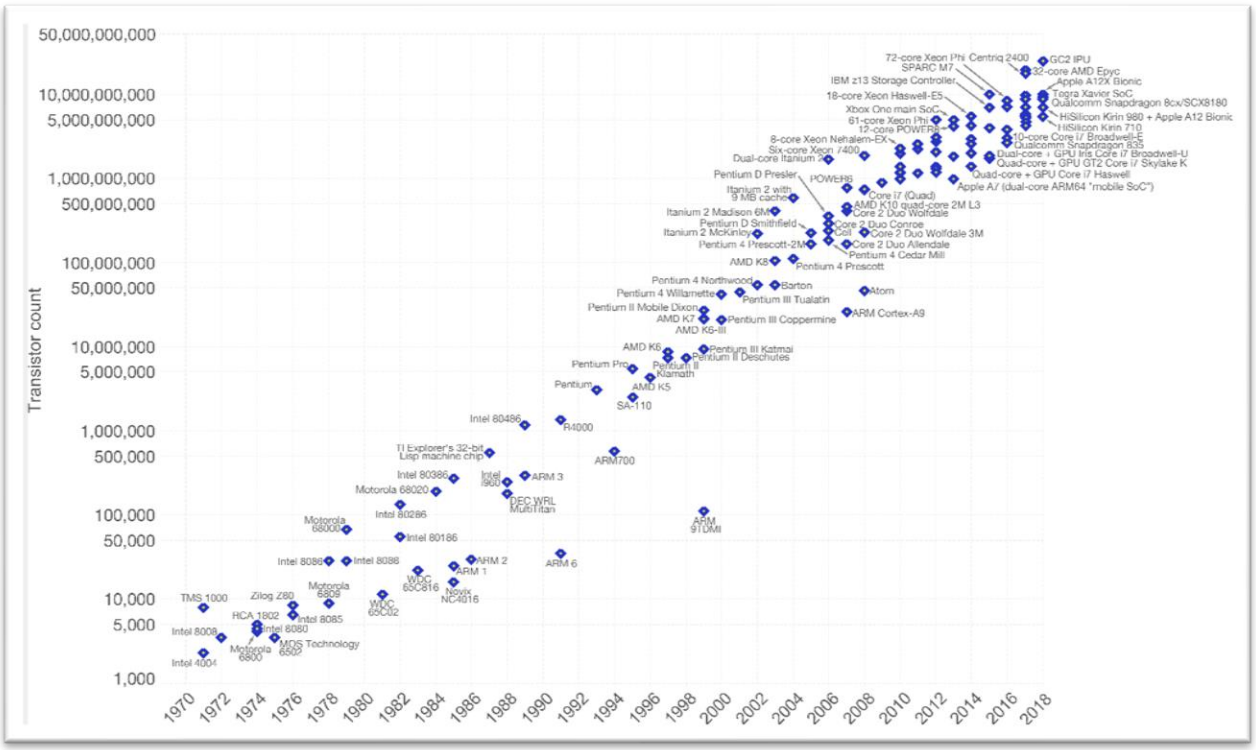


Figure.1.1, Graph depicting Moore's Law that predicted the doubling of the transistors per die every two years

However, the world was still farfetched from becoming fully connected. Two main inventions provided the next milestone that facilitated the convergence towards an IoT world, the first was the development of high-performance multi-core processors, and the second was the emergence of high bandwidth wireless technologies.

The nanoscale scale parallel microprocessors were capable computing large chunks of data for a very slight energy consumption. This in turn encouraged the incorporation of these smart technologies into all types of electronic devices especially inside battery powered devices (figure 1.2). One main

example for the use of these smart devices other than home automation domains as shown in the figure above was the deployment of smart devices for sensing and monitoring tasks, such as office monitoring, agricultural monitoring, traffic monitoring, defense monitoring, space monitoring, and not to mention even human monitoring through medical devices and wearable technologies. These areas were situated with a handful of sensing instrumentation for temperature, humidity, fire, air pollution, traffic jam, rain wind, storms, etc.) [4].

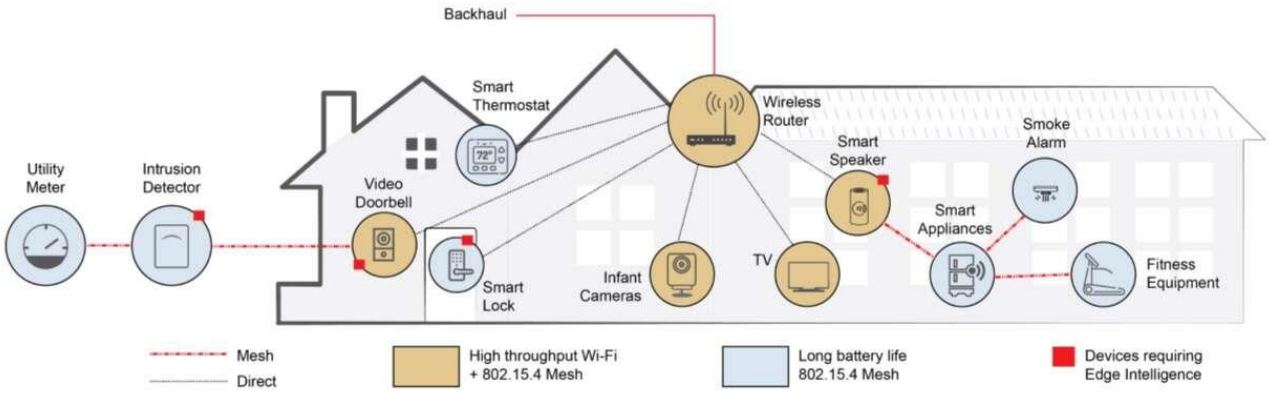


Figure.1.2. Typical IoT devices in homes

However, these smart devices needed to be accessed over long distances. and this is where the emergence of wireless technologies played a key role in which they were capable of providing a connection between two nodes over large distances. But one main drawback to wireless transmission was that; the larger the distance got between the two nodes; the more transmission power was needed to maintain the nodes connected. Another challenge was the exorbitant increase in the bandwidth over the years, required by certain streaming applications, and in order to provide these large bandwidths, the wireless technologies needed to transmit over higher frequencies in the spectrum as shown in figure 1.3.

Figure.1.3. The bandwidth growth with the frequency growth

However, the power consumption required to transmit a certain packet of data over a certain distance 'X' is much higher than the power consumption required to transmit the same packet over a lower frequency, and figure 1.3 showed that larger bandwidths broadcasted at higher frequencies. The tradeoff between coverage area and frequency when transmitting over the same frequency is shown in figure 1.4.

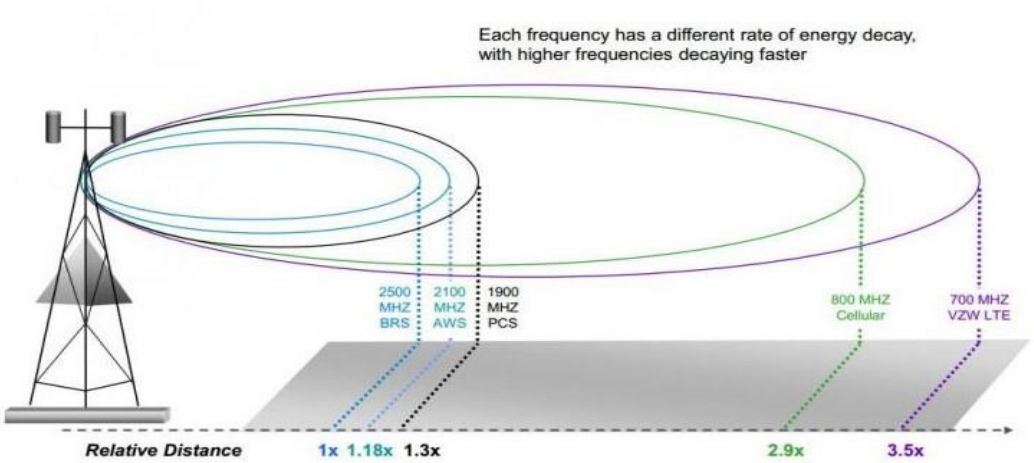


Figure.1.4. Coverage area for a set of transmission frequencies

Figure 1.4 shows that coverage area for transmitting over the same power (dBm), but different frequency ranges was very different. Such that transmitting over 700MHz covered the 3.5 times the distance for transmitting over 2.5GHz.

The challenge was to accommodate the demand to transmit high bandwidth of data over very large distances, while still maintaining low power consumption. Thus, came the third milestone which was connecting these smart devices to local gateways either through a wire or wirelessly, and the gateways are connected to a global system of interconnected nodes communicating with an open protocol; called TCP/IP otherwise known as the internet.

Providing internet connectivity to smart devices made them capable of transmitting very high data bandwidths over high frequencies to local wireless nodes that are only a few meters away from the transmitter. These communicating nodes are otherwise known as wireless local area networks (WLAN). The WLANs are then connected to the internet and provide access to these smart devices globally. This connection of the various smart devices from over the internet is what is now known as the Internet of Things (IoT).

However, not every device that has IP connectivity is considered IoT. For example, desktops, laptops, cellphones, tablets, game consoles are not considered to be IoT [27]. An IoT device is a network of devices that can communicate without human interactions. In other words, it is a network of things. Figure 1.5 shows the number of IoT devices available till date, and their projected growth over the next five to six years.

IoT encompasses only device-to-device interactions and connectivity. Although human interaction can be present at some endpoint of the IoT network, but all the intermediate device communications are considered IoT. For example, a wearable smart watch interacts with the cellphones over wireless personal area networks (WPAN), and cellular mobile stations through LTE, and connect to GPS systems to provide continuous tracking. All these communications are part of the IoT network, and the final presentation to the human interface would be the non-IoT human factor in this network [27].

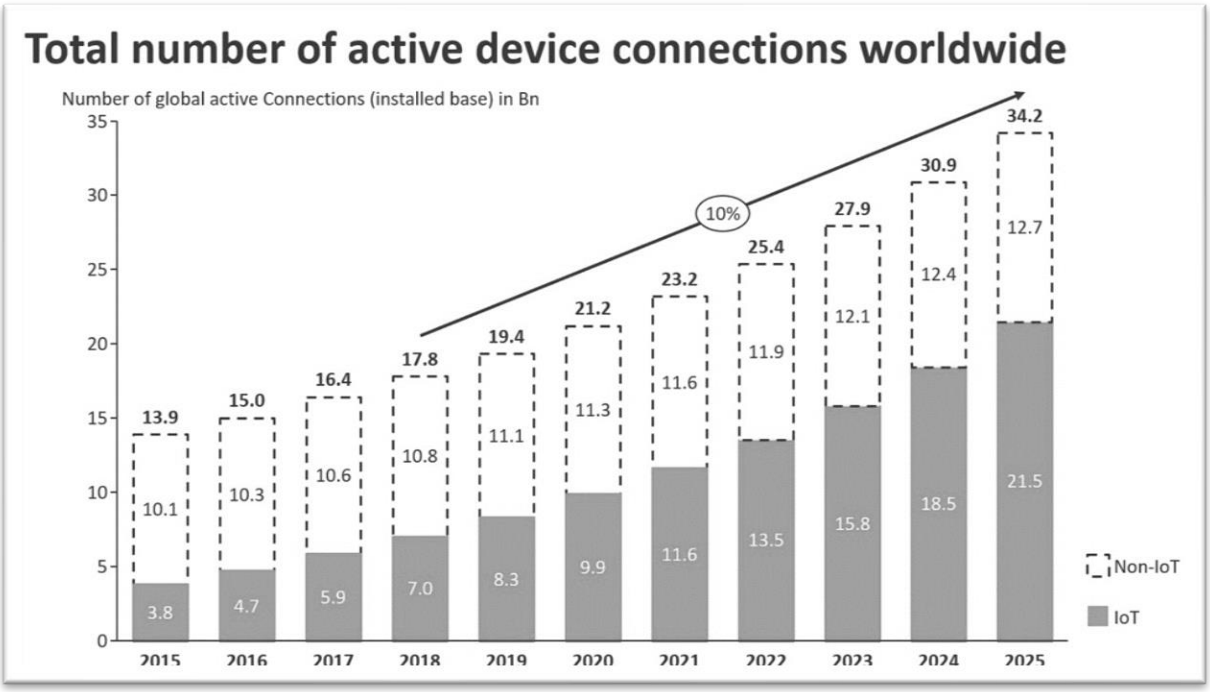


Figure.1.5. Number of IoT devices to non-IoT and their project growth

1.2. Energy efficient IoT devices:

Gradual increase in the integration of convolutional neural networks in low power embedded IoT devices by applying image recognition and classification was prevalent in the recent years [5]. IoT devices were able to move AI algorithms from cloud computing down to the edge computing [6]. IoT endpoint SoC refer to a large number of microcontrollers interfacing a various class of sensors on one end, and a wireless device on the other end. The IoT end-nodes might contain specialized units for fast memory access such as scratchpad units [22]. The IoT end-node design demands low-power specialized processors [24][25][26], in which they will be used to collect and pre-processes information from the peripheral devices, and sends the data over the wireless channel (figure 1.6). Preprocessing might include in many cases speech and/or image recognition. This is why we developed a RISC-V processor that can exploit IoT applications which interface multiple peripheral devices, and also, can pre-process images quickly with high performance and energy efficient CNN accelerators.

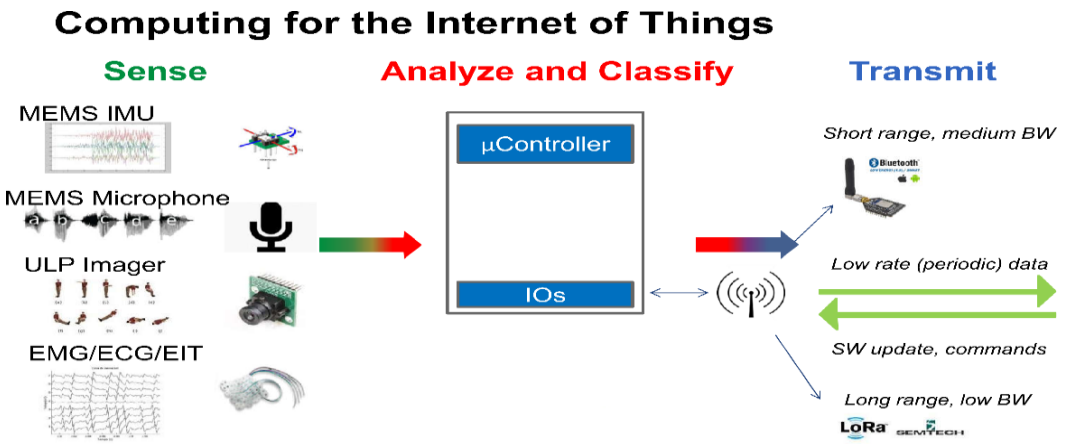


Figure.1.6.Typical depiction of an IoT Embedded System

1.3. Artificial neural networks

1.2.1. **Background**:

The human brain is a collection of billions of neurons connected to each other through synapses and can pass the signal from one neuron to the next either electrically or chemically. Artificial neural networks although not identical to biological neural networks, however, they were inspired by them. They aimed to loosely imitate the behavior of the brain in order to solve some of the problems the brain does through emulating its learning ability.

ANNs are a collection of artificial neurons that connect to one another to form a large system of artificial neurons. These systems are an aggregate of layers that are connected to each other, they are capable of learning through continuous feedback loop connections, or through algorithms in single-forward pass networks that modify the weights after the whole operation is done (such as the case in feed-forward networks like convolutional neural networks). During the learning process, the system adjusts the weights which can either strengthen or weaken the connection between the two neurons. Figure 1.7 shows the basic organization of an ANN.

The first layer takes the external data that is known as the input layer, and performs a transformation of these data and sends its output to the next layer. The final layer of the networks is the output layer that infers the final result from all the transformations of the previous layers. Between the input and the output layers, there might exist some intermediate layers also known as hidden layers (figure 1.7).

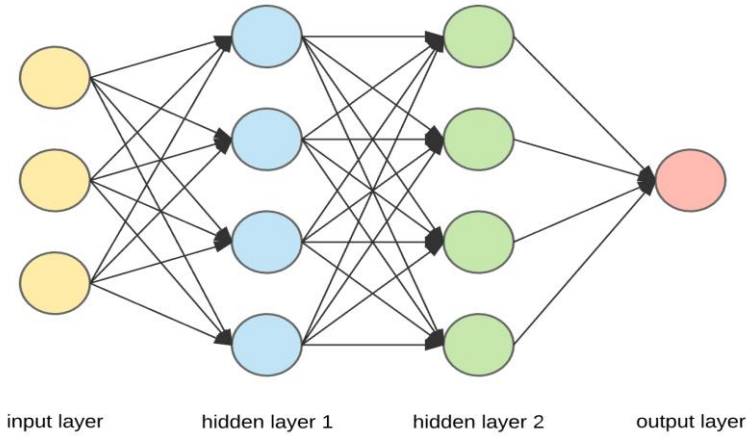


Figure.1.7. Layers in an artificial neural network

The layers can be fully-connected by having every neuron in layer[i] connect to every other neuron of layer[i+1], or the connections can be pooling by connecting a set of neurons in layer[i] to a single neuron in layer[i+1] thereby reducing the number of neurons in layer[i+1].

1.2.2. Learning in ANN

Learning is a continuous process of adjusting connections between the neurons by modifying the weights, so that the output results will converge towards the correct output after running the network in each iteration. The learning can be considered complete if the error rate ideally becomes zero, or that if each iteration of running the network does not reduce the error rate. In order to try and avoid oscillations of weights inside the neural network during learning, adaptive learning must be implemented to in order to maintain a gradient ascent or descent of the weights.

Final results of the network are mapped into a probability distribution of predicted outputs by using normalizing functions such as *softmax*. However, the actual output might not be the desired output.

The error rate in ANN does not typically reach zero, even after the learning is done. A cost function maps the desired real results to the actual results, and if the error rate determined by the cost function is deemed too high, then the network is basically is not designed very well, and re-designing it must be put into consideration.

1.2.3. Deep Convolutional Neural Networks and Deep Learning

A deep neural network (DNN) is a subset of ANN where there exists a large number of layers between the input the and the output layers. The extra layers in DNN enable the extraction of features from the previous layers. DNN are feedforward in nature. They do not provided feedback to the previous layers, and the adjusting of the weights is done at the end of network after the probability distribution has been calculated.

One of the main fields of DNN is convolutional or deep convolutional neural networks (CNN / DCNN), they are used in computer vision [28], or speech recognition. CNNs are fully-connected networks in which each neuron in one layer connects to all the neurons in the next. CNN employ mathematical convolutions in order to transform the input data into the output. There are a large class of CNN that were developed over the years. Figure 1.8 arranges them in accuracy versus number of operations in a single forward pass. One single forward pass indicates how many operations (G-OPS) are required in order to transform the input data of the network to the output result. The size of the circles indicates the memory footprint of each network.

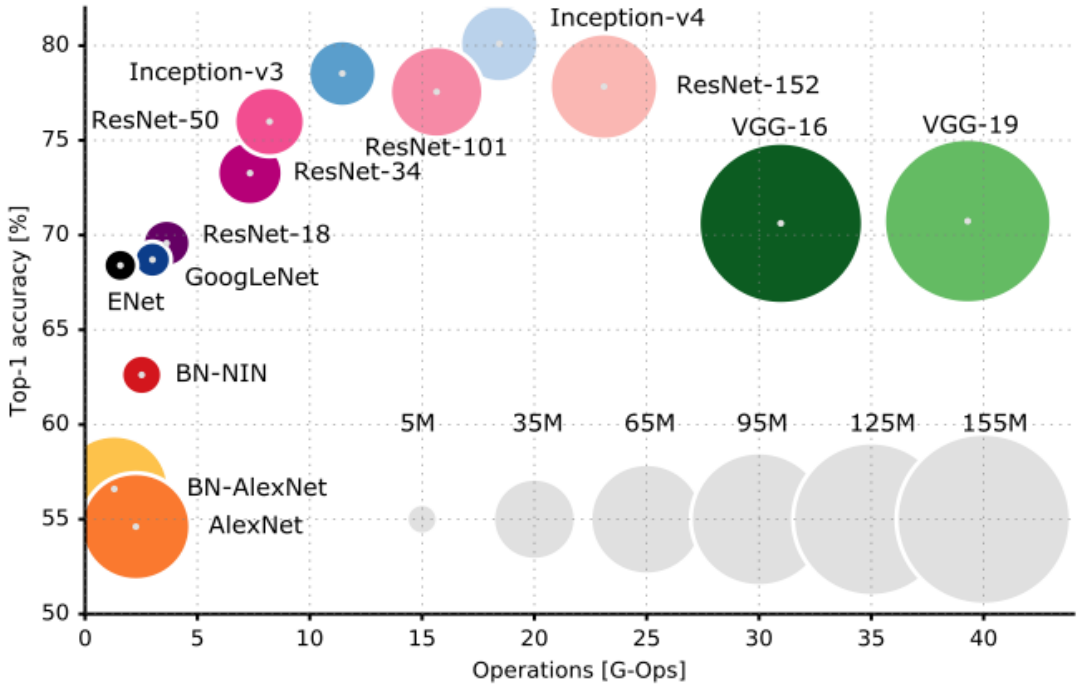


Figure.1.8. Accuracy versus number of operations single forward pass for a certain class of CNN

Chapter 2 RISC-V and the Klessydra Processor Family

2.1. Motivation behind adopting RISC-V

The first step in building Klessydra a majorly open source family of processing cores, was through choosing an instruction set. Our choice in that matter considering we are a group of researchers with limited funding was to adopt an open instruction set free from royalties.

Our motivation for adopting the RISC-V instruction set, was basically similar to the motivation of the team from University of California, Berkley when they developed the RISC-V ISA. Which was to make instruction sets free. Another reason encouraged us was that RISC-V was designed to tailor and exploit all types of architectures. In-order, out-of-order, embedded low-power, supercomputers and etc. The third reason was that, RISC-V providing encoding space for custom instructions, helped flourish the research community by allowing students, researchers and industries to test, and experiment their own non-native instruction sets.

Also, comparing both RISC-V and OpenRISC, RISC-V being a more revised and well-studied ISA made the case that they were a better option to adopt than OpenRISC for several reasons, most importantly is that openRISC supports condition codes and branch delay slots which complicate higher performance implementations. Also, OpenRISC supporting fixed sized 16-bit immediates made little encoding space to let the ISA grow.

2.2. Background

RISC-V is an open instruction set architecture, the project was started in 2010 at the University of California, and it still continues to expand the ISA specification till the present day.

The ISA is based on reduced instruction set computing, and it provides two reference manuals. The first being the user-level ISA, and the second being the privileged architecture [7]. The main motivation behind having an open source instruction set, was the availability of the open source Linux operating system, and the open networking protocols TCP/IP [8]. The question came as to why instruction sets cannot be free as well. This motivated the engineers at Berkley to create an ISA being open and royalty free. Commercial ISAs from Intel, ARM, and IBM being proprietary limited the research in computer architectures to those companies themselves. And in order to adopt the standards, one must undergo a rigorous process of negotiations in order to arrive at an agreeable price for adopting the proprietary standards, and the process is reported to take about six to twenty-four months.

RISC-V till date supported the computer architecture research and education consortium in developing their own proprietary or open-source processors. Currently there are tens of RISC-V implementations, like Rocket, RI5CY, Ariane, Klessydra, BOOM, Taiga, and many more [9]. One of their main future goals is to have the instruction set adopted also in industry implementations.

In the next sections in this chapter we will make a brief summary or the RISC-V instruction sets, then we will discuss one huge advantage provided by RISC-V that enabled researches to innovate even more in the computer architecture domain, by giving more implementation freedom to the user. Finally, we will discuss which architecture and ISA extensions were adopted in the Klessydra-T cores presented in this thesis.

2.3. Instruction set architecture briefing

The RISC-V ISA is the base integer ISA, which must be defined in any implementation. The base integer ISA is the backbone of the entire standard that delivers a minimal set of instructions sufficient to be provided to compilers, linkers, assemblers, and operating systems. The base integer ISA can be implemented for both 32-bit and 64-bit architectures.

The base integer ISA is labeled "I" and is preceded by either one of the following labels. "RV32" or "RV64". It supports 32 general purpose registers from "x0-x31" with "x0" being a read only register hardwired to 0. Table 2.1, shows the application binary interface (ABI) of the integer and floating point registerfiles.

Table.2.1. RISC-V mnemonics for RISC-V integer and floating point registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

The return address register "x1" is not hardwired automatically in function calls, but rather jump instruction branching to call environments use register "x1" by default to hold the return address. The stack pointer "x2" is identical to each hardware thread or core, and in RISC-V it always points to the beginning of the stack, and the loads and stores to the stack are relative to the base address (i.e. stack pointer in this case).

The base ISA has four instruction formats, as shown in figure 2.1. All instructions have a fixed length and must be aligned 32-bit aligned.

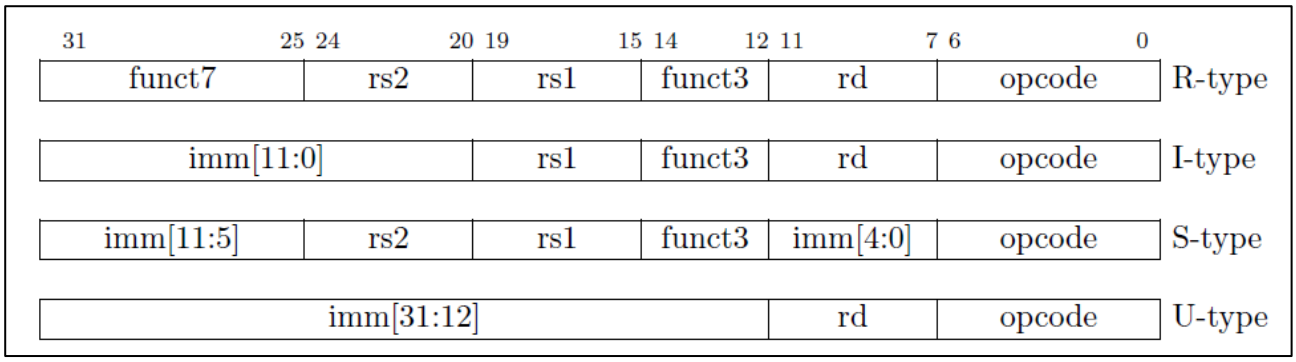


Figure.2.1. Base Instruction Formats

The source *rs1, rs2* and destination *rd* operands always fixed in their positions in order to keep the decoding simple. The immediates are always sign extended except for CSR immediates.

The base ISA is divided into five categories of instructions:

- The **integer computational instructions** have a subset of arithmetic, logic, and shifting operations. That either in majority the I-type or R-type format. LUI/AUIPC use the U-type.
- The **control transfer instructions** have a subset of conditional and unconditional jumps. Conditional jumps are relative to the program counter, and do not link any registers. Unconditional jumps can behave like a *goto* statement if there are no pushes to the return address stack (RAS), or they could behave like function calls, or function returns by pushing and popping to the RAS (Table 2.2).

Table.2.2. RAS stack prediction hints

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	push and pop
link	*link*	1	push

- The **load and store** instructions get the memory address by adding the base address stored in rs1 to the Immediate in the instruction. Load instructions have the I-immediate, and Store use the S-Immediate. They can fetch/write bytes, half-words, and words.
- The **memory fence** instructions insure that one hart performs its memory access before the other hart by fencing the memory accesses.
- The **control and status instructions** access the CSR registers, and modify the ones that are not read only. A large subset of these are registers used for performance counting.
- The last are **environment call and break points** which transfer the execution to a more privileged environment or to a debugger.

RISC-V supports more extensions that include operations being ubiquitous in the computing world. They include the M-extension for Multiply/Divide, A-extension for Atomic operations that help ensure thread synchronization, and memory region locks, F/D-extension for single and double floating-point instructions, and many more that are still being drafted.

2.4. Custom instruction set extensions

RISC-V has been designed to support extensive customization by providing encoding space for custom-instructions as shown in table 2.3. Any custom implementation is considered to be a part of the *non-standard* extensions. The following table shows the map of the base 7-bit opcode and the spaces reserved for each opcode.

Table.2.3. RISC-V based opcode map, inst[1:0] = 11 i.e. compressed instructions are not included in the table

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

As seen from table 2.3 the are four base opcode spaces reserved for custom instruction extensions: *custom-0, custom-1, custom-2,* and *custom-3*.

2.5. RISC-V support in Klessydra

All Klessydra implementations till date support the "I" base integer instruction set in 32-bit. The introduction of the later multithreaded Klessydra-T0 required at least minimal support of the atomic extensions, by implementing the *AMOSWAP* instruction from the A-extension. The Klessydra-Fx implementation continued to support multithreading thus maintaining the atomic support. Also. the M-extension has been augmented in later releases to provide fast multiplication, especially in the Klessydra-T1 to help execute small vectors quickly in convolutional neural networks.

As for the custom instruction set augmentation, they were included only in the Klessydra-T1, they base opcode encoded for the custom instruction was as follows:

- Custom memory instructions encode the opcode space reserved for "*custom-0"*, the opcode[6:0] being "7b'0001011 "
- Custom vector arithmetic instructions encode the opcode space for "*custom-1"*, the opcode[6:0] being "7b'0101011".

Table 2.4 shows the augmented instructions in Klessydra-T1, and their description will be found in appendix A.

Table.2.4. Klessydra K custom instruction set extensions

Name	Binary format	Assembly syntax	Opcode
KMEMLD	R	kmemld rd, rs1, rs2	*custom-0*
KMEMSTR	R	kmemstr rd, rs1, rs2	*custom-0*
KBCASTLD	R	kaddv rd, rs1, rs2	*custom-0*
KADDV	R	kaddv rd, rs1, rs2	*custom-1*
KSUBV	R	ksubv rd, rs1, rs2	*custom-1*
KVMUL	R	kvmul rd, rs1, rs2	*custom-1*
KVRED	R	kvred rd, rs1, rs2	*custom-1*
KSVADDSC	R	ksvaddsc rd, rs1, rs2	*custom-1*

KSVADDRF	R	ksvaddrf rd, rs1, rs2	*custom-1*
KSVMULSC	R	ksvmulsc rd, rs1, rs2	*custom-1*
KSVMULRF	R	ksvmulrf rd, rs1, rs2	*custom-1*
KDOTP	R	kdotp rd, rs1, rs2	*custom-1*
KDOTPPS	R	kdotpps rd, rs1, rs2	*custom-1*
KSRLV	R	ksrlv rd, rs1, rs2	*custom-1*
KSRAV	R	ksrav rd, rs1, rs2	*custom-1*
KRELU	R	krelu rd, rs1, rs2	*custom-1*
KBCAST	R	kbcast rd, rs1	*custom-1*
KVCP	R	kvcp rd, rs1	*custom-1*

In addition to instructions, also custom CSR registers were added, table 2.5 lists the custom CSR registers.

Table.2.5. Klessydra K custom CSR extensions

Name	CSR_Addr	TYPE	Reg_Size	Description
MVSIZE	*0xBF0*	R/W	Log_2(SPM_Size)	Contains the vector size the maximum being the SPM size
MVTYPE	*0xBF8*	R/W	2-bits	Contains the type of data the vector has (8-bit, 16-bit. 32-bit)
MPSCLFAC	*0xBE0*	R/W	5-bits	Post scaling factor for right shifts (used by kdotpps instruction)

2.6. Patches to the riscv-gnu-toolchain:

Two simple modifications were to be made, to the sources in the RISC-V GCC toolchain [35], the first was to "*riscv-opc.c*", where it had all the structures of the RISC-V instruction listings. As seen below:

```
1  /* Vector Extensions */
2  {"kmemld",    "I",  "d,s,t",     MATCH_K_MEMLD , MASK_K_MEM , match_opcode, 0 },
3  {"kmemstr",   "I",  "d,s,t",     MATCH_K_MEMSTR, MASK_K_MEM, match_opcode, 0 },
4  {"kbcastld",  "I",  "d,s,t",     MATCH_K_BCASTLD , MASK_K_MEM , match_opcode, 0 },
5  {"kaddv",     "I",  "d,s,t",     MATCH_K_ADDV, MASK_K_ARITH, match_opcode, 0 },
6  {"ksubv",     "I",  "d,s,t",     MATCH_K_SUBV, MASK_K_ARITH, match_opcode, 0 },
7  {"kvmul",     "I",  "d,s,t",     MATCH_K_VMUL, MASK_K_ARITH, match_opcode, 0 },
8  {"kvred",     "I",  "d,s",       MATCH_K_VRED, MASK_K_ARITH | MASK_RS2, match_opcode, 0 },
9  {"kdotp",     "I",  "d,s,t",     MATCH_K_DOTP , MASK_K_ARITH, match_opcode, 0 },
10 {"ksvaddsc",  "I",  "d,s,t",     MATCH_K_SVADDSC, MASK_K_ARITH, match_opcode, 0 },
11 {"ksvaddrf",  "I",  "d,s,t",     MATCH_K_SVADDRF, MASK_K_ARITH, match_opcode, 0 },
12 {"ksvmulsc",  "I",  "d,s,t",     MATCH_K_SVMULSC, MASK_K_ARITH, match_opcode, 0 },
13 {"ksvmulrf",  "I",  "d,s,t",     MATCH_K_SVMULRF, MASK_K_ARITH, match_opcode, 0 },
14 {"ksrav",     "I",  "d,s,t",     MATCH_K_SRAV, MASK_K_ARITH, match_opcode, 0 },
15 {"ksrlv",     "I",  "d,s,t",     MATCH_K_SRLV, MASK_K_ARITH, match_opcode, 0 },
16 {"kbcast",    "I",  "d,s",       MATCH_K_BCAST, MASK_K_ARITH | MASK_RS2, match_opcode, 0 },
17 {"krelu",     "I",  "d,s",       MATCH_K_RELU, MASK_K_ARITH | MASK_RS2, match_opcode, 0 },
18 {"kdotpps",   "I",  "d,s,t",     MATCH_K_DOTPPS, MASK_K_ARITH, match_opcode, 0 },
19 {"kvcp",      "I",  "d,s",       MATCH_K_VCP, MASK_K_ARITH | MASK_RS2, match_opcode, 0 },
```

The second modification was made to the “riscv-opc.h”, where all the defines were made that include the instruction mask and instruction opcode, as well as the CSR defines.

```
/* Klessydra Extensions */

/* CSR Extensions */
#define CSR_MVSIZE 0xbf0
#define CSR_MVTYPE 0xbf8
#define CSR_MPSCLFAC 0xbe0

/* Vector Instructions Extensions */
#define MASK_K_MEM      0xfe00707f
#define MATCH_K_MEMLD   0xb
#define MATCH_K_MEMSTR  0x200000b
#define MATCH_K_BCASTLD 0x400000b
#define MASK_K_ARITH    0xfe00707f
#define MATCH_K_ADDV    0x200202b
#define MATCH_K_SUBV    0x400202b
#define MATCH_K_VMUL    0x800202b
#define MATCH_K_VRED    0xC00202b
#define MATCH_K_DOTP    0x1000202b
#define MATCH_K_SVADDSC 0x1800202b
#define MATCH_K_SVADDRF 0x1a00202b
#define MATCH_K_SVMULSC 0x1c00202b
#define MATCH_K_SVMULRF 0x1e00202b
#define MATCH_K_SRAV    0x2000202b
#define MATCH_K_SRLV    0x2200202b
#define MATCH_K_RELU    0x3000202b
#define MATCH_K_DOTPPS  0x3200202b
#define MATCH_K_BCAST   0x3c00202b
#define MATCH_K_VCP     0x3e00002b
```

```
DECLARE_CSR(mvsize , CSR_MVSIZE)
DECLARE_CSR(mvtype, CSR_MVTYPE)
DECLARE_CSR(mpsclfac , CSR_MPSCLFAC)

DECLARE_INSN(kmemld, MATCH_K_MEMLD, MASK_K_MEM)
DECLARE_INSN(kmemstr, MATCH_K_MEMSTR, MASK_K_MEM)
DECLARE_INSN(kbcastld, MATCH_K_BCASTLD, MASK_K_MEM)
DECLARE_INSN(kaddv, MATCH_K_ADDV, MASK_K_ARITH)
DECLARE_INSN(ksubv, MATCH_K_SUBV, MASK_K_ARITH)
DECLARE_INSN(kvmul, MATCH_K_VMUL, MASK_K_ARITH)
DECLARE_INSN(kvred, MATCH_K_VRED, MASK_K_ARITH)
DECLARE_INSN(kdotp, MATCH_K_DOTP, MASK_K_ARITH)
DECLARE_INSN(ksvaddsc, MATCH_K_SVADDSC, MASK_K_ARITH)
DECLARE_INSN(ksvaddrf, MATCH_K_SVADDRF, MASK_K_ARITH)
DECLARE_INSN(ksvmulsc, MATCH_K_SVMULSC, MASK_K_ARITH)
DECLARE_INSN(ksvmulrf, MATCH_K_SVMULRF, MASK_K_ARITH)
DECLARE_INSN(ksrav, MATCH_K_SRAV, MASK_K_ARITH)
DECLARE_INSN(ksrlv, MATCH_K_SRLV, MASK_K_ARITH)
DECLARE_INSN(krelu, MATCH_K_RELU, MASK_K_ARITH)
DECLARE_INSN(kdotpps, MATCH_K_DOTPPS, MASK_K_ARITH)
DECLARE_INSN(kbcast, MATCH_K_BCAST, MASK_K_ARITH)
DECLARE_INSN(kvcp, MATCH_K_VCP , MASK_K_ARITH)
```

2.7. Concluding remarks

In the end RISC-V is not only an open source ISA available for simulations, it is a real ISA suitable for inherent hardware implementations. The standards were provided to be balanced to be exploited by all types of architectures. It supports 32 and 64-bit address space and IEEE standard floating-point standards, it provides custom instruction encoding space to allow researchers to explore native non-standard custom extensions, or companies to integrate their own specialized instructions and finally it still has a great potential to become even more pervasive throughout the industry.

Chapter 3 The PULPino Microcontroller Platform

3.1. Motivation behind choosing PULPino

Having already chosen to build a RISC-V processor required also choosing a SoC. Designing our own SoC from scratch was not feasible since our group of researchers were limited. RISC-V being an emerging technology, the choices among the open SoCs available were not many. Pulpino being part of the ultra-low power projects also was a good reason to adopt the Systen. Finally, having close relations and collaborations with the University of Bologna, provided an ongoing communication channel in order to get continuous support from their side. For the above reasons, we can say that Pulpino was our choice.

Pulpino is an open-source System-on-Chip embedding a 32-bit RISC-V based microprocessor. Pulpino targets embedded systems and embeds ultra-low power designs. The Pulpino SoC was adopted by a large group of researchers globally either for research or commercial purposes.

3.2. Background

PULPino is a smaller version of PULP which stands for Parallel Ultra Low Power processor. The idea behind starting the PULP project, was that in order to achieve low dynamic power consumption, the processors needed to be operated at near threshold voltage levels [10]. The speed will drop rapidly when operating at near threshold voltages since the delay follows a quadratic curve (figure 3.1). Their solution was to re-ramp up the speed by embedding several processors in PULP to work in parallel.

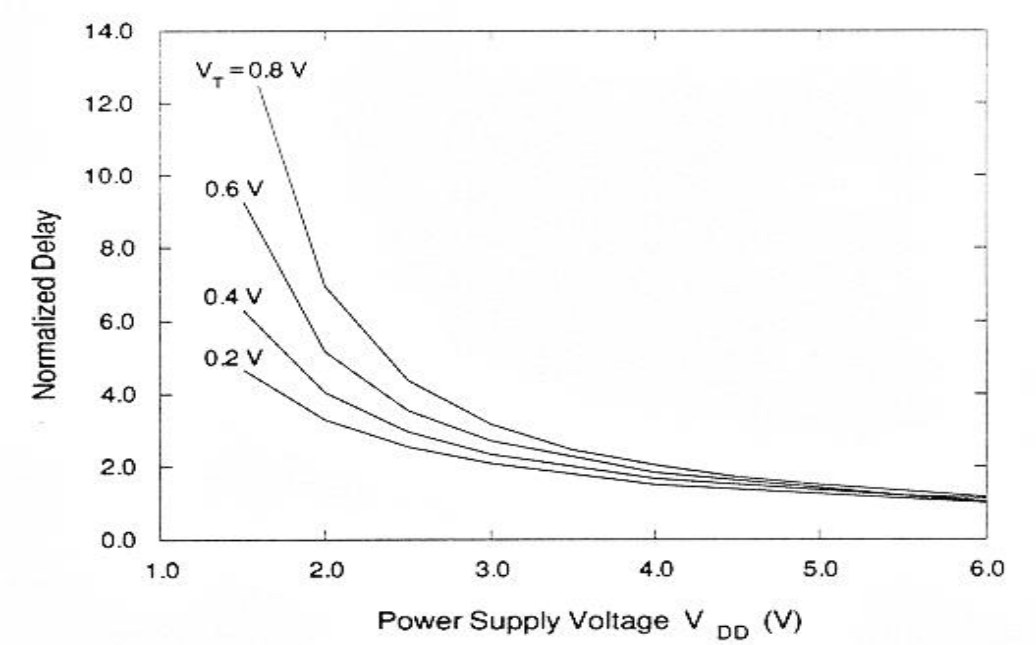


Figure.3.1. Propagation delay versus power supply voltage

PULP is a large project with a very wide scope of work, it incorporates a large group of engineers, and specialized experts. The project includes open source processors, peripherals, communication buses, an integrated all-in-one environment to build and test the embedded cores with Modelsim and Vivado and the entire SoC, also adds a custom RISC-V toolchain.

PULPino is a miniaturized version of PULP which embeds only one core. PULPino is completely open source[17][18], and can be found on GitHub. Figure 3.2 shows the building blocks of PULPino.

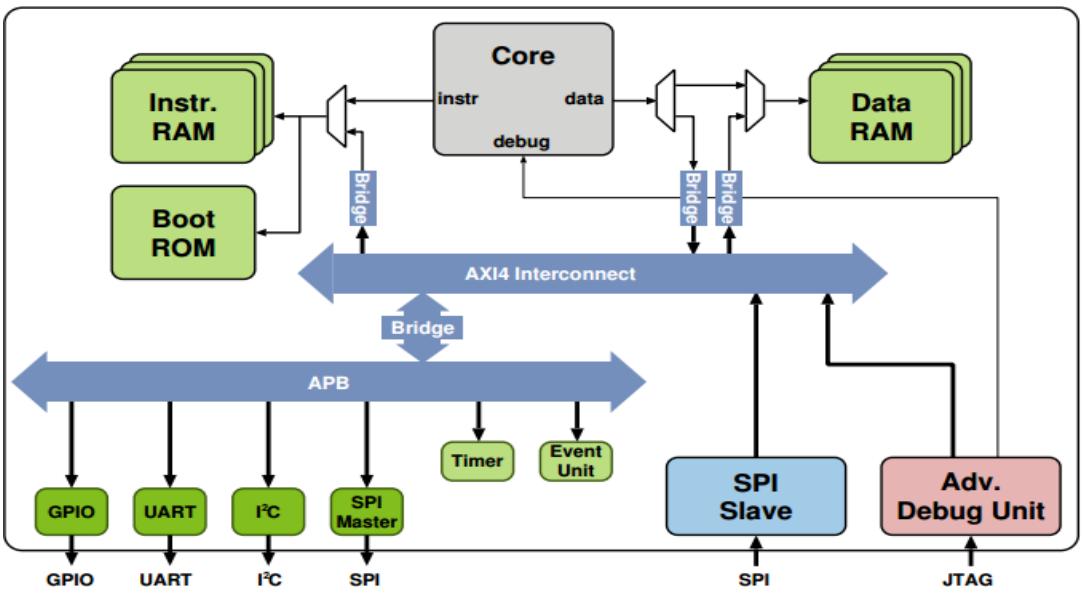


Figure.3.2 Architecture of PULPino

Pulpino targets RTL simulations, FPGAs, and ASICs. It has by default a 32KB program memory, and a 32KB data memory. The boot ROM is 512B. Peripherals are mapped in the upper region of the core and are dedicated 4KB each. The peripherals in Pulpino communicate through sending interrupts. All the interrupts are saved in an interrupt vector table (IVT). When servicing the interrupt, the core will check the IVT in order to jump to the appropriate interrupt handling routine.

Other than the Peripherals, it features an SPI Slave port that can be used to pre-load programs into the memories without the help of the core. It is connected on the AXI as an AXI master which allows external access to all memories and peripherals. Also, Pulpino has a JTAG debugging interface that accesses all peripherals and memories, and can halt and single step the core.

3.3. PULPino native processor cores

Pulpino integrates two RISC-V processors. They are RI5CY and Zero-Riscy. RI5CY is an in order four pipeline stage processors. It supports the base integer instruction set RV32I, compressed instructions RV32C, multiplication extension RV32M, and single precision floating point extensions RV32F. RI5CY also implements other extensions to the ISA such as hardware loops, bit manipulation instructions, MAC operations, packed SIMD instructions and many more [52][53].

Zero-Riscy is an in-order, single-issue processor with only two pipeline stages. It supports the base integer instruction set RV32I, the compressed instructions RV32C, and the multiplication extension RV32M. The core can be configured to support the embedded extension RV32E, and thus reducing the registerfile to half its size. A tiny version of zero-riscy can be implemented by enabling the

embedded extension (RV32E), and disabling the multipliers and dividers (RV32M). This implementation is called Micro-Riscy which is the smallest version supported.

3.4. Embedding non-native Klessydra processing cores in PULPino

Figure 3.3 shows the Klessydra and Pulpino Roadmap. Klessydra targeting FPGA implementations, while Riscy cores targeting ASIC implementations.

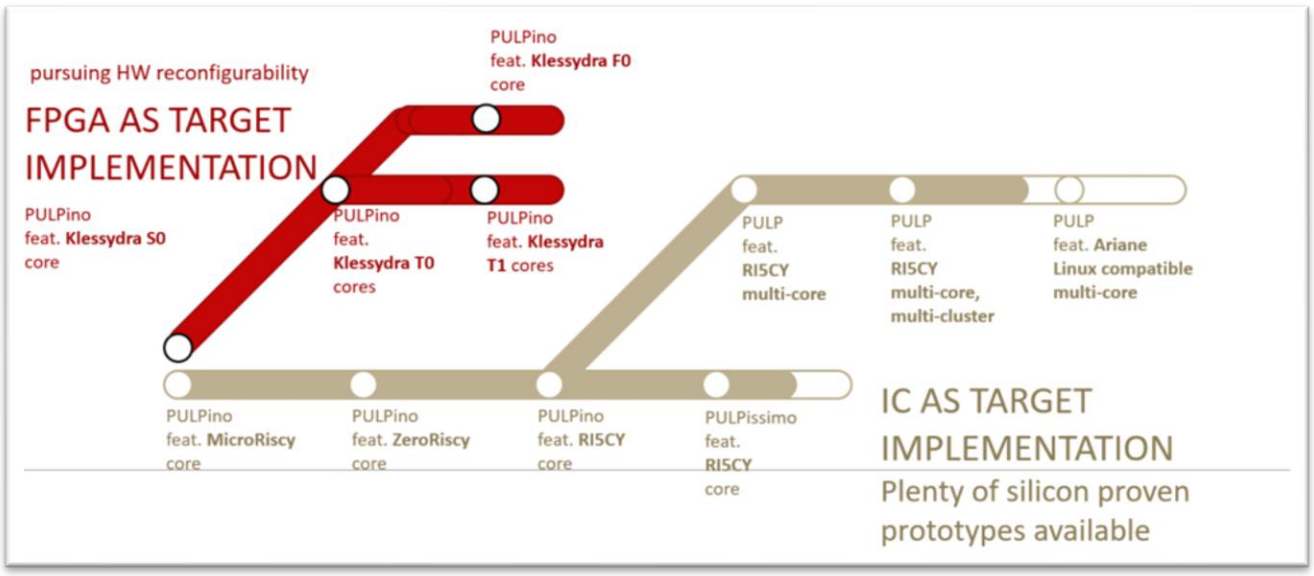


Figure.3.3 Klessydra family roadmap

In order to correctly embed Klessydra core and software libraries inside Pulpino, changes had to be made to the Pulpino environment on many levels:

- **Modifying the Klessydra RTL:** The pinout of the Klessydra was made one hundred percent compatible with the riscy cores from Pulpino. Also, the interrupt handling, and exception, and event handling had to be modified so that it passes the generic tests.
- **Modifying the Pulpino RTL:** The systemverilog of the Pulpino RTL and testbench were modified to add the instances of Klessydra cores, and pass the added generic parameter.
- **Modifying the Software Environment:** The CMake files were modified to include the generic Klessydra tests and software libraries. Also, they were modified along with a shell script in order to pass the arguments to the Tcl simulate scripts.
- **Modifying the Modelsim compile and Simulate scripts:** In addition to the software environment and RTL, compile scripts were also modified to compile the different versions of Klessydra among the compiled Pulpino libraries, and similarly the simulate scripts.

Chapter-4 Klessydra T0 Architecture

4.1. The Klessydra-T family

Klessydra is a processing core family that features full compliance with the RISC-V instruction set. Klessydra cores were designed in order to be fitted inside the PULPino SoC. The Klessydra family is composed of a single in-order two pipeline-stage core named Klessydra-S0 [11], a set of multithreaded cores named Klessydra-Tx, and a set of fault tolerant cores named Klessydra-Fx [20][21]. This thesus will cover the Klessydra-Tx family and its different variants. All the Tx cores have been synthesized and tested for FPGAs from XILINX. FPGA synthesis being our main target, was because soft-cores are wildly available on embedded systems [11]. A customizable embedded core is favorable since it can be reconfigured to adapt easily to the user's target applications.

Klessydra cores support RISC-V ISA, all versions support the base integer instruction set in 32-bit |"RV32I" in bare metal, the *Tx* and *Fx* versions extend the ISA with the atomic instruction extension, some *Tx* variants further extend the ISA with multiplication and division extension from RISC-V, and some augment a set of specialized custom instructions augmented to the RISC-V ISA designed to accelerate convolutional neural networking applications. The ports of the Klessydra cores are pin-to-pin compatible with the RISCY cores inside PULPino. The Tx versions of Klessydra support a multithreading paradigm called interleaved multithreading (IMT) also known as barrel processing. This chapter illustrates the early version of the Tx cores known as the T0 cores, and the different variants of the T0 cores. Chapter 5 upgrades the optimal T0 implementation adopted in this chapter and adds a specialized neural network accelerator that is specifically designed to exploit the IMT architectures. The upgraded version is known as the T1 core.

4.2. Motivation for choosing interleaved multithreading

A good guideline to follow in order to increase the energy consumption per instruction of an embedded processor, is through decreasing the idle time of the embedded systems by eliminating the pipeline stalls.

In-order architectures stall the processor's pipeline to fence between same-operand read and write access. These stalls are unfavorable as they degrade the performance of the processor, as well as decrease the energy efficiency by continuously accumulating the total idle time of the processor.

Out-of-order architectures can easily eliminate the pipeline stalls [49][50][51], however in order to do that, they employ highly complex dynamic scheduling logic to resolve the data dependency hazards. These data dependency eliminating schemes give rise to anti-dependency hazards, and again out-of-order architectures employ register renaming approaches to remove those anti-dependencies. In addition, these architectures being highly pipelined must integrate a well-advanced branch predicting logic, since branch miss prediction will greatly impact the overall performance. This type of architecture succeeded in greatly mitigating the pipeline stalls and improves the overall performance. However, these designs being very complex greatly increased the area and the power

consumption of those architectures. In other words, the performance was actually a tradeoff with the power and area.

One existing approach named barrel processing or interleaved multithreading (IMT) [16] aimed at replacing the out-of-order processor's highly complex approach to mitigate the pipeline stalls with another relaxed approach. That is by employing hardware threads to utilize the idle time of the core and fence between the registerfile read and write accesses.

An IMT architecture interleaves a hardware thread (hart) to fill the bubbles in the instruction pipeline in order to avoid Read-after-Write (RAW) data hazards. Doing so, it does not introduce a new class of anti-dependency hazards such as Write-after-Read (WAR) and Write-after-Write (WAW) as in the case of *OOO* architectures.

A basic IMT processors emulates a single-core single-issue processor with zero pipeline stalls. IMT processors with their ability to continuously issues instructions without data dependency stalls can converge easily towards 1 IPC in single issue processors, bit for a certain class of applications. The first class being decoupled sequential applications, and the second being balanced parallel applications. Regarding sequential applications, if the IMT processor was running in a way such that the programs are executing only on one hart and the other harts are idle, the overall performance will surely suffer from the overhead of the interleaving the other harts in the core, and the bigger the number of harts an IMT core has, the worse it performs when executing sequential code. A good practice that exploits the nature of such cores is to have every hart run its own sequential program. Such that the inputs data of one hart are completely independent from the output results of another hart. Such applications might include for example a microcontroller interfacing multiple sensors, and monitoring the changes, then transmitting the data over a wireless channel in order to be interacted by a human interface.

As for the second class of applications easily exploitable by IMT processors, one might quickly deduce that an IMT architecture can perform well in applications with parallel workloads. Although that is partly true, however, the evaluation of how an IMT core performs when running a parallel application is mainly dependent on how balanced the divided workload is between the harts.A balanced workload in a parallel program can have inter-thread dependencies that require thread synchronization; however, the nature of the workload being balanced makes the overhead of thread synchronization unnoticeable. If the parallel applications are balanced and loosely coupled, they will perform better than a balanced workload with tightly coupled applications. Such application classes are very much suitable for IMT architectures since they utilize all the interleaving harts very efficiently. There are many examples of such applications like; data sorting, searching algorithms, Monte-Carlo simulations, computational fluid dynamics (CFD) simulations, molecular modeling and simulations.

4.3. Klessydra-T0 introduction and background information

The Klessydra-T0 core is a basic IMT microprocessor which supports the RV32IMA instruction set extensions of RISC-V in bare metal. The 'T' symbol indicates that the core architecture is multithreaded. The multithreading paradigm supported is Interleaved Multithreading or IMT. The Klessydra-T0 can be parametrized to run without the M-extension, and also the registerfiles can also be parametrized to support the Embedded E-extension instead for area critical environments. Throughout this chapter, I will refer to the core as "T0" as an abbreviation to the name Klessydra-T0.

The T0 IMT is a single-issue in-order processor which is available in different variants, and the variants each of which has a different instruction pipeline organization, and they are designated by

the following abbreviation: "*T0ab*". Where the letter 'a' following the zero is the identifier for the minimum number of hardware threads needed to be interleaved in a core in order to avoid inserting any bubbles in the pipeline and this is known as the *thread pool baseline*. The 'b' identifier is to indicate the number of harts present in the current version of the core or otherwise known as *thread pool size.*

In order to build an IMT architecture, the following entities must be replicated for each hart:

- Registerfile
- Program Counter
- CSR Unit

After having replicated the above units, a hardware context counter "harc" must be built. The harc interleaves between the harts in the IMT core, such that on every instruction grant, we send to the program memory a request from another hart.

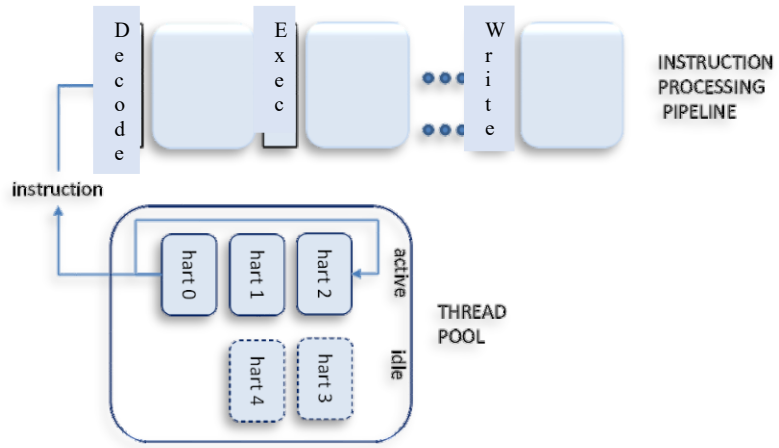


Figure.4.1. Conceptual view of hardware context counter (harc) interleaved execution

Klessydra-Tx cores have a parameterizable number of harts to interleave where the hart count is identified in the package file by a parameter called "THREAD_POOL_SIZE". The recommended number of harts to put in a core should be less than or equal to the thread pool baseline. In other words, *T0ab* is recommended to be configured such that 'b' is less than or equal to 'a'.

Configuring 'b' to be greater than 'a' is allowed, however, it will not give any performance boosts, rather it will significantly slow down the performance when running sequential applications. And running parallel applications as well degrade the performance by augmenting bigger stall overheads from idle harts, that will remain idle until all the other harts would have arrived at a thread synchronization barrier. Not to mention the area of the architecture will grow bigger, and as the layouts grow bigger, the elements in the FPGA selected during place and route will be placed ever so farther away from each other, which in turn will yield slower layouts resulted from larger net-delays between the FPGA element slices.

In order to know the minimum thread baseline needed so that no data hazards arise, we have to know how many pipeline stages exist from the read port of the registerfile till the write port of the registerfile. For every pipeline stage separating the read and write ports, a hart must be interleaved, else the user can choose to configure the core to have a hart count less than the minimum baseline and NOP operations will be introduced in the pipeline to fence between instructions belonging to the same hart.

4.4. Choosing the optimal IMT pipeline organization:

In this section, we will demonstrate the framework that followed in choosing the optimal pipeline organization to use in interleaved multithreaded processors [15]. In the end of the section we will show which *T0ab* organization was chosen as the most ideal processor to use in our research. This section is oriented around three main keywords:

- **TPS** or Thread pool size, which indicates the total number hardware threads in the core.
- **TPB** or Thread pool baseline that indicates the minimum number of harts needed to avoid data dependency stalls.
- **NT** or Number of active threads, which indicates the number of active harts M, in a core with a TPS equal to N, such that always: .

The exploration parameters of IMT architectures was first studied by implementing a set of pipeline organizations ranging from two stages to four stage [14]. each being run with a different set of thread pool sizes. The pipeline implementations studied were as follows:

a. ***F / RDEW (two pipeline stages)***
b. ***F / R / DEW (three pipeline stages)***
c. ***F / RD / EW (three pipeline stages)***
d. ***F / RD / E / W (four pipeline stages)***
e. ***F / R / DE / W (four pipeline stages)***

In the pipeline schemes listed above, *F* designates the instruction fetch stage, *R* is the registerfile reading, *D* is decoding, *E* is executing, and *W* is registerfile writeback. Early T0 versions included a fetch stage, and flushing logic to discard instruction of the same hart in the fetch when a branch is taken. However, later releases ignored the stage and the incoming instruction goes directly to the decode unit. The requested instruction goes directly to the stage after the *F.* These pipeline structures were designed to study the optimal pipeline organization to use in an interleaved multithreaded bare metal RISC-V processor. Synthesis runs were done on XILINX 7 Series FPGAs [3]. The synthesis timing constraints were set low to make the Vivado compiler generate fast netlists.

The FPGA element utilization from the synthesis runs of the set of configurations is shown in table 4.1. As well as the minimum cycle time of each layout. For instance, *T012* architecture has a TPS of 2 and thread pool baseline of 1.

Table.4.1. Resource Utilization, and Minimum cycle time [ns]

Architecture	TPS	Codename	LUT	FF	Tck
F / RDEW	2	T012	3264	2410	12.7
	3	T013	4018	3577	13.9
	4	T014	4351	4744	15.9
F / R / DEW	2	T022	3211	2544	8.9
	3	T023	3892	3711	9.7
	4	T024	4217	4882	9.5
F / RD / EW	2	T022_v2	3583	2653	9.6
	3	T023_v2	4461	3853	9.6
	4	T024_v2	4608	5052	9.4
F / R / DE / W	2	T032	3242	2679	8.6
	3	T033	4011	3914	8.9
	4	T034	4187	5144	8.6
F / RD / E / W	2	T032_v2	3635	2725	7.1

	3	T033_v2	4520	3958	7.3
	4	T034_v2	4825	5189	7.4

It is evident from table 4.1 that every increment of a hart (TPS) in the core, increased the number of flip-flops count by more than 1024 (32*32) registers. And every pipeline stage introduced increased the flip-flop count by 100~200 or 5% to 7%. For example, going from the pipeline organization T012 to T022 revealed only a 5% increase in the total flip-flop count and a slight decrease in the total LUT count, and going from the T023_v2 organization to T033_v2 increased the flip-flop count by 6% and the LUT count by 1%.

The cycle time of each organization is also shown in table 4.1. One concern we had was that the overhead of the interleaving new harts would increase the area utilization in the FPGA such that during the post-synthesis place and route phase, Vivado would place the elements very far away from each other, making the net delay of the critical path a lot bigger. However, the Vivado timing reports [48] only showed evidence to that situation happening in the *F/RDEW* pipeline organization where the cycle time increased from 12.7ns in the T012 to 13.9ns in the T013, and up to 15.9ns in the T014. However, we don't care about these implementations, since they were only control configurations used for comparative purposes to the other T0 pipeline organizations.

Looking at the other implementations shows only little cycle time increase due to interleaving more harts, and more significant cycle time decrease due to pipelining which is good. Hence, we conclude from the timing report that the increase overhead of adding a new hart to resolve the data dependency problems does not really impact the cycle time, and that with every pipeline the maximum frequency of the core keeps on increasing, such that the cycle time demonstrated a sharp drop from 12.7ns in the *T012* down to 7.4ns in the *T034_v2*.

The throughput of an IMT processor running an integer arithmetic application at maximum frequency is shown in table 4.2. The table shows the number of MIPS for each TPS configuration in every pipeline organization, when the active number of threads NT is less than or equal to the TPS.

- When , the number of MIPS suffers from data dependencies and pipeline flushes.
- When , the number of MIPS will suffer only due to pipeline flushes.
- When , the number of MIPS will not suffer from any pipeline flushes, and data dependency stalls. However, the MIPS will also not increase with the further increase of NT.

Table.4.2. Throughput at Maximum Frequency [MIPS] (N.A. = NOT APPLICABLE)

Architecture	TPS	TPB	Codename	Number of Active threads NT			
				NT=1	NT=2	NT=3	NT=4
F / RDEW	2	1	T012	67.9	78.8	n.a.	n.a.
	3		T013	61.9	71.8	71.8	n.a.
	4		T014	54.4	63.1	63.1	63.1
F / R / DEW	2	2	T022	69	96.4	n.a.	n.a.
	3		T023	63.6	88.8	103	n.a.
	4		T024	65	90.8	105.3	105.3
F / RD / EW	2	2	T022_v2	64.6	90.2	n.a.	n.a.
	3		T023_v2	64.2	89.6	104	n.a.
	4		T024_v2	65.6	91.6	106.2	106.2
F / R / DE / W	2	3	T032	50.8	74.6	n.a.	n.a.
	3		T033	49.1	72.2	100.8	n.a.
	4		T034	50.6	74.3	103.8	120.4

	2		T032_v2	58.8	86.4	n.a.	n.a.
F / RD / E / W	3	3	T033_v2	57.4	84.3	117.7	n.a.
	4		T034_v2	56.6	83.1	116	134.6

Not applicable are set in cases where NT is greater than TPS (NT >TPS), which is impossible.

Let's study one example from the table above. Take a look at the T023_v2 implementation, this implementation has a TPB of 2. When NT is equal to 1, the number of MIPS reported shows the throughput of the core that is affected by data dependencies and pipeline flushing, while setting NT to be equal to TPB which is 2, shows the throughput with stalls only due to pipeline flushing, and as NT becomes greater than TPB (i.e. NT=TPS=3), the pipelines in the core will **only have one instruction per hart** at a given time, thus making pipeline flushing unnecessary, and so the throughput maximizes to the top attainable values.

Table 4.3 and table 4.4 report the average dynamic power consumption when running at the maximum frequency for each implementation an integer arithmetic application, and the average energy efficiency of the processor to execute one instruction. Static power consumption was not reported, since FPGAs consume a constant static power independent of the parameters or test, they are running. Also, the designs do not provide any ad-hoc mechanisms to reduce the leakage currents [11][12].

Table.4.3. Average Dynamic Power at Maximum Clock Frequency [mW] (N.A. = NOT APPLICABLE)

Architecture	TPS	TPB	Code-name	Number of Active threads NT			
				NT=1	NT=2	NT=3	NT=4
F / RDEW	2	1	T012	43.57	45.67	n.a.	n.a.
	3		T013	38.44	40.29	40.29	n.a.
	4		T014	37.2	38.99	38.99	38.99
F / R / DEW	2	2	T022	53.43	58.43	n.a.	n.a.
	3		T023	46.77	51.14	53.61	n.a.
	4		T024	44.08	48.2	50.53	50.53
F / RD / EW	2	2	T022_v2	45.72	50	n.a.	n.a.
	3		T023_v2	45.44	49.69	52.08	n.a.
	4		T024_v2	38.98	42.63	44.68	44.68
F / R / DE / W	2	3	T032	60.16	65.06	n.a.	n.a.
	3		T033	49.16	53.17	58.14	n.a.
	4		T034	52.49	56.76	62.07	65.06
F / RD / E / W	2	3	T032_v2	67.72	73.24	n.a.	n.a.
	3		T033_v2	57.92	62.64	68.49	n.a.
	4		T034_v2	61.05	66.02	72.2	75.68

Table.4.4 Average Energy Efficiency [nj/instr] (N.A. = NOT APPLICABLE)

Architecture	TPS	TPB	Codename	Number of Active threads NT			
				NT=1	NT=2	NT=3	NT=4
F / RDEW	2	1	T012	1.63	1.43	n.a.	n.a.
	3		T013	1.7	1.49	1.49	n.a.
	4		T014	1.92	1.68	1.68	1.68
F / R / DEW	2	2	T022	1.75	1.3	n.a.	n.a.
	3		T023	1.79	1.33	1.17	n.a.
	4		T024	1.71	1.27	1.12	1.12
F / RD / EW	2	2	T022_v2	1.74	1.3	n.a.	n.a.

	3		T023_v2	1.75	1.3	1.15	n.a.
	4		T024_v2	1.62	1.2	1.05	1.05
F / R / DE / W	2	3	T032	2.5	1.77	n.a.	n.a.
	3		T033	2.37	1.66	1.24	n.a.
	4		T034	2.36	1.67	1.24	1.1
F / RD / E / W	2	3	T032_v2	2.29	1.62	n.a.	n.a.
	3		T033_v2	2.18	1.54	1.15	n.a.
	4		T034_v2	2.26	1.6	1.2	1.06

It is obvious from table 4.3 that implementations with a smaller NT consume less dynamic power than implementations with bugger NT. However, that does not mean they are more energy efficient, since within the same implementation, the tests that were utilizing achieved the highest throughput as shown previously from table 4.2. This is evident, were the implementation running at higher *NT*, have the highest energy efficiency. Also, take note that pipelining boosted the top frequency of the core such that the throughput increase was larger than the dynamic power consumption increase, thus we can say, and as seen from table 4.4 that the pipelined architectures were not only faster, but also more energy efficient than their non-pipelined counterparts.

The reported results in the preceding tables show that the most energy efficient implementations were the T024_v2, and the T034_v2. That is due to the T024_v2 having a very low dynamic power consumption, and the T034_v2 having the highest throughput. However, our choice as the optimal IMT implementation to be used in our research was the T033_v2, which is slightly less energy efficient than T034_v2. One might argue why was our choice not following directly the results in the tables. That is because of the following reasons:

a) As we suggested at the beginning of this chapter, the recommended number of TPS in an IMT architecture should be set equal to the TPB. So, the best choice in each pipeline organization should be as follows, T011, T022, and T033.

b) Fetch buffers were present in the reported results in order to demonstrate the impact of pipeline flushing on the performance and energy efficiency. They will be removed in the chosen T033_v2 implementations. In the upgraded implementations of the T033_v2, the fetched instruction will directly go to the decode stage, and no flushing will be needed.

c) Removing the fetch from the T033_v2 will increase its throughput to match that of the T034_v2, thus making the T033_v2 to have the highest energy efficiency.

d) T033_v2 is a better choice than T034_v2 in parallel applications, since thread synchronization overheads will be smaller in the T033_v2.

e) The bigger area increase in the T034_v2 over the T033_v2 tell us that if the two implementations will be attaining the same throughput at best, then the area increase in the former does not justify its usage as an efficient processor over the latter.

f) Finally, although not very evident in the pipelined organizations, but the cycle time actually does slightly increase due to interleaving more harts.

For all the reasons above, they justify that the best option is to use the most pipelined version in which TPS is set equal to TPB (TPS=TPB). Having chosen the T033_v2 as our ideal implementation for a fast, and energy efficient processor, in the next section we will see why deeper pipelines like T04 and T05 were not explored.

4.5. Deeper pipeline organizations

4.5.1. Pipelines stages after registerfile read access:

Following the trend from the above tables, it was evident that deeper pipelines provided higher operating frequencies for the core, and interleaving sufficient threads utilized the wasted energy in the core by allowing another hart to execute instead of having a delay slot. Figure 4.2, shows the datapath of the T0 in two different pipeline organizations. The first having the memory accessed from the execute stage, the second included the memory access from a dedicated memory stage where the memory address was calculated in the previous pipeline stage.

Although deeper pipelines yielded better results as shown from the previous section. One evident problem was saturation in the cycle time decrease as the pipelining got deeper, and implementations such as T044 from figure 4.2b, might not really have higher operating frequencies, since the area overhead of supplementing additional threads will start to decrease the top frequency by increasing the net delay more than the increase in the top frequency gained by decreasing the logic delay.

Also, there will be a definite bigger overhead of stalls when synchronizing the hardware threads, or when there are idle harts in the more pipelined implementations (T044). For example, a program running on a single hart in the T033 will execute one instruction on the first hart followed by **two** wait-for-interrupt (WFI) instructions that act like a NOP. While running an application with a single hart on the T044 will execute one instruction on the first hart followed by **three** WFI instructions. This additional augmented overhead will make deeper pipeline implementations perform worse on single threaded sequential applications, and unbalanced parallel applications. While for balanced parallel applications they will maybe not perform much better due to the saturation in the top frequency increase due to pipelining.

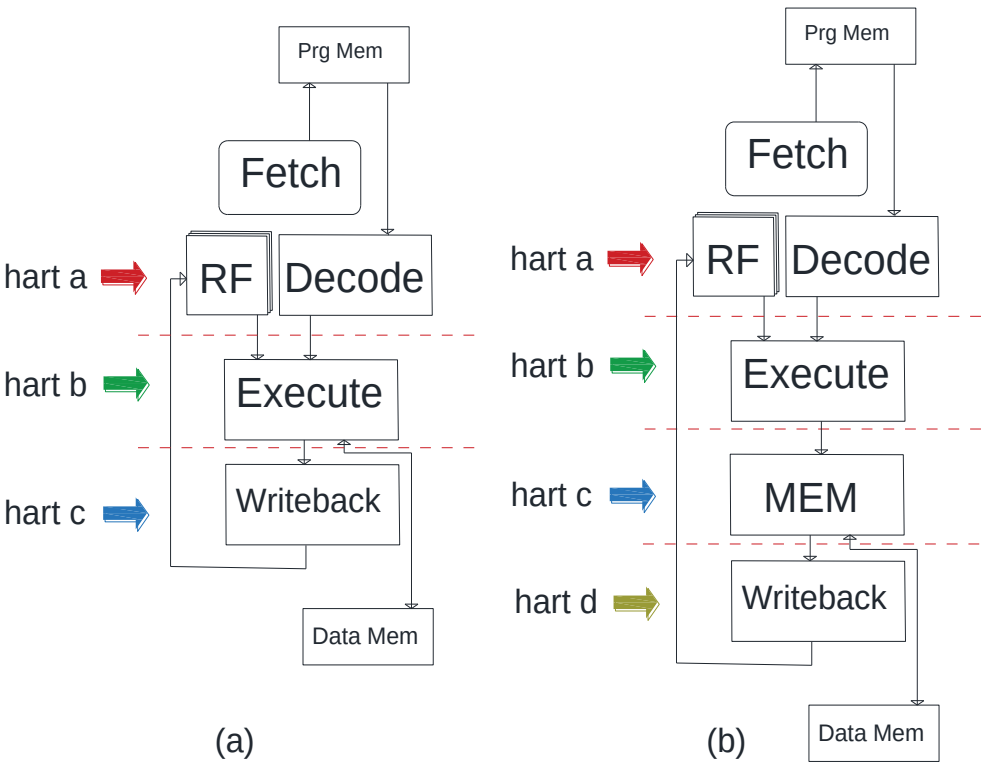


Figure.4.2. (a) Klessydra T033 datapath, three harts interleave from RF to WB,

(b) T044 datapath interleaves four harts between RF and WB

So, in-order to have a balanced IMT architecture that is fast enough and does not burden the other harts with a big overhead, T033 remained as our best choice, and post registerfile stage pipelining was ignored.

4.5.2. Pipelines stages before registerfile read access:

However, there are pipeline implementations that can be made before the registerfile read access, that do not require the IMT to increase the thread pool baseline as shown in figure 4.3. That is because the registerfile read and write accesses will still be fenced by the interleaving harts. The first is separating the decode and the registerfile into separate stages, by placing the decode before the regfile access as seen in figure 4.3a. The second can be to install a pre-fetch buffer as seen from figure 4.3b.

T033 pipeline was written such that the registerfile access is completely independent of the decode access, meaning that both entities will work in parallel. Hence separating the decode and the registerfile stages does not give any performance boosts. Second of all, introducing pre-fetch buffers will increase the number of instructions per hart in the core such that each hart will have two instructions in the pipeline, and any branch taken requires the implementation of flushing logic in order to flush the instruction of the same hart that is present in the pre-fetch buffer. Re-introducing flushing is completely avoidable, and as demonstrated from the previous section that it has a significant impact on the throughput of the core thus making it unfavorable as well.

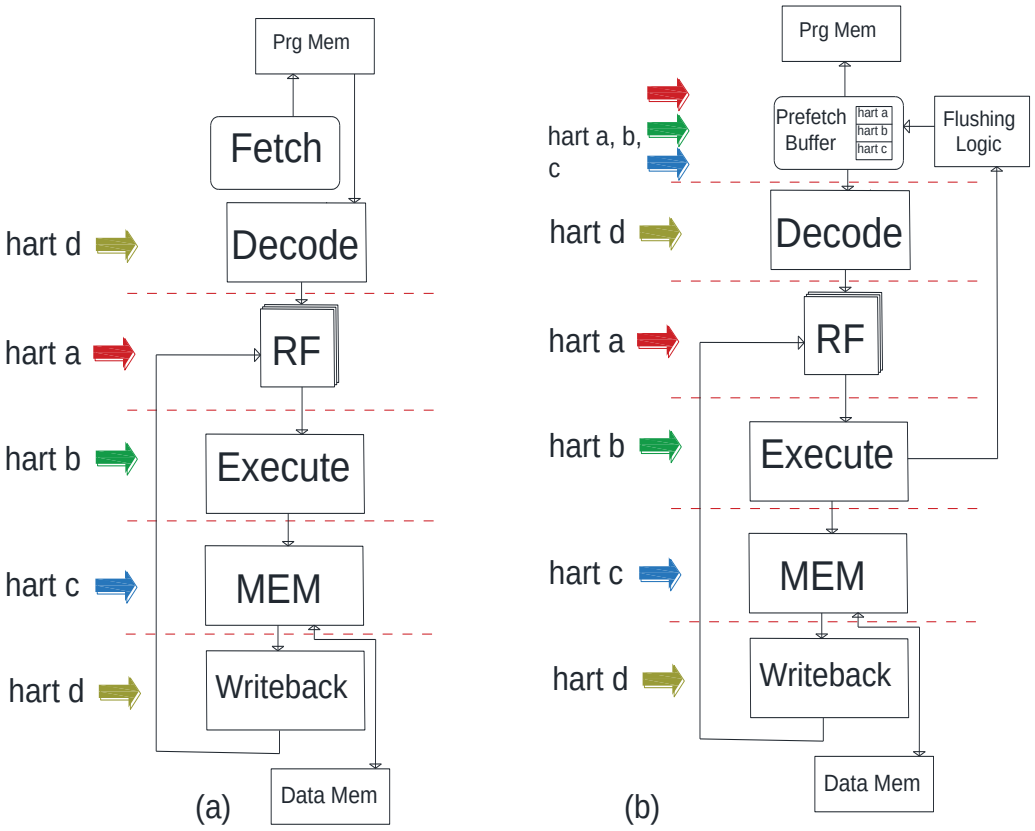


Figure.4.3. (a) Klessydra T044 datapath five pipeline stage but still works by interleaving only four harts (b) Klessydra T044 eight pipeline stage still interleaves four harts, and needs flushing logic for branch miss prediction

For the reasons mentioned earlier, pre-registerfile pipelining is avoided as well since it is either unnecessary or affects the processor's throughput by introducing branch delay slots, so we stick again with the T033 implementation.

4.5.3. Conclusion:

Choosing to maintain T033 as the optimal version of the core. In the remaining part of this chapter we will elaborate more about the building blocks of the T033, and the software developed to facilitate it. Also, one final note; from here on out, any references to T033_v2 and T022_v2 will be made as 'T03' and 'T02' respectively since our aim from the beginning was to use IMT cores to have a TPS equal to the TPB (TPS=TPB).

4.6. The T03 core

In figure 4.4, we show the basic block organization of the T03 core. It is a balanced [23] four-pipeline stage in order interleaved multithreaded processor. The pipeline stages are Decode/Regfile, Execute, and Writeback. The Fetch stage does not have any buffers to hold the incoming instructions, hence incoming instructions directly pass to the Decode stage, but since the fetch has a one cycle latency, then the fetching is still considered a pipeline stage. And the Registerfile is read in the first stage and written back in the last stage.

Registerfile reading and instruction decoding is insured to be done in parallel, and all dependencies between the two processes are eliminated. Since a dependency between instruction decoding and regfile reading will result in a high logic path delay in that pipeline stage, making the critical path to become present in that particular pipeline.

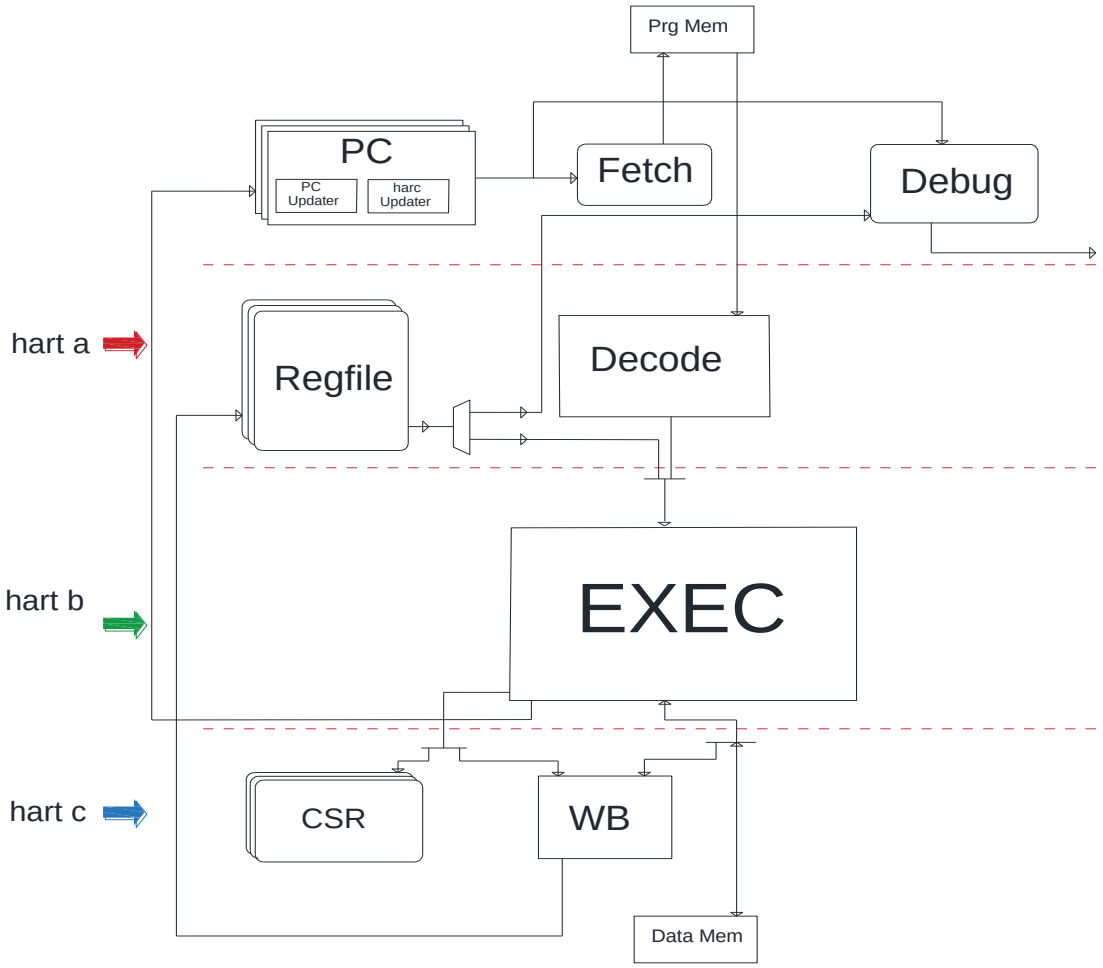


Figure.4.4. Klessydra T033 block organization, interleaves three harts in the instruction pipeline

The principle of operation of each module in figure 4.4 will be illustrated over the next the few pages. First, we will start with the datapath in the instruction pipeline, and then moving on to the remaining modules in the core.

- **Fetch:** the fetch unit is a simple finite state machine that sends a fetch request packet containing the program counter of the current active hart whenever the pipeline is not busy. The received instruction is sent to the decode stage. The RTL description of such a process is shown in the following code.

```
1  fsm_IF_nextstate : process(all)  -- acts as the control unit of the synchronous program memory
2  begin
3    if busy_ID = '0' then  – checks for a stall from the decode stage
4      instr_req_o <= '1';  -- request next instruction
5    else
6      instr_req_o <= '0'; -- stall the instruction requests
7    end if;
8  end process;
9
10 process(clk_i, rst_ni)
11 begin
12   if rising_edge(clk_i) then
13     if instr_gnt_i = '1' then – grant from the program memory
14       -- pc propagation
15       pc_ID   <= pc_IF; -- push the program counter of the incoming instruction to the decode stage
16       -- harc propagation
17       harc_ID <= harc_IF; -- push the hart identifier to the decode stage
18     end if;
19   end if;
20 end process;
```

- **Decoder:** Fetched instructions directly go into the decoder. The time to fully decode an instruction is one clock cycle only. The decoded instruction can be issued to the IE_unit or Instruction Execute unit, in which all the instructions are executed. Both execution units receive the type of the decoded instruction in a form of *one hot decoding* in which the instruction to be executed corresponds to one bit only of the entire bit-vector. This decoding scheme passed to the execute stage might generate big vectors as the instruction set supported grows larger, however, it will relieve the execution stage by making it perform the simplest re-decoding of the instruction, and limit its parts to only do the execution. The pipeline is halted whenever the decoder receives a *busy_IE* signal, from execution stage for each instruction that requires more than one cycle to execute. The pseudo code below shows the one hot decoding of the RISC-V instructions, and demonstrates how this one hot pattern are encoded in the decode stage to be passed to the IE-stage.

```
1  -- EXEC UNIT INSTR SET ----------
2  constant ADDI_pattern    std_logic_vector(INSTR_SET_SIZE-1 downto 0) := "00000000000000000000000000000000000000000000000000000000000001";
3  constant SLTI_pattern    std_logic_vector(INSTR_SET_SIZE-1 downto 0) := "00000000000000000000000000000000000000000000000000000000000010";
4  constant SLTIU_pattern   std_logic_vector(INSTR_SET_SIZE-1 downto 0) := "00000000000000000000000000000000000000000000000000000000000100";
```

```
1  fsm_ID_sync : process(clk_i, rst_ni, instr_word_ID_lat)  -- synch single state process
2  begin
3    if rst_ni = '0' then
4      ……..
5    elsif rising_edge(clk_i) then
6      if busy_IE = '1'  then   --  halt the decodeif the IE-unit is busy
7        …….
```

```
      elsif instr_rvalid_ID = '0' then  -- halt if there is no incoming valid instruction
        …….
      else      -- else decode the incoming instruction
       -- Decode OF INSTRUCTION (BEGIN) ----------------------

       ie_instr_req <= '1';   -- enable the IE stage
       case OPCODE_wires is
        when OP_IMM =>
         if(rd(instr_word_ID_lat) /= 0) then  -- instructions referencing rd=x0 instructions are executed as NOPs
          case FUNCT3_wires is
           when ADDI =>           -- ADDI instruction
            decoded_instruction_IE <= ADDI_pattern;  -- assign the correct one hot pattern to the ADDI instruction
           when SLTI =>           -- SLTI instruction
            decoded_instruction_IE <= SLTI_pattern;  -- assign the correct one hot pattern to the SLTI instruction
            …
            …
       when LUI =>               -- LUI instruction
          if (rd(instr_word_ID_lat) /= 0) then
           decoded_instruction_IE <= LUI_pattern; -- assign the correct one hot pattern to the LUI instruction
          else                    -- R0_INSTRUCTION
           decoded_instruction_IE <= NOP_pattern; -- assign the NOP pattern to the LUI instruction
          end if;
        when AUIPC =>      -- AUIPC Instruction
          …
          …
        when others =>       -- ILLEGAL_INSTRUCTION
          decoded_instruction_IE <= ILL_pattern;  -- assign illegal pattern to instructions with unrecognized opcode
       end case;  -- OPCODE_wires cases

       -- Decode OF INSTRUCTION (END) --------------------------
      end if;  -- instr. conditions
     end if;  -- clk
    end process;
```

- **Registerfile:** The T03 has a 2Rd/1Wr operand registerfile with register '*x0'* being statically bounded to 0. The registerfile can be configured to be 32x32 regfile following the RV32I instruction set, or it can be configured to be a 16x32 registerfile thus following the RV32E extension. While the instructions get decoded, it's operands are read in parallel by the registerfile.

- **Comparators**: are used to make branch decisions. Three comparators are needed to determine whether the operands satisfy a BEQ, BNE, BLT, BLTU, BGE, BGEU. The comparators will send a signal to the execute stage to indicate whether the branches will be taken or not. The separation of the comparators from the execute stage was in order to balance the decode and the execute stages.

```
-- COMPARATORS ------------------------------------------------------------------------------
      if (signed(regfile(harc_ID)(rs1(instr_word_ID_lat))(31 downto 0)) =
signed(regfile(harc_ID)(rs2(instr_word_ID_lat))(31 downto 0))) then
        pass_BEQ_ID <= '1';
      else
        pass_BNE_ID <= '1';
      end if;
      if (signed(regfile(harc_ID)(rs1(instr_word_ID_lat))(31 downto 0)) <
signed(regfile(harc_ID)(rs2(instr_word_ID_lat))(31 downto 0))) then
        pass_BLT_ID <= '1';
      else
        pass_BGE_ID <= '1';
```

```
      end if;
      if (unsigned(regfile(harc_ID)(rs1(instr_word_ID_lat))(31 downto 0)) <
unsigned(regfile(harc_ID)(rs2(instr_word_ID_lat))(31 downto 0))) then
        pass_BLTU_ID <= '1';
      else
        pass_BGEU_ID <= '1';
      end if;
-------------------------------------------------------------------------------------------------------
```

- **Execute**: The execute has a four state fsm machine:
  - **Reset State:** initial state before the core begins executing instructions.
  - **Sleep State:** Idle state in which the core waits for a *fetch_en_i* signal or an interrupt.
  - **Debug State:** Indicates that the core is currently in debug mode.
  - **Data Valid Waiting State:** Core is waiting for data to be loaded or stored into the mem**.**
  - **CSR Instruction Wait State:** Indicates that the core is handling CSR instructions.

  The execute stage encapsulates all the functional units required to execute the RISC-V instructions. The functional units are shared by the instructions, and a mapper is included in order to correctly map the instruction operands to their corresponding FUs:

  - ADDI, ADD, SUB, AUIPC, JAL, and JALR share the same adder.
  - SLLI, and SLL instructions share the same left shifter.
  - SRLI, SRAI, SRL, SRA share the right shifter.
  - AND, ANDI, OR, ORI, XOR, XORI each share their corresponding logical units.
  - MUL. MULH, MULHU, MULHSU share the same multiplier.
  - DIV, DIVU, REM, REMU share the same divider.
  - LOAD, STORE, instructions have their own adder for address creation.

  Branch instructions update the program counter of the corresponding hart if the branch is taken. T03 implementations of the core do not need any flushing logic, since each hart is only one instruction in the pipeline at a time. The execute stage also handles CSR instructions, it puts the registerfile data on the CSR write bus and the CSR data on the read bus.

  In addition, pending interrupts are served in the IE stage, more details about interrupt handling will be elaborated on later in this chapter.

```
-------------------------------------------------------------------------------------------------------
 fsm_IE_sync : process(clk_i, rst_ni)

  -- pragma translate_off
  variable row : line;  -- local variable for instruction tracing, not synthesizable
  -- pragma translate_on

 begin
  if rst_ni = '0' then
    …
  elsif rising_edge(clk_i) then
   case state_IE is                -- stage state
    when normal =>
     -- check if there is a valid instruction and the thread it belongs to is not in a delay slot:
     if instr_rvalid_IE = '0' then
       instr_rvalid_WB <= '0'; -- do nothing and wait for valid instruction and finished delay slot
     elsif irq_pending(harc_IE) = '1' then
       instr_rvalid_WB <= '0'; -- in the sync process we don't need to do anything here
```

```
          else -- process the instruction
           -- EXECUTE OF INSTRUCTION (BEGIN) ------------------------------------------

           if decoded_instruction_IE(ADDI_bit_position)  = '1' or
             decoded_instruction_IE(ADD7_bit_position)  = '1' or
             decoded_instruction_IE(SUB7_bit_position)  = '1' or
             decoded_instruction_IE(AUIPC_bit_position) = '1' or
             decoded_instruction_IE(JAL_bit_position)   = '1' or
             decoded_instruction_IE(JALR_bit_position)  = '1' then
            if (rd(instr_word_IE) /= 0) then  -- condition for JAL and JALR ops which execute when "rd = x0"
              IE_WB_EN <= '1';
            end if;
            IE_WB <= std_logic_vector(signed(add_op_A)+signed(add_op_B));  -- ADDER
           end if;

           if decoded_instruction_IE(SLLI_bit_position) = '1' or
             decoded_instruction_IE(SLLL_bit_position) = '1' then
            WB_EN <= '1';
            WB <= to_stdlogicvector(to_bitvector(sl_op_A) sll to_integer(unsigned(sl_op_B)));-- LEFT SHIFTER
           end if;
           …
           …
           if decoded_instruction_IE(SW_MIP_bit_position) = '1' then
            if sw_mip = '1' and halt_IE = '0' then
              core_busy_IE_wires := '1';  -- halt the core since the instruction takes more than one cycle
              nextstate_IE_wires := csr_instr_wait_state;-- software ints write to the MIP registers of the target
hart
            end if;
           end if;
           …
           …
           -- EXECUTE OF INSTRUCTION (END) ----------------------------------------------------
         end if;  -- instr_rvalid_IE values
        when csr_instr_wait_state =>
          …
        when others =>
          …
      end case;  -- fsm_IE state cases
    end if;  -- refers to reset signal
  end process;
;-------------------------------------------------------------------end of IE stage -----------------
-------------------------------------------------------------------------------------------------------
```

- **Writeback**: The writeback writes the result from the IE stage back to the registerfile when it receives a "WB_EN" from the IE stage. Since all the execution units are encapsulated in one entity, we will get up to one result per cycle only. Certainly, a hart can only write to its own regfile, so each registerfile needs only one write port since only one result will be ready at a time.

- **Program Counter:** A *pc_updater* fsm updates the program counter of each hart, to fetch the next instruction by incrementing the current *pc* address. A program counter may be updated by events coming from various signals:

    - ***set_branch_condition***: event happens in case of unconditional jumps or taken branches.
    - ***set_except_condition***: event happens due to executing an illegal instruction, or misaligned memory access or due to executing an Environment Call ECALL instruction, and the program counter will be updated to jump to the exception handling routine.

- ***irq_pending***: event occurs due to incoming external or timer interrupts, or inter-thread software interrupts. the program counter will be updated to jump to the interrupt handling routine.
- ***rst_ni***: event occurs only once at the startup time of execution, and updates the program counter with the boot_pc that contains the boot pointer.

The program counter has the hart interleaving unit also known as the hardware context counter (harc). The harc updates the program counters of each hart in an interleaved fashion.

- **CSR Unit:** The control and status register unit handle the execution of the CSR instructions, the automatic update of some registers due to certain events such as exceptions or interrupts (also maps the inter-thread software interrupts to the appropriate CSR unit), and handles the MRET instructions. A subset of the CSR registers is supported in Klessydra and they are listed in table 4,5 Each CSR unit has a unique identification number in the read only MHARTID register. More details about the implementation of the CSR registers can be found in appendix A.

Table.4.4. Control and status registers supported by Klessydra cores

Name	R/W	Description
MSTATUS	R/W	status register
MEPC	R/W	exception program counter
MCAUSE	R/W	trap cause
PCER	R/W	performance counter enabler
MESTATUS	R/W	exception status register backup
MHPMCOUNTER	R/W	performance-monitoring counter
MHPMEVENT	R/W	performance-event selector
MCPUID	R	cpu description
MIMPID	R	implementation description
MHARTID	R	hardware thread integer id
MIP	R/W	interrupt pending type
MTVEC	R/W	trap-handler base address
MIRQ	R	ext. interrupt request number
MBADADDR	R/W	misaligned address value

- **Debug Unit:** The core also augments a basic debug unit which can halt the execution through a debug request or an EBREAK instruction. In debug mode the core can be in two states *halt state* in which the cores halts execution after the last fetched instruction, and *single step mode* in which the core steps through every instruction in the core. In debug mode, the debug unit can read the registerfile contents of the hart in the execute stage, to read the contents of the other harts, the debug unit must single step through the instructions until the desired hart arrives to the execute stage.

4.7. Trap handling

4.7.1. Trap Handling through hardware

When a trap occurs, the IE stage automatically sends a signal to the program counter so that it updates the *pc* value of the hart in the IE stage to jump to the machine trap vector address MTVEC. The CSR unit updates the corresponding CSR registers:

- **MCAUSE** is updated with the type of exception if the trap was due to an exception.
- **MIP** is updated with the type of interrupt if the trap cause was an interrupt request.
- **MEPC** is updated with the *pc* value of the executing instruction when the trap occurred.
- **MSTATUS** indicates trap handling in progress, and disables nested traps handling.
- **MESTATUS** is backed with the pre-trap MSTATUS value.

- **MBADADDR** holds the misaligned address if the trap was due to a misaligned access.

Interrupts: Klessydra cores support three types of interrupts, *external, timer,* and *software interrupts*. Hart 0 handles timer and external interrupts, and is done as shown from the code below.

```
-- synchronous assignment to MIP_internal bits:
-- this is Pulpino-specific assignment, i.e. the timer-related IRQ vector value
– the h index refers to the hart, in this case hart 0 only enters the condition
if h = 0 and unsigned(irq_id_i) >= 28 and irq_i = '1' then  -- the irq is a timer interrupt
  MIP_internal(h)(7) <= '1';
else
  MIP(h)(7) <= '0';
end if;
-- this detects the other IRQ vector values in Pulpino
if h = 0 and unsigned(irq_id_i) < 28 and irq_i = '1' then  -- the irq is an external interrupt
  MIP(h)(11) <= '1';
else
  MIP(h)(11) <= '0';
end if;     -- the MIP(h)(3), software interrupt bit is handled by all the harts
```

All the harts on the other hand can send and receive software interrupts through a store word instruction to a specific address in the memory map. The address tag (upper bits of the SW address) is checked in the ID stage, and if the tag maps to the software interrupt's address tag, the store instruction will instead act as a CSR instruction that writes to the MIP register of the other harts as shown in the VHDL code below.

```
if decoded_instruction_IE(SW_MIP_bit_position) = '1' then – a store word that writes to the MIP of a hart
  if sw_mip = '1' then  -- the upper bits of  the address are decoded in the ID stage to know if the SW is a SW_MIP
    csr_op_i          <= CSRRW;  -- set the type of CSR instruction
    csr_instr_req     <= '1';   -- enable the CSR unit
    ie_csr_wdata_i <= RS2_Data_IE; -- put the data on the CSR bus
    csr_wdata_en   <= '1';  -- enable csr write
    csr_addr_i        <= MIP_ADDR;  -- the csr address is the MIP register
     -- the lower address bits of the SW instruction are decoded to know which hart receives the software interrupt
    for i in harc_range loop
      if data_addr_internal_IE(3 downto 0) = std_logic_vector(to_unsigned((4*i),4)) then
            harc_to_csr <= i;  -- harc_to_csr enables the target CSR unit
      end if;
    end loop;
  end if;
end if;
```

When a hart receives an interrupt of any type, it will be directly serviced as soon as the hart arrives at the IE stage in the pipeline, and the instruction that is currently in the IE stage will not be executed. The hart will jump to the interrupt servicing routine, and will return at the end of the routine with an *MRET* instruction to the same address in order to execute the instruction that was discarded before. If the instruction discarded happened to be a *WFI*, this case will be registered when the trap occurs in the MSTATUS(h)(30) register of the hart indexed in *h,* and the return from the interrupt routine during the MRET execution will be to the "WFI_ptr + 4" instead. This is essential in order to break the core from being stuck in an infinite loop. The following code briefly shows how the CSR units updates the CSR registers for each type of event Interrupt/exception and how the MSTATUS recovers after servicing the interrupt routine.

```
-- Interrupt-cause CSR updating ---------------------------------
-- note: PC just udpdated, MIP_internals can't have been cleared yet.
if served_irq(h) = '1' and MIP_internal(h)(11) = '1' then
  -- it is the MEIP bit, ext. irq
```

```
      MCAUSE_internal(h) <= "1" & std_logic_vector(to_unsigned(11, 31));  -- ext. irq
      MESTATUS(h)(2 downto 1)    <= MSTATUS_internal(h); -- push the MSTATUS to back MESTATUS register
      if WFI_Instr = '1' then   -- Indicates to the MEPC that the return address contains a WFI instruction
        MCAUSE_internal(h)(30) <= '1';
      else
        MCAUSE_internal(h)(30) <= '0';
      end if;
      MSTATUS_internal(h)(0) <= '0';  -- interrupt handling temporarily disabled,
      MSTATUS_internal(h)(1) <= MSTATUS_internal(h)(0);  -- trap handling pending in progress
    elsif served_irq(h) = '1' and MIP_internal(h)(3) = '1' then
      -- it is the MSIP bit, sw interrupt req
      MCAUSE_internal(h) <= "1" & std_logic_vector(to_unsigned(3, 31));  -- sw interrupt
      MIP_internal(h)(3) <= '0'; -- we reset the sw int. request just being served
      …  -- similar assignments as the ext irq
      …
    elsif served_irq(h) = '1' and MIP_internal(h)(7) = '1' then
      -- it is the MTIP bit, timer interrupt req
      MCAUSE_internal(h) <= "1" & std_logic_vector(to_unsigned(7, 31));  -- timer interrupt
      …-- similar assignments as the ext irq
      …
    --  Exception-cause CSR updating ---------------------------------
    elsif served_except_condition(h) = '1' then
      if served_ie_except_condition(h) = '1' then
        MCAUSE_internal(h)     <= ie_except_data;  -- exception cause passed from IE Stage
      end if;
      MESTATUS(h)(2 downto 1)  <= MSTATUS_internal(h); -- push the MSTATUS to backup register MESTATUS
      MEPC_internal(h)   <= pc_except_value_wire(h);
      MSTATUS_internal(h)(0) <= '0';  -- interrupt handling temporarily disabled,
      MSTATUS_internal(h)(1) <= '1';  -- trap handling pending in progress
      if misaligned_err = '1' then
        MBADADDR(h) <= data_addr_internal; -- store the misaligned address that caused the trap
      end if;

    -- MRET-cause CSR updating ----------------------------------------
    elsif served_mret_condition(h) = '1' then
      MSTATUS_internal(h)(1) <= '1';  -- re-enable the trap handling
      MSTATUS_internal(h)(0) <= MSTATUS_internal(h)(1); -- indicate the core is no longer handling traps
    end if;
```

4.7.2. Trap handling through software

In the startup code there is a an MTVEC label indicating the start of the routine to execute during a trap. The routine will simply compare the MCASUE value to the table of trap handlers to know which trap handling to execute, and once the MCAUSE matches the value in the trap table, it will jump to the trap handling routine defined by PULPino, and then returns back to the execution environment. Below is a partial assembly snippet of the trap handling routine from the *klessydra_startup.S* file.

```
mtvec_routine:
        addi      sp,sp,-KLESSYDRA_EXC_STACK_SIZE;  // decrement the stack pointer
        sw        t4,0x00(sp);  // save the register to be modified on the stack
        sw        t5,0x04(sp);
        sw        t6,0x08(sp);
        csrrs t5, k_mcause, x0;  // load the casue of the trap
        csrr t4, k_mirq;  // load the the interrupt id
        li t6, EXT_INTERRUPT_CODE;
        bne t5, t6, no_ext_interrupt;  // Check whether the trap was due to an external interrupt
        lw t5, 0x04(sp);
        lw t6, 0x08(sp);
```

```
        jr t4;

no_ext_interrupt:
        li t6, SW_INTERRUPT_CODE_WFI;   //In klessydra, if we have a WFI, we write a "1" to the bit mcause(30),
to return to the instruction following the WFI
        beq t5, t6, sofware_insn_handler;
        li t6, SW_INTERRUPT_CODE_NO_WFI; // Check whether the trap was due to a software interrupt
        beq t5, t6, sofware_insn_handler; // They jump to the same routine the since mepc is incrememnted in hardware,
when the mepc return value is a WFI instruction
        li t6, TIMER_INTERRUPT_CODE; // Check whether the trap was due to a timer interrupt
        bne t5, t6, exception_trap;
        lw t5, 0x04(sp);
        lw t6, 0x08(sp);
        jr t4;

exception_trap:
        li t6, ECALL_EXCEPT_CODE;  // Check whether the trap was due to an ECALL instruction
        beq t5, t6, ecall_insn_handler;
        li t6, ILLEGAL_INSN_EXCEPT_CODE;   // Check whether the trap was due to executing an illegal instruction
        beq t5, t6, illegal_insn_handler;
        li t6, LOAD_ERROR_EXCEPT_CODE;  // Check whether the trap was due to a load error
        beq t5, t6, invalid_addr_handler;
        li t6, STORE_ERROR_EXCEPT_CODE;  // Check whether the trap was due to a store error
        beq t5, t6, invalid_addr_handler;
        li t6,LOAD_MISALIGNED_EXCEPT_CODE;  // Check whether the trap was due to a misaligned access
        beq t5, t6, invalid_addr_handler;

        lw t4,0x00(sp);  // recover the stack
        lw t5, 0x04(sp);
        lw t6, 0x08(sp);
        addi      sp,sp, KLESSYDRA_EXC_STACK_SIZE;  // recover the stack pointer
        mret;  // return to the execution environment
```

Klessydra specific C functions that have been integrated to the libraries inside Pulpino to be used to quickly send software interrupts. The following is the body of the C function that sends a software interrupt to a target hart. The function takes one argument which is the *hart_id.* From the *hart_id* it will generate the MIP address and send a store word to that MIP value.

```
int send_sw_irq(int targethart){
        int mip_data_send = 8;
        int store_addr = 0xff00;  // Base address of the software interrupt memory section
        if(targethart >= THREAD_POOL_SIZE) return 0;  // the thread in which the interrupt was sent doesn't exist
        else {    store_addr = store_addr + (4*targethart);     // MIP address generation
                  store_mem(mip_data_send, store_addr);     // Send a store word with address with the MIP address
                  return 1;}}

void store_mem(int data_send, int store_addr) {
        __asm__("sw %0, (%1);"
                  :/*no output register*/
                  :"r"(data_send), "r"(store_addr)
                  :/*no clobbered register*/);}
```

4.8. Thread synchronization.

4.7.3. Atomic Instruction Support:

The atomic extensions were augmented to the instruction set supported by Klessydra-T cores in order to support thread synchronization of the harts. However, only a minimal integration of the atomic extension was done such that the only atomic instruction implemented was the '*amoswap'*. Implementing the *amoswap* instruction is sufficient enough in order to have thread synchronization, and implement region locks (acquire, and release) on a memory location. Briefly an *amoswap* instruction loads a key value from a memory and swaps the loaded value with a lock. In order for the *amoswap* to work correctly, the pointers of the instruction must be addressing the regions in the shared *.data* section of the data memory by assigning them as global variables, and not the dedicated *.stack* section, since each hart has its own dedicated stack region in the memory. The following are the body of the functions which do lock acquire, and lock release to memory regions. Both functions take an argument which is a pointer to the lock that is a global variable.

```
1  void klessydra_lock_acquire(int *lock){
2          int temp0 = 1;
3          __asm__(
4                  "loop: "
5                  "amoswap.w.aq %1, %1, (%0);" // Set the lock by swapping the key '0' with '1'.
6                  "bnez %1,loop;"  // loop until the lock is released.
7                  ://no output register
8                  :"r" (lock), "r" (temp0)
9                  :/*no clobbered registers*/);}
10
11 void klessydra_lock_release(int *lock)
12 {
13         __asm__(
14                 "amoswap.w.rl x0, x0, (%0);" // Release lock by storing 0.
15                 ://no output
16                 :"r" (lock)
17                 ://no clobbered register);}
```

4.7.4. Barrier Functions:

The previous functions can ensure the safe access to shared memory regions by blocking the access of all the other harts. However, in order to have thread synchronization, the Klessydra libraries include an additional set of *sync_barrier* functions to synchronize the threads.

- ***sync_barrier_reset*** is used once at the beginning of the code and when the harts are in sync. The function does a *csrw* to the MSTATUS register in order to enable the handling of interrupts (i.e. software interrupts in our case). And initializes all the variables to be read in the following functions.

- ***sync_barrier_thread_registration*** is used when the harts are in sync, and it registers every hart that calls this function. This registration process is essential to know the total number of harts interleaving in the IMT core.

- ***sync_barrier*** function synchronizes the harts. The harts to be synchronized call the function in chronological order, all the harts except the last one that enter the function register themselves in array to indicate they arrived at the barrier. A conditional structure will compare the number of harts registered versus the number of harts that arrived at the barrier function,

and if the number of harts arrived is less than the number of harts registered, then the hart that entered the barrier function will go to a WFI state. Once the last hart enters the functions and registers itself, the if condition will check that all the harts arrived, and this hart will go to 'else' state and starts sending software interrupts to every sleeping hart in the core. The harts will return from this function synchronized. One important note about the barrier function is that the routine for the barrier-arrival-registration of the harts, and the following code to check the number of the harts arrived is done atomically. Performing this routine without atomicity might in some cases confuse the hart reading the global variables, and will thus send all the harts to a WFI state.

The *sync_barrier* function bodies are shown below.

```
1  void sync_barrier_reset(){
2          int i;
3          int key = 1;
4          static int section = 0;
5          int* ptr_section = &section;
6          asm volatile
7          (       "csrrw zero, mstatus, 8;"   // enable the interrupt handling
8                  "amoswap.w.aq %[key], %[key], (%[ptr_section]);"
9                  :[key] "r" (key), [ptr_section] "r" (ptr_section):);
10         if (section == 0){
11           for (i=0;i<THREAD_POOL_SIZE; i++)  {
12                 sync_barrier_register[i] = 0;  }}}
13
14 void sync_barrier_thread_registration(){
15   int my_hart;
16   my_hart = Klessydra_get_coreID();
17   arrived_at_barrier[my_hart] =  0;
18   sync_barrier_register[my_hart] = 1;}
19
20 void sync_barrier(){
21         int my_hart, i;
22         int *ptr_key = &key_barr;
23         my_hart = Klessydra_get_coreID();
24         if(syc_barrier_register[my_hart] == 1) {  // checks if the hart entering was registered
25               klessydra_lock_acquire(ptr_key);   // the following routine must be done atomically
26               barrier_completed[my_hart] = 1; // set to 1 to indicate that all harts arrived, else it will be set to zero
27               arrived_at_barrier[my_hart] = 1;  // notifies the core that the hart with the hart_id in "my_hart" has arrived
28               for (i=0;i<THREAD_POOL_SIZE; i++)  {
29                  if (arrived_at_barrier[i] == 0 && sync_barrier_register[i] == 1) {
30                       barrier_completed[my_hart] = 0;}} // reset to zero, since not all the harts arrived at the barrier
31               if (barrier_completed[my_hart] == 0){  // send the waiting threads to a WFI state
32                  klessydra_lock_release(ptr_key);  // release lock acquired previously
33                  __asm__("WFI;");}  // put the hart to sleep with a WFI
34                else{
35                  klessydra_lock_release(ptr_key);  // release lock acquired previously
36                  for (i=0;i<THREAD_POOL_SIZE; i++){
37                           if (my_hart != i  &&  sync_barrier_register[i] == 1) {
38                              send_sw_irq(i);}
39                              sync_barrier_register[i]=0; } // unregister all of the registered harts
40                              barrier_completed[my_hart] = 0;}}}
```

4.9. Conclusion

Throughout this chapter we studied the IMT processors, and we made an an experimental and analytical assessment in order to determine the optimal pipeline organization to be adopted. Having chosen T03 as our optimal IMT implementation, we integrated the T03 inside Pulpino, and we adjusted the support of the exceptions, and interrupts in order to be compatible with the SoC. Also, we added a set of libraries to Klessydra that can be utilized to exploit the architecture, In the next chapter we will see how we can further improve the T03 IMT core.

Chapter 5 Klessydra-T1 Architectures

5.1. Background

In the previous chapter we have shown how an IMT processor, can be easily exploited in two classes of applications. Decoupled applications, each of which runs on a dedicated hart, and balanced parallel applications that allocate equal or semi-equal workloads to every hart, and the nature of the workloads being balanced among the harts gives only a tiny overhead during thread synchronization. A good example of threads running dedicated applications is for instance when using the SoC in an environment in which each hart interfaces its own peripheral device for instance; I/O devices or sensors or wireless devices and etc. The previous study from chapter 4 showcased the performance of the T03 when executing some basic control, or integer arithmetic applications. The advantages of using the T0 cores covered only small portions of the entire spectrum of applications. However, this chapter shows that IMT cores can be utilized in broader areas, in which harts can work together to run specialized applications that are easily exploited with superscalar hardware accelerators coupled with dedicated low latency local energy efficient scratchpad memories [36][37]. In this chapter, our aim is to exploit IMT processors to perform well in a broader set of the computing application spectrum and that is through the augmentation of specialized hardware accelerators. The T03 version supporting specialized hardware acceleration is called the T13 core.

As mentioned in the previous chapter that T03 is a short hand for T033, and also in this chapter, the T13 is a short hand for T133. The T13 is part of the Klessydra open source project. [31][32][33][[34] and it expands the instruction set of T03 with two extensions; the first being the “M” (multiply/divide) extension which is handled in the IE block, and the second is the “K” custom instruction set extension, specifically designed to facilitate vector calculations, that is managed by the SPMU. So, the ISA supported by the T13 core is RV32IMAK. The T13 core was designed to allow superscalar execution, and yet still interleave only three harts in the core. The superscalar execution of the T13 is done without creating any highly multi-ported registerfiles as those available in Out-of-Order architectures. It parallelizes the execution in IMT processors while still maintaining the pipeline stages, and the thread pool baseline of the T03. It demonstrates how simple it is to augment a hardware accelerator, and shows how to design the accelerators in order exploit thread level parallelism. Different hardware accelerator schemes have been implemented in order to see which approach yields the best performance, area, and energy efficiency.

This chapter starts by demonstrating the motivation for augmenting a hardware accelerator to the T03 architecture in section 5.2. Then it would describe the microarchitecture of the augmented hardware accelerator in 5.3. Section 5.4 shows how our accelerator can be built in different implementations Then in section 5.5 a set of different hardware accelerator schemes are provided in order to study the optimal choice to use for exploiting an IMT processor. Followed by a performance benchmark of the different hardware accelerator schemes from section 5.4. In section 5.6 the FPGA synthesis results are reported when synthesizing the T13 core with the different accelerator schemes shown in section 5.3. In section 5.7, supplementary tests are made to further test the T13 hardware accelerator

5.2. Motivation for augmenting the T03 core with a hardware accelerator

The IMT core presented in this chapter is called the *Klessydra-T13* (*T13* for short). The T13 block organization is shown in figure 5.1, it maintains the same hart count of its predecessor the T03. However, unlike the T03, the T13 introduces superscalar execution giving rise to the possibility of having instructions from different harts in the execute stage as seen below.

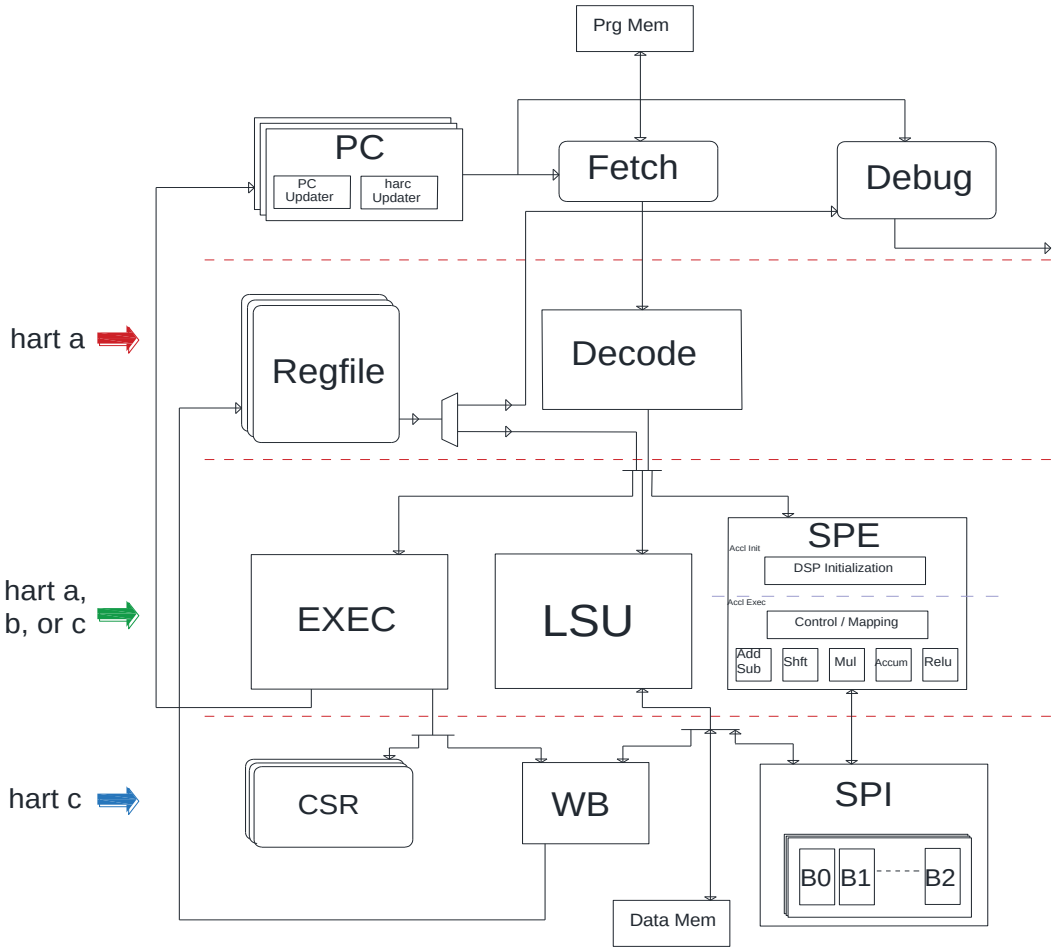


Figure.5.1. Klessydra T133 block organization, interleaves three harts and has three execution units working in parallel

A good practice to make a superscalar processor is to let each augmented execution unit write into its own memory. Take a look at figure 5.1 for example; The Load-Store Unit (LSU) only allows superscalar execution with the other units is when the instruction its handling is a store. Since stores write to the external memory, and not the registerfile. Following the same concept, we have created a hardware accelerator called the Special Purpose Mathematical Unit (SPMU) that has its own execution units and its own dedicated local Scratchpad Memories. The SPMU has its own custom instructions that can read from the SPMs or the registerfile, however, it only writes to the scratchpads and never to the registerfile. Working in this fashion, the SPMU can automatically be said to work in parallel with the other execution units, since it does not perform any concurrent writes to shared memories.

Following this practice, hardware accelerators can be easily augmented to IMT architectures, to increase their capabilities in targeting a large portion of the spectrum of computing applications.

5.3. Special Purpose Mathematical Unit Microarchitecture

The SPMU is the hardware accelerator. It was given the name “Special Purpose” because it performs a certain subset of mathematical operations specifically designed to accelerate the execution of Convolutional Neural Networking Applications (CNN). The SPMU is comprised of two main sub-systems as seen in figure 5.2. The Special Purpose Engine (SPE) which maps, controls, and executes the SPMU instructions, and the Scratchpad Memory Interface (SPI) that manages the SPE and LSU access to the scratchpad memories (not to be confused with SPI “serial peripheral interface”).

The SPMU can be compared to a vector processor rather than a packed SIMD [46][47] processor since it executes on sets of data of variable vector length, unlike SIMD instructions that have a fixed vector length. However, throughout the rest of this chapter and the next, the word "SIMD will be used to refer to the nature of the execution of the instructions and not the type of the instructions. The instructions are of type vector, and not SIMD.

In the T13, the length of the vector to execute in each instruction is set in a custom CSR called Machine Vector Size "MVSIZE". Also, similarly the SPMU compares to a vector processor by allowing the configuration of different data types, the data types supported in the SPMU are integer 8-bit, 16-bit, and 32-bit.

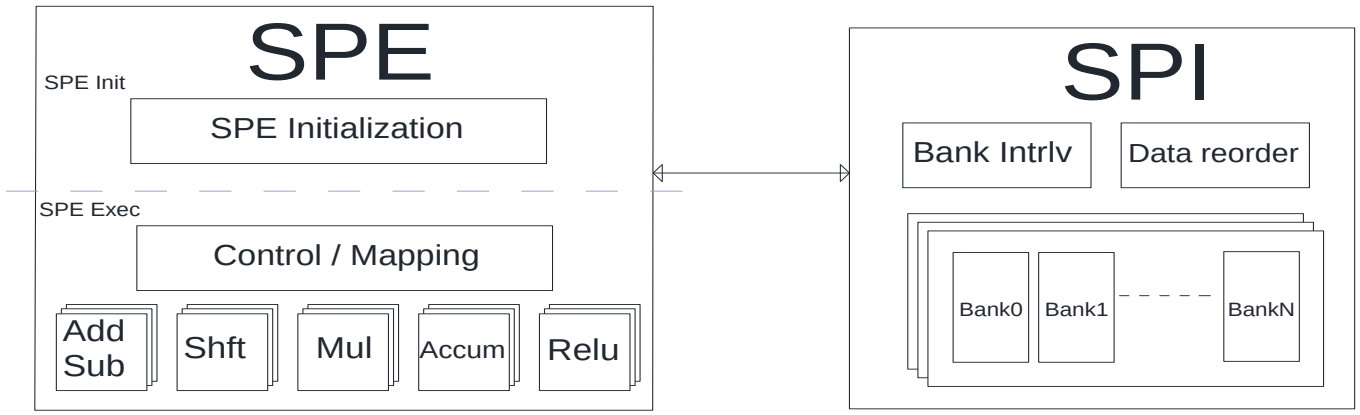


Figure.5.2. SPMU Block Diagram

5.3.1. Special Purpose Engine

The execution of the T13 custom *K* instruction set extensions is done in the SPE. The SPE is composed of many integral sub-systems, which handle the configuring, fetching, mapping, executing, and writing of the instruction. At any point in time the SPMU can be in any of the following two states:

- **SPE_INIT:** The default state of the SPE, and also the initial state for every instruction, this state handles the configuring of the functional units, and the exception control checking, fetching of the first data elements, and buffer the signals coming from the Decode, and CSR units.
- **SPE_Exec:** The SPE transfers to this state if there are no exceptions, and in the *SPE_Exec* state, we handle the hardware-loops, mapping, fetching the next elements, executing operations, and writing the results. After the results have been written successfully the SPE returns back to the SPE_INIT state.

Each of the SPMU's sub-systems will be described in the following paragraphs detailing their functions, and also showcasing VHDL snippets of how they were implemented.

The exception handler is a part of the initialization phase which checks for any current exceptions, and predicts for any future exceptions right at the very first cycle of the execution of a custom instruction from the "K" extension. All the exceptions are regarding the SPM access.

The main reason for controlling exceptions in the first cycle is that after the first cycle, the core enables the dispatch of the instructions of the other harts, and the state of the registerfile. So, in the

case of encountering an exception in the first cycle, the core will recover the state of the processor precisely to the time before the exception occurred without having the registerfile being modified. Detecting exceptions after the first cycle requires a history file to recover the processor's state precisely for when the program counter returns from the trap handling routine, which is an efficient procedure seeing that the nature of an exception happening is quite exceptional.

The following are a list of what might be exception triggers in the SPMU:

1. **Out of bound SPM access**; in this case, one of the pointers to a data element is pointing to an address not belonging to any of the SPM memories.
2. **Dual SPM read access**; a SPM has one read port, and when the two instruction operands point to the same SPM, we encounter an exception.
3. **Overflow data read and write**; this happens when the SPM pointer plus the vector size will overflow the address of the SPM being indexed. This overflow exception only traps when the operand being indexed is used as a vector, and not scalar.
4. **Misaligned access**; SPMs are 32-bit word aligned and any misaligned access will trigger this exception.

Below is the RTL description of the exception handler in the SPMU.

```
1   ---------- Exception handler of SPE Unit ----------
2   SPE_Excpt_Cntrl_Unit_comb : process(all)
3   begin
4   …
5   …
6     if spe_instr_req = '1' or busy_SPE_internal_lat = '1' then
7       case state_SPE is
8         when SPE_init =>
9           overflow_rs1_spm <= std_logic_vector('0' & unsigned(RS1_Data_IE(Addr_Width -1 downto 0)) +
10                                                    unsigned(MVSIZE(harc_EXEC)) -1);
11          overflow_rs2_spm <= std_logic_vector('0' & unsigned(RS2_Data_IE(Addr_Width -1 downto 0)) +
12                                                    unsigned(MVSIZE(harc_EXEC)) -1);
13          overflow_rd_spm  <= std_logic_vector('0' & unsigned(RD_Data_IE(Addr_Width  -1 downto 0)) +
14                                                   unsigned(MVSIZE(harc_EXEC)) -1);
15          if MVSIZE(harc_EXEC) = (0 to Addr_Width => '0') then  -- don't execute instructions with zero vector elements
16            null;
17          elsif MVSIZE(harc_EXEC)(1 downto 0) /= "00" and MVTYPE(harc_EXEC)(3 downto 2) = "10" then
18            except_condition_wires := '1'; -- Set exception if the number of bytes are not divisible by four
19            except_data_wire         <= ILLEGAL_VECTOR_SIZE_EXCEPT_CODE;
20          elsif MVSIZE(harc_EXEC)(0) /= '0' and MVTYPE(harc_EXEC)(3 downto 2) = "01" then
21            except_condition_wires := '1'; -- Set exception if the number of bytes are not divisible by two
22            except_data_wire         <= ILLEGAL_VECTOR_SIZE_EXCEPT_CODE;
23          elsif (rs1_to_spm  = "100" and vec_read_rs1_ID = '1') or
24                (rs2_to_spm  = "100" and vec_read_rs2_ID = '1') o
25                 rd_to_spm   = "100" then
26            except_condition_wires := '1'; -- Set exception for non-scratchpad access
27            except_data_wire         <= ILLEGAL_ADDRESS_EXCEPT_CODE;
28          elsif rs1_to_spm = rs2_to_spm and vec_read_rs1_ID = '1'
29                                       and vec_read_rs2_ID = '1' then
30            except_condition_wires := '1'; -- Set exception for same read access
31            except_data_wire         <= READ_SAME_SCARTCHPAD_EXCEPT_CODE;
32          elsif (overflow_rs1_spm(Addr_Width) = '1' and vec_read_rs1_ID = '1') or
33                (overflow_rs2_spm(Addr_Width) = '1' and  vec_read_rs2_ID = '1') then
34            except_condition_wires := '1';  -- Set exception if reading overflows the scratchpad's address
35            except_data_wire         <= SCRATCHPAD_OVERFLOW_EXCEPT_CODE;
36          elsif overflow_rd_spm(Addr_Width) = '1'  and vec_write_rd_ID = '1' then
37            except_condition_wires := '1'; -- Set exception if reading overflows the scratchpad's address
38            except_data_wire <= SCRATCHPAD_OVERFLOW_EXCEPT_CODE;
39          else  -- else we process the instruction
40            if halt_hart = '0' then
```

```
41              nextstate_SPE <= spe_exec;
42             else
43              nextstate_SPE <= spe_halt_hart;
44             end if;
45             busy_SPE_internal_wires := '1';
46            end if;
47           when others =>
48            null;
49           …
```

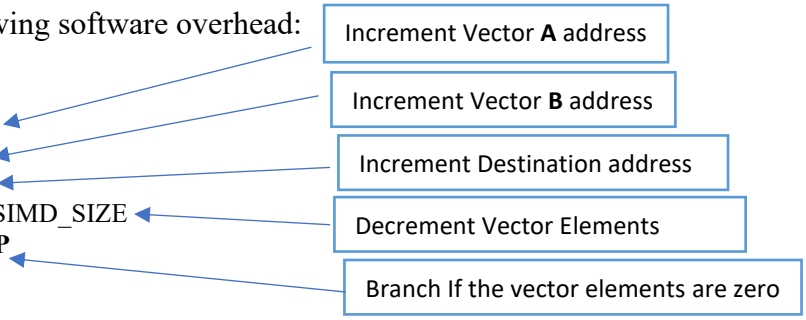
The initialization block configures the functional units correctly in order to execute the instructions in flight. An example of some configurations might be; Setting the *FU* controls to execute the data type to be computed on, such as; *chars*, *shorts* or *ints*. Other configurations might also be to transform the input operands into their two's complement or they might be to configure outputs to either become sign extended or zero extended.

```
1   -------- FU Inittialiazion phase ----------------------------------------------------------------------------------------------------
2           -- Set signals to enable correct virtual parallelism operation
3            if (decoded_instruction_SPE(KADDV_bit_position)    = '1'      or
4                decoded_instruction_SPE(KSVADDSC_bit_position) = '1') and
5              MVTYPE(3 downto 2) = "10" then
6              carry_pass <= "111";  -- pass all carry_outs
7            elsif decoded_instruction_SPE(KSVADDRF_bit_position) = '1' and
8              MVTYPE(3 downto 2) = "10" then
9              carry_pass <= "111";  -- pass all carry_outs
10             rf_rs2 <= '1';
11             …
12           elsif decoded_instruction_SPE(KSUBV_bit_position) = '1' and
13             MVTYPE(3 downto 2) = "10" then
14             carry_pass <= "111";  -- pass all carry_outs
15             twos_complement <= "00010001000100010001000100010001";
16           elsif decoded_instruction_SPE(KSUBV_bit_position) = '1' and
17             MVTYPE(3 downto 2) = "01" then
18             carry_pass <= "101";  -- pass carrries 9, and 25
19             twos_complement <= "01010101010101010101010101010101";
20           elsif decoded_instruction_SPE(KSUBV_bit_position)  = '1' and
21             MVTYPE(3 downto 2) = "00" then
22             carry_pass <= "000";  -- don't pass carry_outs and keep addition 8-bit
23             twos_complement <= "11111111111111111111111111111111";
24             …
25           elsif decoded_instruction_SPE(KDOTP_bit_position) = '1' and
26             MVTYPE(3 downto 2) = "10" then
27             FUNCT_SELECT_MASK <= (others => '1');  -- This enables 32-bit multiplication with the 16-bit multipliers
28             dotp <= '1';
29           elsif decoded_instruction_SPE(KDOTP_bit_position) = '1' and
30             MVTYPE(3 downto 2) = "01" then
31             dotp <= '1';
32             MVTYPE(3 downto 2) = "00" then
33             dotpps <= '1';
34           elsif decoded_instruction_SPE(KSVMULRF_bit_position) = '1' and
35             MVTYPE(3 downto 2) = "10" then
36             FUNCT_SELECT_MASK <= (others => '1');
37             rf_rs2 <= '1';
38           elsif (decoded_instruction_SPE(KVMUL_bit_position)       = '1'  or
39                  decoded_instruction_SPE(KSVMULSC_bit_position) = '1') and
40             MVTYPE(3 downto 2) = "10" then
41             FUNCT_SELECT_MASK <= (others => '1');
42           end if
43  ---------------------------------------------------------------------------------------------------------------------------- ------------
```

In the execute state of the SPE, the hardware-controlled loops or shortly **hardware loops** (hw-loops) eliminate the overhead required for looping operations. It increments the source operand pointers to fetch the next element of each operand only when the instruction operands are defined as vector sources and not scalar sources. The same applies for the writing of the resutls. The hw-loops also handles decrementing the vector length continuously. When the vector size becomes zero, the hw-loops stop, and the instruction is considered done. A masking vector is created depending on the number of elements left, such that if the number of elements is less than the number of bytes processed in one cycle, the mask will disable the upper bytes of the fetched elements. This is essential when elements fetched get accumulated. In this case, we need to avoid accumulating data not belonging to the instruction in order to get correct accumulation results.

A hardware loop saves the following software overhead:

- **SIMD_LOOP:**
- **VADD** *dest, *opA, *opB;
- **ADDI** *opA, *opA, SIMD_SIZE
- **ADDI** *opB, *opB, SIMD_SIZE
- **ADDI** *dest, *dest, SIMD_SIZE
- **SUB** VEC_SIZE , VEC_SIZE, SIMD_SIZE
- **BEQZ** VEC_SIZE, **SIMD_LOOP**




```
1   if halt_spe = '0' then  -- the hardware loops work only when there is no halt from the SPI
2    -- Increment the write address when we have a result as a vector
3    if vec_write_rd_lat = '1' and wb_ready = '1' then  -- destination address increment
4       RD_Data_IE_lat  <= std_logic_vector(unsigned(RD_Data_IE_lat)  + SIMD_RD_BYTES);
5    end if;
6    if wb_ready = '1' then -- decrement by SIMD_BYTE Execution Capability
7     if to_integer(unsigned(MVSIZE_WRITE)) >= SIMD_RD_BYTES then
8       MVSIZE_WRITE <= std_logic_vector(unsigned(MVSIZE_WRITE) - SIMD_RD_BYTES);
9     else  -- decrement the remaining bytes
10      MVSIZE_WRITE <= (others => '0');
11    end if;
12   end if;
13   -- Increment the read addresses
14   if to_integer(unsigned(MVSIZE_READ)) >= SIMD_RD_BYTES and data_gnt_i = '1' then
15    if vec_read_rs1_lat = '1' then -- source 1 address increment
16      RS1_Data_IE_lat <= std_logic_vector(unsigned(RS1_Data_IE_lat) + SIMD_RD_BYTES);
17    end if;
18    if vec_read_rs2_lat = '1' then  -- source 2 address increment
19      RS2_Data_IE_lat <= std_logic_vector(unsigned(RS2_Data_IE_lat) + SIMD_RD_BYTES);
20    end if;
21   end if;
22   -- Decrement the vector elements that have already been operated on
23   if data_gnt_i = '1' then -- decrement by SIMD_BYTE Execution Capability
24    if to_integer(unsigned(MVSIZE_READ)) >= SIMD_RD_BYTES then
25      MVSIZE_READ <= std_logic_vector(unsigned(MVSIZE_READ) - SIMD_RD_BYTES);
26    else -- decrement the remaining bytes
27      MVSIZE_READ <= (others => '0');
28    end if;
29   end if;
30   spm_data_read_mask <= (others => '0');
31   if data_gnt_i_lat = '1' then
32    if to_integer(unsigned(MVSIZE_READ_MASK)) >= SIMD_RD_BYTES then
33      spi_data_read_mask <= (others => '1');
34      MVSIZE_READ_MASK<=std_logic_vector(unsigned(MVSIZE_READ_MASK) -
35                                                          SIMD_RD_BYTES);
36    else
37      MVSIZE_READ_MASK <= (others => '0');
```

```
            spi_data_read_mask(to_integer(unsigned(MVSIZE_READ_MASK))*8-1 downto 0)<=(others => '1');
          end if;
        end if;
      end if;
```

The fetched input operands go into the **mapping unit**, that maps the fetched input data to their corresponding functional units. Some instructions use multiple functional units and so the outputs of the first functional unit re-route to the next one. The operands can be either scalar or vector, and they can be fetched from the SPM or the registerfile. The final outputs of the functional units will connect again to the mapping unit, in which they will be written back to the SPMs. Below is a brief snippet from the RTL of the input operand mapper, as for the output mapping, the assignments would be similar but reversed.

```
------ INPUT OPERAND MAPPING   ------------------------------------------------------------------------------
      if (decoded_instruction_SPE_lat(KDOTP_bit_position)   = '1'  or   -- dot product instruction
          decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1') and – dot product instruction with post scaling
         (MVTYPE_SPE = "01" or MVTYPE_SPE = "10") then
        mul_operands(0) <= spi_data_read(0) and spi_data_read_mask;
        mul_operands(1) <= spi_data_read(1) and spi_data_read_mask;
        if dotp = '1' then
          accum_operands  <= out_mul_results;
        elsif dotpps = '1' then
          shift_amount    <= MPSCLFAC_SPE;
          shifter_operand <= out_mul_results;
          accum_operands  <= out_shifter_results;
        end if;
      end if;
      …
      if decoded_instruction_SPE_lat(KADDV_bit_position) = '1' then – vector-vector add instr
        adder_operands(0)   <= spi_data_read(0);
        adder_operands(1)   <= spi_data_read(1);
      end if;

      if decoded_instruction_SPE_lat(KSVADDSC_bit_position)  = '1' and  -- vector-scalar add instruction
         MVTYPE_SPE = "10" then
        adder_operands(0)   <= spi_data_read(0);
        for i in 0 to SIMD-1 loop
          adder_operands(1)(31+32*(i) downto 32*(i))   <= spi_data_read(1)(31 downto 0);
        end loop;
      end if;

      if decoded_instruction_SPE_lat(KSRAV_bit_position) = '1' or – right arithmetic shift instruction
        decoded_instruction_SPE_lat(KSRLV_bit_position) = '1' then  -- right logic shift instruction
        shifter_operand     <= spi_data_read(0);
        shift_amount        <= RS2_Data_IE_lat(4 downto 0); -- map the scalar value (shift amount)
      end if;
      …
      if decoded_instruction_SPE_lat(KRELU_bit_position)  = '1' then – relu instruction
        relu_operands <= spi_data_read(0);
      end if;
```

The **control unit** controls the requests to fetch the input operands and write the output results. It also halts the vector processor in case the source SPMs are being accessed by the load-store unit. When the SPE gets a halt signal, all the data in the pipes will maintain their state, and the hardware loops will stop counting until the SPM accessed becomes free. The Control for KADDV and KDOTP is shown below. Other instructions have a similar control.

```
      if decoded_instruction_SPE_lat(KADDV_bit_position)  = '1' or  -- control for KADDV and KSUBV instructions
```

```
 decoded_instruction_SPE_lat(KSUBV_bit_position)  = '1' then
 if adder_stage_3_en = '1' then
  wb_ready <= '1';  -- the results of the final stage are ready to be written back
 elsif recover_state = '1' then
  wb_ready <= '1';  -- latch the writeback ready signal for as soon as the write is granted
 end if;
 if MVSIZE_READ > (0 to Addr_Width => '0') then – keep on reading until all the data has been fetched
  spe_to_spm(to_integer(unsigned(rs1_to_spi_lat)))(0) <= '1';  -- assign vs1 to the first SPI read port
  spe_to_spm(to_integer(unsigned(rs2_to_spi_lat)))(1) <= '1';  -- assign vs2 to the second SPI read port
  spi_req(to_integer(unsigned(rs1_to_spi_lat)))  <= '1';  -- request vs1
  spi_req(to_integer(unsigned(rs2_to_spi_lat)))  <= '1';  -- request vs2
  spi_read_addr(0)  <= RS1_Data_IE_lat(Addr_Width - 1 downto 0);  -- send the address of  vs1
  spi_read_addr(1)  <= RS2_Data_IE_lat(Addr_Width - 1 downto 0);  -- send the operand of vs2
 end if;
 if MVSIZE_WRITE > (0 to Addr_Width => '0') then
  nextstate_SPE <= spe_exec; -- latch the execute state of the SPE
  busy_SPE_internal_wires := '1'; – the SPE is considered busy until all the outputs are written
 end if;
 if wb_ready = '1' then – first batch of the vector results becomes ready
  spi_we(to_integer(unsigned(rd_to_spi_lat)))    <= '1'; -- enable the writeback
  spi_write_addr <= RD_Data_IE_lat; -- send the write address which is incremented by the hw_loops
 end if;
end if;
…
if decoded_instruction_SPE_lat(KVRED_bit_position)    = '1' or --Control of the accumulator using instructions
  decoded_instruction_SPE_lat(KDOTP_bit_position)     = '1' or
  decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1' then
 if accum_stage_3_en = '1' then
  wb_ready <= '1';
 elsif recover_state = '1' then
  wb_ready <= '1';
 end if;
 if MVSIZE_READ > (0 to Addr_Width => '0') then – keep on reading until all the data has been fetched
  if vec_read_rs2_SPE = '1' then
   spi_req(to_integer(unsigned(rs2_to_spi_lat))) <= '1'; -- request vs2
   spe_to_spi(to_integer(unsigned(rs2_to_spi_lat)))(1) <= '1'; -- assign vs2 to the second SPI read port
   spi_read_addr(1)  <= RS2_Data_IE_lat(Addr_Width - 1 downto 0); -- send the address of  vs2
  end if;
  spi_req(to_integer(unsigned(rs1_to_spi_lat))) <= '1'; -- request vs1
  spe_to_spm(to_integer(unsigned(rs1_to_spi_lat)))(0) <= '1'; -- assign vs1 to the first SPI read port
  spi_read_addr(0)  <= RS1_Data_IE_lat(Addr_Width - 1 downto 0); -- send the address of  vs1
  nextstate_SPE <= spe_exec;
  busy_SPE_internal_wires := '1';
 elsif MVSIZE_WRITE = (0 to Addr_Width => '0') then
  nextstate_SPE <= spe_init; -- return to the init state when the accumulation is done
 else
  nextstate_SPE <= spe_exec;  -- latch the execute state until all the elements have accumulated
  busy_SPE_internal_wires := '1';  -- the SPE is considered busy until all the values have been accumulated
 end if;
 if wb_ready = '1' then  -- final scalar result is ready
  spi_we(to_integer(unsigned(rd_to_spi_lat)))    <= '1'; -- enable the writeback
  spi_write_addr <= RD_Data_IE_lat; -- send the write address of the scalar value
 end if;
end if
```

The SPE has five different **functional units (FUs)**. All the units work with different data types (8-bit, 16-bits, 32-bit) both signed and unsigned. Three of the FUs work in partial mode; the adder, shifter, and the multiplier. The partial FUs increase the parallelism for smaller data width elements while maintaining a small area occupation. Table 1.1 shows how many operations we do in one cycle in every FU and for each data type when the SIMD parameter is configured to be 1. Bigger SIMD configurations will double the number of parallelisms on all the functional units.

Table.5. 1 Type, and parallelism of the functional units in the SPE

Instruction	FU Type	Data Type	Parallelism
Adder	Partial	32	1*SIMD
		16	2*SIMD
		8	4*SIMD
Shifter	Partial	32	1*SIMD
		16	2*SIMD
		8	4*SIMD
Multiplier	Partial	32	1*SIMD
		16	2*SIMD
		8	**2*SIMD**
Accumulator	Normal	32	1*SIMD
		16	2*SIMD
		8	**2*SIMD**
ReLu	Normal	32	1*SIMD
		16	2*SIMD
		8	4*SIMD

We can see the **partial adder** from figure 5.3, there are a set of four 8-bit adders cascaded together. To produce 8-bit sums, the initialization block will configure the adders to block the carries propagated from the partial sums giving four 8-bit sums as outputs. For 16-bit additions, only the first and the third adders are allowed to propagate their carries, giving two 16-bit outputs. While for the 32-bit sums all the carries are allowed to be propagated giving one 32-bit output.The adders as seen from figure 5.3 are split into two pipe stages, the carry from the lower 16 bits, goes to the upper sixteen bits through a register and not a wire.

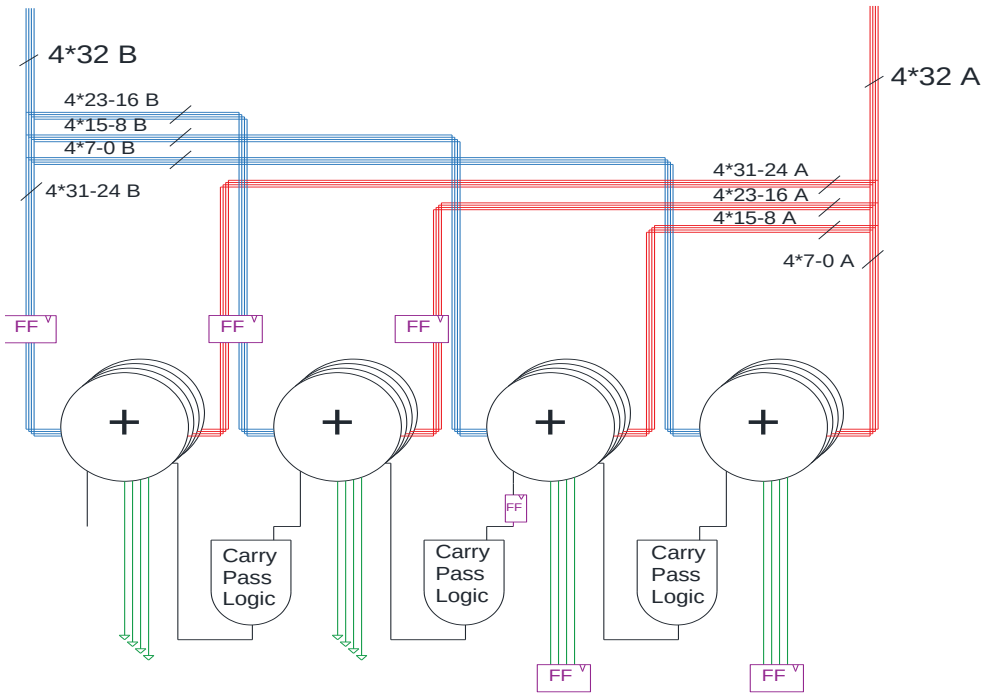


Figure.5.3. Partial Adder Circuit in SIMD=4

The RTL describing the behavior of the SIMD pipelined partial adders is shown below.

```
for i in 0 to SIMD-1 loop
  if (adder_stage_1_en = '1' or recover_state_wires = '1') then
    add_8_0_wire(i)   <= std_logic_vector('0' & unsigned(adder_ops(0)(7+8*(4*i)  downto 8*(4*i))) +
                                                unsigned(adder_ops(1)(7+8*(4*i)  downto 8*(4*i)))  +
                                                twos_complement(0+(4*i)));
    add_16_8_wire(i)  <= std_logic_vector('0' & unsigned(adder_ops(0)(15+8*(4*i) downto 8+8*(4*i)))  +
                                                 unsigned(adder_ops(1)(15+8*(4*i) downto 8+8*(4*i))) +
                                                 carry_8_wire(i) +
                                                 twos_complement(1+(4*i))); --
    -- Carries are either passed or blocked for the 9-th, 17-th, and 25-th bits
    carry_8_wire(i)  <= add_8_0_wire(i)(8)   and carry_pass(0); -- carry_pass is configured in the init stage
    carry_16_wire(i) <= add_16_8_wire(i)(8)  and carry_pass(1); -- carry_pass is configured in the init stage
  end if;
```

```
for i in 0 to SIMD-1 loop – index 'i' is for the SIMD depth of the SPMU
  if (adder_stage_2_en = '1' or recover_state_wires = '1') then
    add_24_16_wire(i) <= std_logic_vector('0' & unsigned(adder_ops_lat(0)(7+8*(2*i) downto 8*(2*i)))+
                                                 unsigned(adder_ops_lat(1)(7+8*(2*i) downto 8*(2*i))) +
                                                 carry_16(i) + twos_complement(2+(4*i)));
    add_32_24_wire(i) <= std_logic_vector('0' & unsigned(adder_ops_lat(0)(15+8*(2*i) downto 8+8*(2*i))) +
                                                 unsigned(adder_ops_lat(1)(15+8*(2*i) downto 8+8*(2*i))) +
                                                  carry_24_wire(i) + twos_complement(3+(4*i)));
     -- All the 8-bit adders are lumped into one output write signal that will write to the scratchpads
     -- Carries are either passed or blocked for the 9-th, 17-th, and 25-th bits
     carry_24_wire(i) <= add_24_16_wire(i)(8) and carry_pass(2); -- carry_pass is configured in the init stage
    end if;
  end loop;
```

```
if add_en = '1' and halt_spe_lat = '0' then
   carry_16          <= carry_16_wire; -- latch the wires
   add_8_0    <= add_8_0_wire;
   add_16_8  <= add_16_8_wire;
   for i in 0 to SIMD-1 loop  – index 'i' is for the SIMD depth of the SPMU
     if (adder_stage_2_en = '1' or recover_state_wires = '1') then
        -- All the 8-bit adders are lumped into one output signal that will write to the scratchpads
      out_adder_results(31+32*(i) downto 32*(i)) <= add_32_24_wire(i)(7 downto 0) & -- form the output result
                                                    add_24_16_wire(i)(7 downto 0) &
                                                    add_16_8(i)(7 downto 0) &
                                                    add_8_0(i)(7 downto 0);
     end if;
   end loop;
  end if;
  for i in 0 to SIMD-1 loop – index 'i' is for the SIMD depth of the SPMU
   for j in 0 to 1 loop  -- index 'j' loops through the upper two 8-bit adders
    adder_ops_lat(j)(15 +16*(i) downto 16*(i)) <= adder_ops(f)(j)(31+32*(i) downto 16+32*(i)); -- latch the ops
   end loop;
  end loop;
```

For the 32-bit **multiplier** the partial multiplication structure is based on four 16-bit multipliers, according to the following implementation:

$$A_{31\text{-}0} * B_{31\text{-}0} = [(A_{31\text{-}16} << 16) + A_{15\text{-}0}] * [(B_{31\text{-}16} << 16) + B_{15\text{-}0}]$$

This method can generate two 8-bit, or two 16-bit MULs per cycle, or one 32-bit MUL per cycle. The circuit describing the multiplier is shown in figure 5.4. If the data type is set to 8-bit, or 16-bit, then the middle multiplications (A_L*B_H and A_H*B_L) will be masked with zeros to block the accumulation of the partial multiplications into making a 32-bit output. The actual multiplier does not use right shifters to give this 16-bit offset of zeros, instead it just concatenates a 16-bit zero vector to the upper portions of the partial multiplications.

The reason this operation was not divided to use 8-bit multipliers instead, was because one DSP [45] slice is utilized in the FPGA whether an 8-bit or a 16-bit multiplication is done. So, for our current implementations of the multipliers, we will only get twice the speed-up for 8-bits of data and not four times as in the case of the partial adders. One note also, the multipliers upper 32-bit outputs are ignored so we do not emulate any 'MULH' operation, because they are not required in our applications.

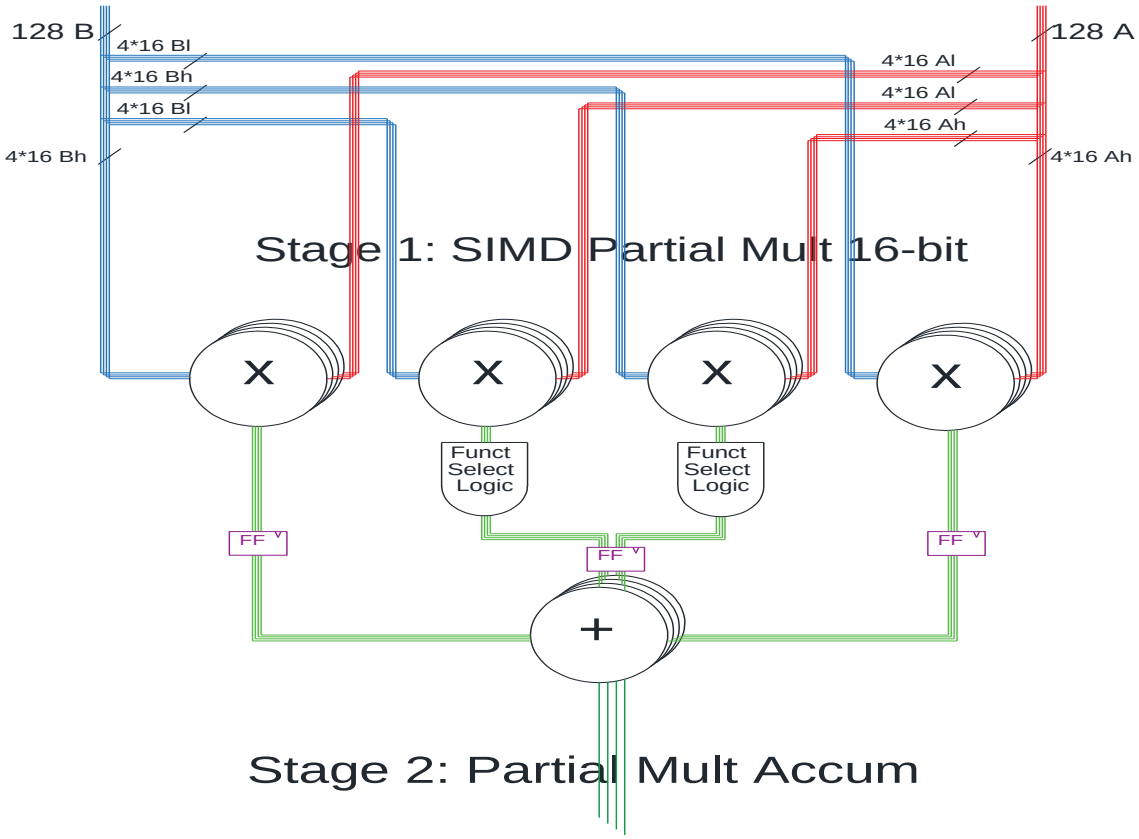


Figure.5.4. Partial Multiplier Circuit in SIMD=4

```
1   ------ Synchronous Partial Multiplication Stage 1 ----------------------------------------------------------------------------
2   if halt_spe_lat = '0' then  – index 'i' is for the SIMD depth of the SPMU
3     if mul_en = '1' and (mul_stage_1_en = '1' or recover_state_wires = '1') then
4       for i in 0 to SIMD-1 loop
5         mul_a(31+32*(i)  downto 32*(i)) <= std_logic_vector(unsigned(mul_ops(0)(15+16*(2*i+1) downto 16*(2*i+1)))*
6                                                           unsigned(mul_ops(1)(15+16*(2*i+1) downto 6*(2*i+1))));
7         mul_b(31+32*(i)  downto 32*(i)) <= std_logic_vector((unsigned(mul_ops(0)(16*(2*i+1) - 1  downto 16*(2*i)))*
8                                                             unsigned(mul_ops(1)(15+16*(2*i+1) downto 16*(2*i+1))))
9                                                         and unsigned(FUNCT_SELECT_MASK));
10        mul_c(31+32*(i)  downto 32*(i)) <= std_logic_vector((unsigned(mul_ops(0)(15+16*(2*i+1) downto 16*(2*i+1)))*
11                                                            unsigned(mul_ops(1)(16*(2*i+1) - 1 downto 16*(2*i))))
12                                                        and unsigned(FUNCT_SELECT_MASK));
13        mul_d(31+32*(i)  downto 32*(i)) <= std_logic_vector(unsigned(mul_ops(0)(16*(2*i+1)  - 1 downto 16*(2*i)))*
14                                                          unsigned(mul_ops(1)(16*(2*i+1) - 1  downto 16*(2*i))));
```

```
    end loop;
  end if;
end if;
-------------------------------------------------------------------------------------------------------------------------
```

```
------ Synchronous Partial Multiplication Stage 2 ----------------------------------------------------------------------
 if mul_en = '1' and (mul_stage_2_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
   for i in 0 to SIMD-1 loop
     out_mul_results((Data_Width-1)+Data_Width*(i) downto Data_Width*(i))  <=
                        (std_logic_vector(unsigned(mul_tmp_a(i)) +
                                          unsigned(mul_tmp_b(i)) +
                                          unsigned(mul_tmp_c(i)) +
                                          unsigned(mul_tmp_d(i))));
   end loop;
 end if;
-------------------------------------------------------------------------------------------------------------------------
```

```
------ Combinational Partial Multiplication  ---------------------------------------------------------------------------
    if mul_en = '1' and (mul_stage_2_en = '1' or recover_state_wires = '1') then
      for i in 0 to SIMD-1 loop
        if MVTYPE_SPE /= "10" then
          ---------------------------------------------------------------------------
          mul_tmp_a(i) <= (mul_a(15+16*(2*i)  downto 16*(2*i)) & x"0000");
          mul_tmp_d(i) <= (x"0000" & mul_d(15+16*(2*i)  downto 16*(2*i)));
          ---------------------------------------------------------------------------
        elsif MVTYPE_SPE = "10" then
         -- The upper 32-bit results of the multiplication are discarded in the SPMU (Ah*Bh)
          mul_tmp_b(i) <= (mul_b(15+16*(2*i) downto 16*(2*i)) & x"0000");            --  (Ah*Bl)
          mul_tmp_c(i) <= (mul_c(15+16*(2*i) downto 16*(2*i)) & x"0000");       -- (Al*Bh)
          mul_tmp_d(i) <= (mul_d(31+32*(i)   downto 32*(i)));                   -- (Al*Bl)
        end if;
      end loop;
    end if;
-------------------------------------------------------------------------------------------------------------------------
```

The **partial right shifter** in the SPE works in the opposite manner (Figure 5.5). One 32-bit right logic shifter slides the input operands and computes one 32-bit shifted output. If the data width was 16-bits, the init config will configure the data to mask the data sliding form one data value to the other. It will execute as follows: The two 16 bits data will go into the right shifter, the output of the shifter will be sent to the next stage where the lower bits of the **upper** 16-bit input that were slided into the upper bits of the **lower** 16-bit input will be masked with a bit a zero if the shift was logical, and sign extended if the shift was arithmetic. A similar approach is applied for 8-bit data types.

The SPMU does not include a left shifter, instead the partial multipliers can be used for left shifting. As for the implementation of the right shifter, it was implemented to be used for pre-scaling and post-scaling of the input and output data to be used in convolutions.

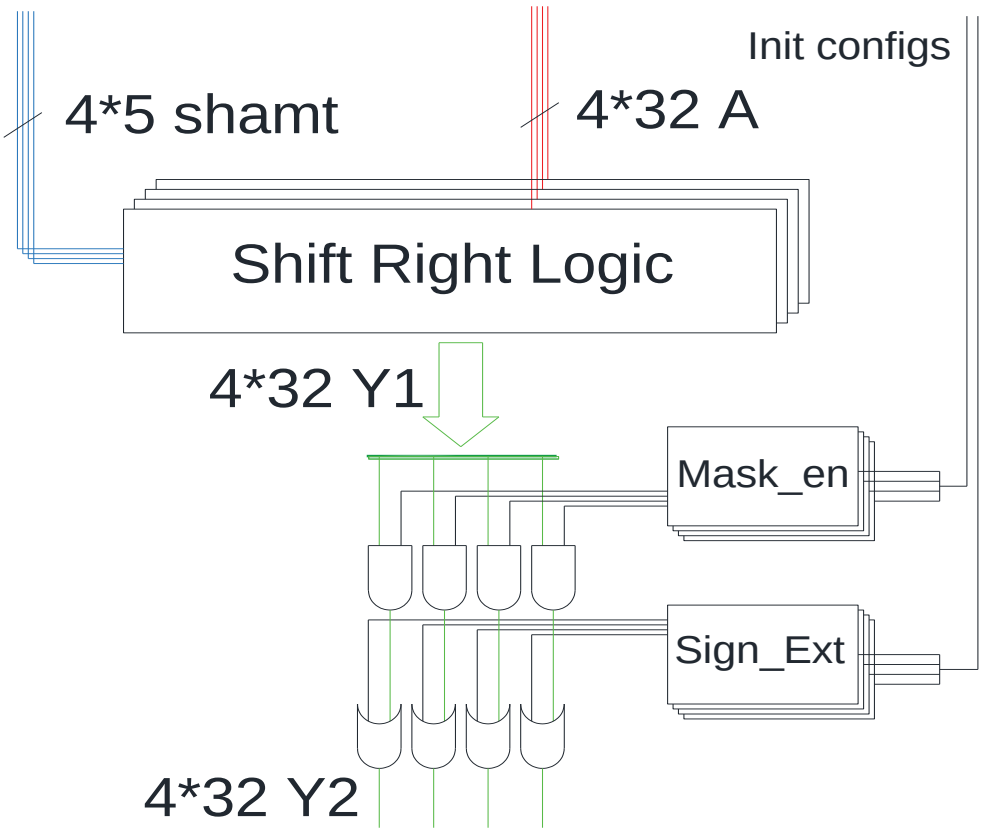


Figure.5.5. Partial Right Shifter Circuit in SIMD=4

```
------ Synchronous Partial Shifter Stage 1 ----------------------------------------------------------------------------------
 if shift_en = '1' and (shifter_stage_1_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
   for i in 0 to SIMD-1 loop
     shifter_op(31+32*(i) downto 32*(i)) <= to_stdlogicvector(to_bitvector(shifter_op(31+32*(i) downto 32*(i))) srl
                                          to_integer(unsigned(shift_amount)));  -- shift as if it was a 32-bit value
   end loop;
   if MVTYPE_SPE = "00" then
     for i in 0 to 4*SIMD-1 loop -- latch the sign bits
       shifter_op_lat(7+8*i downto 8*i) <= (others => shifter_op(7+8*i)); -- latch 8-bit data sign bit for arith shifts
     end loop;
   elsif MVTYPE_SPE = "01" then
     for i in 0 to 2*SIMD-1 loop -- latch the sign bits
       shifter_op_lat(15+16*i downto 16*i) <= (others => shifter_op(15+16*i)); -- latch 16-bit data sign bit for arith shifts
     end loop;
   elsif MVTYPE_SPE = "10" then
     for i in 0 to SIMD-1 loop -- latch the sign bits
       shifter_op_lat(31+32*i downto 32*i) <= (others => shifter_op(31+32*i)); -- latch 32-bit data sign bit
     end loop;
   end if;
 end if;
---------------------------------------------------------------------------------------------------------------------- ----------
```

```
------ Synchronous Partial Shifter Stage 2 ----------------------------------------------------------------------------------
  if shift_en = '1' and (shifter_stage_2_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
    if    MVTYPE_SPE = "10" then
      for i in 0 to SIMD-1 loop
        out_shifter_results(31+32*(i) downto 32*(i)) <= shifter_op_lat_wire(31 +32*(i) downto 32*(i))  or
                                                        shifter_op(31+32*(i) downto 32*(i));
      end loop;
    elsif MVTYPE_SPE = "01" or (decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1' and
          MVTYPE_SPE = "00") then
```

```
    -- KDOTPPS8 is added here because the element number loaded per cycle for mul ops is the sane for 8, and 16 types
      for i in 0 to 2*SIMD-1 loop
        out_shifter_results(15+16*(i) downto 16*(i)) <= shifter_op_lat_wire(15 +16*(i) downto 16*(i)) or
                                                        (shifter_operand(15+16*(i) downto 16*(i)) and
                                                         shift_enabler(15 downto 0));
      end loop;
    elsif MVTYPE_SPE = "00" then
      for i in 0 to 4*SIMD-1 loop
        out_shifter_results(7+8*(i) downto 8*(i)) <=  shifter_operand_lat_wire(7 +8*(i) downto 8*(i)) or
                                                      (shifter_operand(7+8*(i) downto 8*(i))  and
                                                       shift_enabler(7 downto 0));
      end loop;
    end if;
  end if;
-------------------------------------------------------------------------------------------------------------------------
```

```
------ Combinational Partial Shifter ----------------------------------------------------------------------------------
 if shift_en = '1' and halt_spe_lat = '0' then
  if MVTYPE_SPE = "01" then
    shift_enabler(15 - to_integer(unsigned(shift_amount(3 downto 0))) downto 0) <= (others => '1');
  elsif MVTYPE_SPE = "00" then
    shift_enabler(7 -  to_integer(unsigned(shift_amount(2 downto 0))) downto 0) <= (others => '1');
  end if;
  if (decoded_instruction_SPE_lat(KSRAV_bit_position) = '1' or
      decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1') and
      MVTYPE_SPE = "10" then    -- 32-bit sign extension for srl in stage 1
    for i in 0 to SIMD-1 loop
      shifter_op_lat_wire(31+32*(i) downto 31 - to_integer(unsigned(shift_amount(f)(4 downto 0)))+32*(i)) <=
      shifter_operand_lat(31+32*(i) downto 31 -  to_integer(unsigned(shift_amount(f)(4 downto 0)))+32*(i));
    end loop;
  elsif (decoded_instruction_SPE_lat(KSRAV_bit_position) = '1' or
         decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1') and
         MVTYPE_SPE = "01" then -- 16-bit sign extension for srl in stage 1
    for i in 0 to 2*SIMD-1 loop
      shifter_operand_lat_wire(15+16*(i) downto 15 - to_integer(unsigned(shift_amount(3 downto 0)))+16*(i)) <=
      shifter_operand_lat(       15+16*(i) downto 15 - to_integer(unsigned(shift_amount(3 downto 0)))+16*(i));
    end loop;
  elsif (decoded_instruction_SPE_lat(KSRAV_bit_position) = '1'  or
         decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1') and
         MVTYPE_SPE = "00" then  -- 8-bit  sign extension for srl in stage 1
    for i in 0 to 4*SIMD-1 loop
      shifter_operand_lat_wire(7+8*(i) downto 7 - to_integer(unsigned(shift_amount(2 downto 0)))+8*(i)) <=
      shifter_operand_lat(       7+8*(i) downto 7 - to_integer(unsigned(shift_amount(2 downto 0)))+8*(i));
    end loop;
  end if;
 end if;
-------------------------------------------------------------------------------------------------------------------------
```

The remaining two functional units are a **2-stage accumulator**, which accumulates an input vector source into a scalar output, and a **ReLu unit** that rectifies all negative vector elements to zero.

```
------ Two Stage Accumulator SIMD 2-------------------------------------------------------------------------------------
  if (decoded_instruction_SPE_lat(KDOTP_bit_position)   = '1'  or
      decoded_instruction_SPE_lat(KDOTPPS_bit_position) = '1'  or
      decoded_instruction_SPE_lat(KVRED_bit_position)   = '1') and
      MVTYPE_SPE = "10" then
    if (accum_stage_1_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
      accum_partial_results_stg_1(31 downto 0)  <= std_logic_vector(unsigned(accum_op(31 downto 0))  +
                                                                    unsigned(accum_op(63  downto 32)));
```

```
      end if;
      if (accum_stage_2_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
            out_accum_results(f) <= std_logic_vector(unsigned(accum_partial_results_stg_1(31 downto 0)) +
                                                     unsigned(out_accum_results));
      end if;
    elsif (decoded_instruction_SPE_lat(KDOTP_bit_position)    = '1'  or
           decoded_instruction_SPE_lat(KDOTPPS_bit_position)  = '1'  or
           decoded_instruction_SPE_lat(KVRED_bit_position)    = '1') and
          (MVTYPE_SPE = "01" or MVTYPE_SPE = "00") then
      if (accum_stage_1_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
        accum_partial_results_stg_1(15 downto 0)   <= std_logic_vector(unsigned(accum_op(15 downto 0))   +
                                                                       unsigned(accum_op(31  downto 16)));
        accum_partial_results_stg_1(31 downto 16) <= std_logic_vector(unsigned(accum_op(47 downto 32))  +
                                                                       unsigned(accum_op(63 downto 48)));
      end if;
      if (accum_stage_2_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
        spe_out_accum_results <= std_logic_vector(unsigned(accum_partial_results_stg_1(15 downto 0))  +
                                                  unsigned(accum_partial_results_stg_1(31 downto 16)) +
                                                  unsigned(out_accum_results));
      end if;
    end if;
-------------------------------------------------------------------------------------------------------------------------
```

```
------ Synchronous Single Stage ReLu ------------------------------------------------------------------------------
 if (relu_stage_1_en = '1' or recover_state_wires = '1') and halt_spe_lat = '0' then
   if    MVTYPE_SPE = "10" then – ReLu for 32-bit data type
     for i in 0 to SIMD-1 loop
       if spe_in_relu_operands(31+32*(i)) = '1' then
         spe_out_relu_results(31+32*(i) downto 32*(i)) <= (others => '0');
       else
         spe_out_relu_results(31+32*(i) downto 32*(i)) <= spe_in_relu_operands(31+32*(i) downto 32*(i));
       end if;
     end loop;
   elsif MVTYPE_SPE = "01" then – ReLu for 16-bit data type
     …
   end if;
 end if;
```

5.3.2. Scratchpad Memory Interface

The engine is interfaced with a set of SPMs through the Scratchpad Memory Interface. Each SPM in the SPI has a read and write port, and every SPM-line has a set of banks that hold a 32-bit word. The number of banks in an SPM is dependent on the SIMD configuration chosen. For example, a configuration with SIMD 4 has four banks. Each of the banks has a read and write port, and the total width of the ports in the SPM will be 128-bits (i.e. 32-bits*4). When a fetch request is granted the data will be read on the next cycle. The RTL below illustrates the implementation of the SPMs in the T13.

```
------ Scratchpad Memory Generation ------------------------------------------------------------------------------
-- 3D array, of memory, the 1st dimension defines the size of each word, the 2nd is number of words in a bank, and the 3rd
is the number of banks.
signal mem : array_3d(SIMD*SPM_NUM-1 downto 0)(2**(Addr_Width-(SIMD_BITS+2))-1 downto 0)(Data_Width-1
downto 0);
attribute ram_style : string;
attribute ram_style of mem : signal is "block";

 spm_banks    : for h in 0 to SIMD*SPM_NUM -1 generate
```

```
    spm_logic: process(clk_i) --
    begin
      if(clk_i'event and clk_i='1') then
        sc_data_rd(h) <= mem(SIMD*SPM_NUM + h)(to_integer(unsigned(sc_addr_rd(h))));
       if sc_we(h) = '1' then         --write mode
         mem(SIMD*SPM_NUM + h)(to_integer(unsigned(sc_addr_wr(h)))) <= sc_data_wr(h);
       end if; -- we
      end if; -- clk
    end process;
  end generate spm_banks;
-------------------------------------------------------------------------------------------------------------------------
```

An SPM read or write access will fetch or write an entire line in one cycle. If the fetch pointer was not pointing to the beginning of the line, the data fetched will be from the line being indexed, and the next line as well, therefore maintaining the fetching of one complete line per cycle.

Misaligned fetches go into a read-rotator circuit to make it appear as if the fetching is from the beginning of the line. The rotator gives a one extra cycle of latency to execute the instruction. In this manner operand_a[i] will always be aligned with operand_b[i] and go the same functional unit. Without rotation, misaligned accesses might send operand_a[i] and operand_b[i+2] to go to the same functional unit, and that produces erroneous outputs. During the result write, the result will be rotated back with a write rotator to go to the correct bank indexed in the write address.

```
------ Synchronous Write Rotator ----------------------------------------------------------------------------------------
    for i in 0 to SIMD-1 loop -- index i loops the words inside each SPM
      if (to_integer(unsigned(spm_write_addr(SIMD_BITS+1 downto 0))) = 4*i) and (i /= 0) then
        wr_offset(i-1 downto 0) <= (others => '1');
      end if;
    end loop;
    for i in 0 to SIMD-1 loop -- index i loops the words inside each SPM
      if (to_integer(unsigned(spm_write_addr(SIMD_BITS+1 downto 0))) = 4*i) then
        for j in 0 to SIMD-1 loop
          if j <= (SIMD-1)-i then
           spm_data_write_int_wire(31+32*(j+i) downto 32*(j+i)) <= spm_data_write_wire(31+32*j downto 32*j);
          elsif j > (SIMD-1)-i then
            spm_data_write_int_wire(31+32*(j-(SIMD-1)+(i-1)) downto 32*(j-(SIMD-1)+(i-1))) <=
                                                                    spn_data_write_wire(31+32*j downto 32*j);
          end if;
        end loop;
      end if;
    end loop;
-------------------------------------------------------------------------------------------------------------------------
```

```
------ Synchronous Read Rotator -----------------------------------------------------------------------------------------
    for k in 0 to 1 loop – index k loops between the two read data operands of the SPI
      for i in 0 to SIMD-1 loop  -- index i loops the words inside each SPM
        if (to_integer(unsigned(spm_read_addr(k)(SIMD_BITS+1 downto 0))) = 4*i) and (i /= 0) then
          rd_offset(k)(i-1 downto 0) <= (others => '1');
        end if;
      end loop;
      for i in 0 to SIMD-1 loop -- index i loops the words inside each SPM
        if (to_integer(unsigned(spm_read_addr_lat(k))) = 4*i) then
          for j in 0 to SIMD-1 loop
            if j >= i then
              spm_data_read_wire(k)(31+32*(j-i) downto 32*(j-i))
                                                  <= spm_data_read_int_wire(k)(31+32*j downto 32*j);
            elsif j < i then
```

```
            spm_data_read_wire(k)(31+32*((SIMD-1)-i+(j+1)) downto 32*((SIMD-1)-i+(j+1)))
                                                    <= spm_data_read_int_wire(k)(31+32*j downto 32*j);
          end if;
        end loop;
      end if;
    end loop;
  end loop;
-------------------------------------------------------------------------------------------------------------------------
```

The SPI has a serialized access grant unit, in which the instruction that comes first in program order will either lock the read and write access of a certain scratchpad. And since T13 is an in-order processor, there will never be data hazards with the serialized access grant.

LSU accesses to SPI read or write one bank at a time instead of writing the entire SPM line at once. A bank interleaver will loop consecutively between each bank in the SPM, and once it reaches the last line of the bank it increments the read or write address, and loops back to bank 0 of the SPM. The RTL describing the implementation of the bank interleaver is shown below.

```
--- Synchronous bank counter ------------------------------------------------------------------------
-- Increments the bank count inside each spm memory
  if data_rvalid_i = '1' then
    if spm_word_count = SIMD-1 then
      spm_word_count <= 0;
    else
      spm_word_count <= spm_word_count + 1;
    end if;
  end if;
-------------------------------------------------------------------------------------------------------
```

```
---  LSU read port ----------------------------------------------------------------------------------
  if ls_data_gnt_i(i) = '1' then     -- LSU read port
    if harc_LS_wire = h then  -- data reads the register from the bank counter to index
      ls_sc_data_read_wire_replicated(h) <= sc_data_rd(h)((SIMD)*i + sc_word_count(h));
    end if;
  end if;

  if ls_spi_req(i) = '1' then        -- LSU read port
    if harc_LS_wire = h then -- address reads the wire from the bank counter to index
      spm_addr_rd(h)(spm_word_count_wire + (SIMD)*i) <= ls_spm_read_addr;
    end if;
  end if;
-------------------------------------------------------------------------------------------------------
```

5.4. SPMU Implementations

This section explores a set of hardware accelerator schemes whose architecture was described in section 5.3, and describes how each one can be used in exploiting the T13 core.

5.4.1. Shared-SPMU (Shared-SPI, Shared-SPE):

The first approach used when augmenting a hardware accelerator to the IMT architecture was having a Shared SPMU being accessed by all the harts in the core. Figure 5.6 shows a block diagram of this scheme. The schematic is very identical to that one showed in figure 5.2. In order to access the Shared-

SPMU, a request signal is sent from the decode stage. If the Shared-SPMU is busy, the pipeline will be halted until it becomes free again.

In order to minimize the halts to the pipeline, functional units can be set to execute in SIMD. Increasing the SIMD multiplies the functional units in the core, and the number of banks in each SPM. The core could be configured to process data in parallel of up to 256-bits per cycle (SIMD 8 max). Smaller data types perform even faster when boosting the data level parallelism. Since most of the functional units work in partial mode, and can compute of up to four results per unit as seen from table 5.1. In the scheme in figure 5.6, all the harts share the same memory space, and the same execution units. The SPMU can work in superscalar with other non-SPMU instructions, however, when an SPMU instruction is decoded, and SPMU unit is busy, then the instruction pipeline will be halted in this scheme.

The RTL describing the implementation of the Shared-SPMU is the same code that was shown in section 5.3.

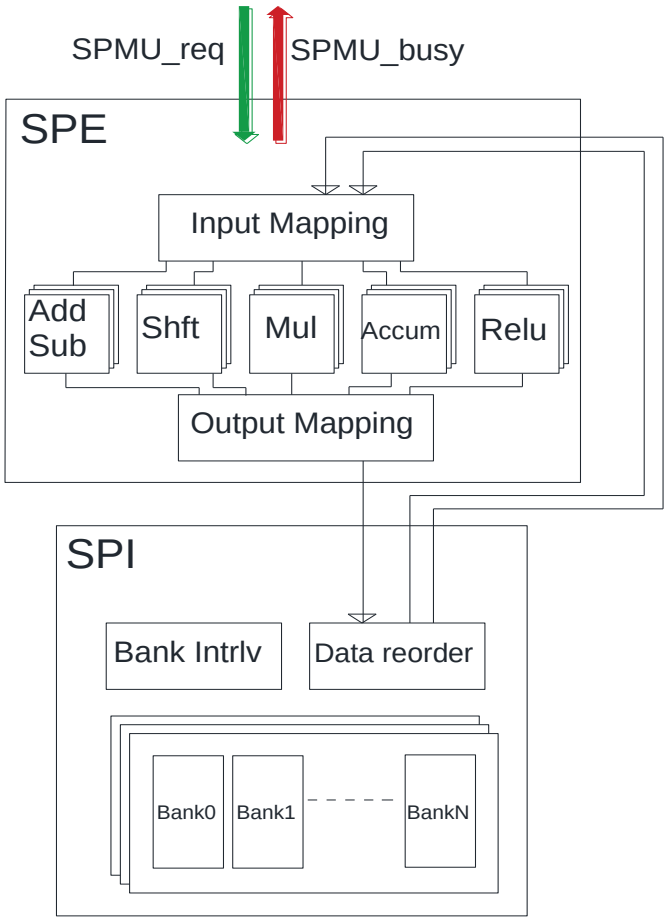


Figure.5.6. Diagram of the Shared-SPMU, all accesses to the SPMU are shared by all the harts

5.4.2. Dedicated-SPI Shared-SPE

The second hardware accelerator scheme is called the Dedicated-SPI Shared-SPE. The diagram showing its implementation is shown in figure 5.7. In this hardware scheme, every hart in the T13 core has its own dedicated memory space, but they all share the same functional units. It can be compared to a multi-threaded hardware accelerator, in which the threads share the access to the logical elements [38]. In the Dedicated-SPI Shared-SPE, any contention to a functional unit is processed by a contention handler to determine which hart requested the access first. Since the hart instructions are issued in order, then there will never be simultaneous requests, and no race conditions. An SPMU

busy signal in this hardware scheme will only block SPMU instructions belonging to the same hart thus minimizing the pipeline halts in the SPMU a lot. Note that there is a buffer to hold the instruction data **for each hart**. This gives a great speed advantage over the Shared-SPMU approach as it exploits thread level parallelism, and still maintains minimal architectural complexity, as no instruction issue logic is needed to issue out of order.

Every hart can load data to its own SPI, and not to any other SPI. In this manner, the SPMs of each hart can have overlapping memory addresses. For example, hart 2 can perform burst loads 'kmemlds' from the main memory to the SPI(2) only, and hart 1 using the same pointers used in *kmemld* instruction from hart 2 can do the same. The decoding of the entire SPM address space becomes much easier to handle, and makes it also easier for the programmer that will be managing the SPMU address space. In a similar manner, all SPMU arithmetic instructions read and write from and to their own SPIs only.

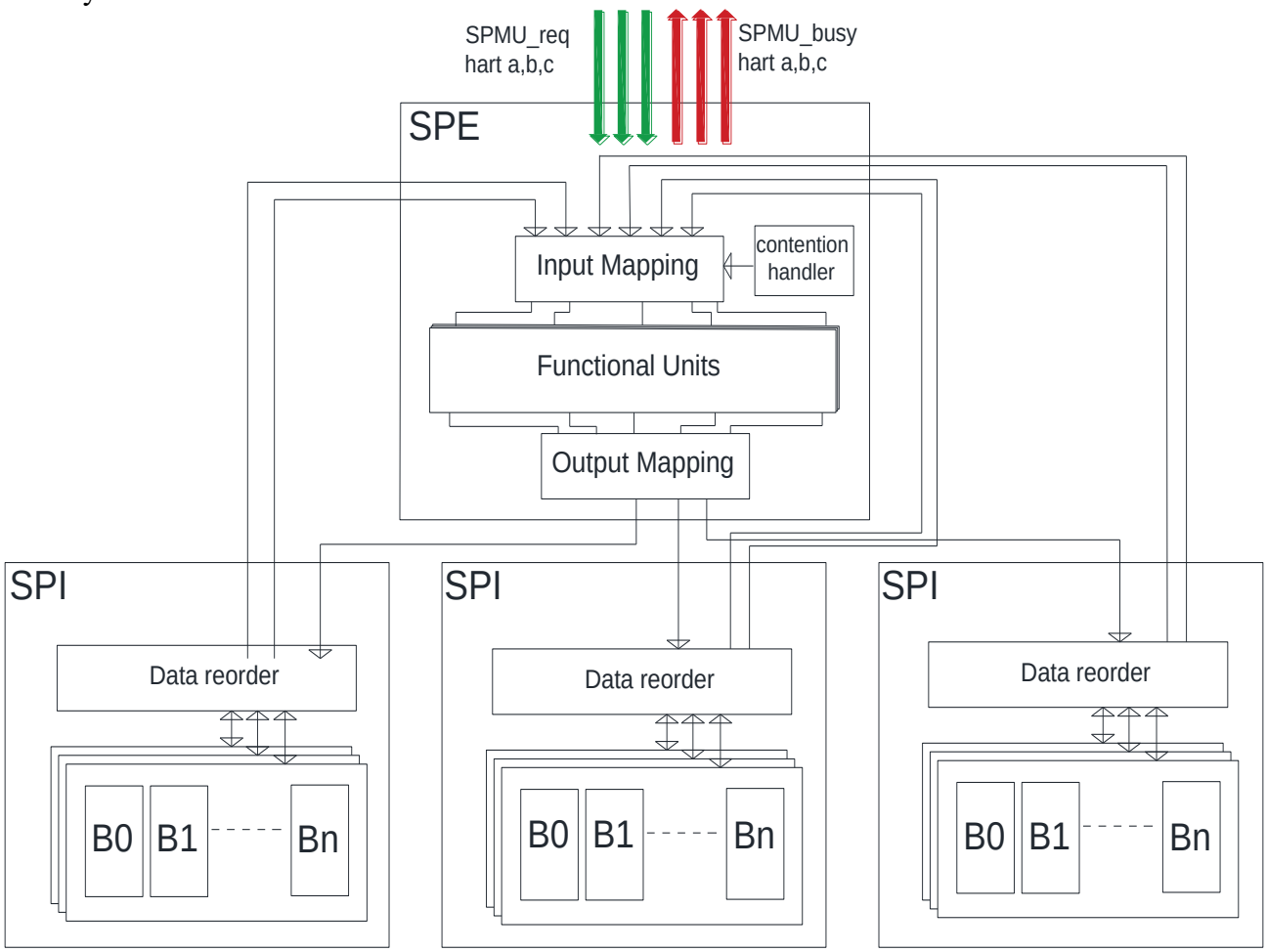


Figure.5.7. Diagram of dedicated SPI shared SPE model. Each hart has a dedicated set of scratchpads, busy signals will only block the hart belonging to the same SPMU

If the user needs to broadcast some input data to all the SPIs, they can execute another type of load instruction called broadcast load "kbcastld". When using *kbcastld*, if the user wants to send some input data to SPM(i), then the *kbcastld* will broadcast the data to the SPM(i) of each SPI. This broadcasting operation relives the core from having to fill three memories sequentially.

Changes to the RTL required to handle this are minimal. First every SPI is replicated with a "for generate" structure is needed and a signal to distinguish the load is a broadcast as shown below,

```
1  SPM_replicated : for h in accl_range generate
2   -- The index 'h' now refers to each SPI
3  SPI_Unit_comb : process(all))
```

```
begin
…
…
    if data_rvalid_i = '1' then        -- LS write port
      if ls_spi_req(i) = '1' and ls_spi_we(i) = '1' and ls_spi_wr_gnt = '1' then
        if harc_LSU_wire = h or spm_bcast = '1' then – spm_bcast indicates we have kbcastld, and we always enter the 'if'
          spm_we(h)((SIMD)*i + spm_word_count(h)) <= '1';
          spm_data_wr(h)(spm_word_count(h) + (SIMD)*i) <= lsu_data_write_wire(31 downto 0);
          spm_addr_wr(h)(spm_word_count(h) + (SIMD)*i) <= lsu_write_addr;
        end if;
      end if;
    end if;
…
…
end process;

end generate SPM_replicated;
```

In addition to the SPI, the SPE must have a new mapping unit, each hart must have its own hardware loops, and there must be a functional unit contention access handler.

The RTL below describes, a brief implementation of how the new mapping unit should be. One disadvantage from the implementation of the mapping unit below, is that all these input SPI operands mapping to these different functional units requires a huge set of multiplexers to map inputs and outputs appropriately.

```
------ Input Mapping ------------------------------------------------------------------------------------
        -- The index 'h' refers the dedicated SPI in the core, and maps them to the adder
        if decoded_instruction_SPE_lat(h)(KADDV_bit_position) = '1' then
          adder_ops(0)   <= spi_data_read(h)(0);
          adder_ops(1)   <= spi_data_read(h)(1);
        end if;

------ Output Mapping -----------------------------------------------------------------------------------
        -- The output results of the adder are again mapped to the appropriate SPI indexed in 'h'
        if decoded_instruction_SPE_lat(h)(KADDV_bit_position) = '1' then
          spe_sc_data_write_wire_int(h) <= out_adder_results;
        end if;
-------------------------------------------------------------------------------------------------------------
```

The FU contention handler on the other hand is a bit more complex to implement. The RTL below shows logic behind the functional unit grant handler. As seen in the RTL below, every functional unit has its own handler, and any reservation on a busy functional unit stores the ID of the hart requesting the access inside a buffer, the buffer write-pointer gets incremented as soon as the request becomes registered, and another hart can reserve access to the busy functional unit at the new write-pointer value. As soon as the functional unit becomes free. The buffer is read, the read-pointer is incremented, and the grant will be given to the hart ID stored in the buffer.

```
------ Synchronous FU access handler ------------------------------------------------------------------
    for h in accl_range loop
      for i in 0 to 4 loop  – loops through the five functional units (add, shift. mul, acc, relu)
        if fu_req(h)(i) = '1' then  -- if a reservation was made, to use a functional unit, store the hart_ID
          fu_issue_buffer(i)(to_integer(unsigned(fu_wr_ptr(i))))  <= std_logic_vector(to_unsigned(h,TPS_CEIL));
          if unsigned(fu_wr_ptr(i)) = THREAD_POOL_SIZE - 2 then
            fu_wr_ptr(i) <= (others => '0');
          else
            fu_wr_ptr(i) <= std_logic_vector(unsigned(fu_wr_ptr(i)) + 1); -- increment the write pointer
          end if;
```

```
            if fu_gnt_en(h)(i) = '1' then
              if unsigned(fu_rd_ptr(i)) = THREAD_POOL_SIZE - 2 then
                fu_rd_ptr(i) <= (others => '0');
              else
                fu_rd_ptr(i) <= std_logic_vector(unsigned(fu_rd_ptr(i)) + 1); -- increment the read pointer
              end if;
            end if;
          end if;
        end loop;
      end loop;
-------------------------------------------------------------------------------------------------------------------------------
```

```
------ Combinational FU access handler ----------------------------------------------------------------
    for h in accl_range loop
      fu_gnt_wire(h) <= (others => '0');
      fu_gnt_en(h)    <= (others => '0');
      if add_en_pending_wire(h) = '1' and busy_add_wire = '0' then
        fu_gnt_en(h)(0) <= '1';
      end if;
      if shift_en_pending_wire(h) = '1' and busy_shf_wire = '0' then
        fu_gnt_en(h)(1) <= '1';
      end if;
      …
         for i in 0 to 4 loop – loops through the five functional units (add, shift. mul, acc, relu)
          if fu_gnt_en(h)(i) = '1' then
            -- give a grant to fu_gnt(h)(i), such that the 'h' index points to the thread in "fu_issue_buffer"
            fu_gnt_wire(to_integer(unsigned(fu_issue_buffer(i)(to_integer(unsigned(fu_rd_ptr(i)))))))(i) <= '1';
          end if;
         end loop;
       …
       end loop;
-------------------------------------------------------------------------------------------------------------------------------
```

Note that the Dedicated-SPI Shared-SPE approach which already exploits thread level parallelism of the T13 core, can still be configured to exploit the data level parallelism of the T13 by configuring the SPMU to execute with larger SIMD settings.

5.4.3. Dedicated-SPMU (Dedicated SPI, Dedicated-SPE)

The Dedicated-SPMU approach, as the name implies assigns a dedicated hardware accelerator to each hart. Just like the previous implementation was compared to a multi-threaded accelerator, this implementation can be compared to a multi-core accelerator. The term multicore can be compared to the CUDA cores in NVIDIA Tesla [39]. Each SPMU has its own SPI and SPE, there is no contention handler needed at all, since each hart will have its own set of functional units. Figure 5.8 shows the implementation of such an approach. The advantage to this approach over the Dedicated SPI Shared-SPE approach is that this approach further decreases the stalls to the instruction pipeline since there will never be contention over functional units. Also, the mapping unit of this approach is also much less complex since it does not need that huge crossbar to map the operands to the functional units, and its implementation will follow that Shared-SPMU.

Like the Dedicated SPI Shared-SPE approach, this unit has one instruction buffer for each hart. A pipeline stall will only happen when the decode stage has an SPMU instruction going to the same hart of a busy SPMU. Also, similarly the SPI implementation of the Dedicated-SPMU approach is exactly the same to that of the previous approach, and it still maintains the support for the broadcast load

instruction, However, a disadvantage for this approach is that this approach might utilize a big area since all the pipelined functional units are replicated.

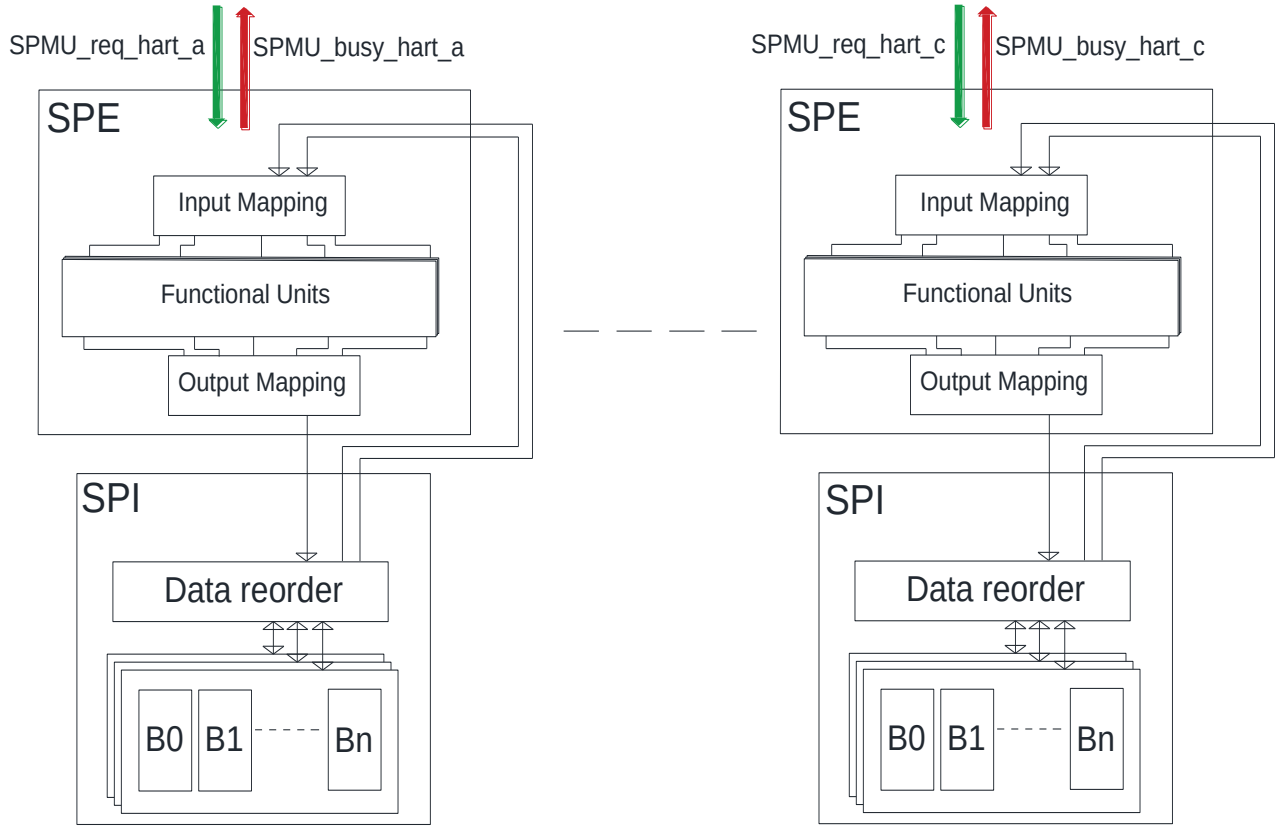


Figure.5.8. Diagram of Dedicated-SPMU, each hart has a dedicated SPE and SPI, a busy signal will only block the hart belonging to the same SPMU

The brief RTL below illustrates how all the signals in the SPMU must be changed relative to the Shared-SPMU approach, in which all the signals now have a new dimension which is called *<accl_tange>,* that ranges through number of hardware accelerators in the core. If the SPMU is replicated as in this case, the *accl_range* is equal to the THREAD_POOL_SIZE. While if the replication was disabled, *accl_range* would become zero. Also as seen in the RTL that a "for-generate" must be added to replicate the assignments in the SPE just like the SPI assignments were replicated in the previous approach. This way, each process assigns to its own dimension indexed in *'h'*.

```
1  signal wb_ready                : std_logic_vector(accl_range);
2  signal SIMD_RD_BYTES           : array_2d_int(accl_range);
3  signal MVSIZE_WRITE            : array_2d(accl_range)(Addr_Width downto 0);
4
5   SPE_replicated : for h in accl_range generate  -- The h index loops through the acc_range above
6   …
7            if wb_ready(h) = '1' then
8              if to_integer(unsigned(MVSIZE_WRITE(h))) >= SIMD_RD_BYTES(h) then
9                MVSIZE_WRITE(h) <= std_logic_vector(unsigned(MVSIZE_WRITE(h)) - SIMD_RD_BYTES(h));
10             else
11               MVSIZE_WRITE(h) <= (others => '0');              -- decrement the remaining bytes
12             end if;
13           end if;
14  …
15  end generate
```

5.5. Performance evaluation of the SPMU implementations.

In order to benchmark the performance of the T13 core when executing vector operations, various tests have been developed. The first batch is a basic series of instruction level testing. These tests benchmark the performance contribution of different approaches provided in the SPMU, that helped boost the execution of arithmetic-vector operations. The second batch of tests, is a set of matrix convolution being executed with the SPMU, in order to show the how the hardware schemes introduced in section 5.4 performed. Lastly, we show results of running entire layers of DCNN on the SPMU, and we compare its performance to the T03, and Riscy cores from Pulpino. Details about the implementation of the tests are laid out in the chapter 6.

5.5.1. Instruction Level Testing:

In order to benchmark some implementations in the SPMU, a set of basic arithmetic tests were performed to see which implementations provided the largest performance boost. Figure 5.9 shows the number of clock cycles took to perform an arithmetic operation in the T13 without using any hardware accelerator, but still utilizing all the harts in the core.

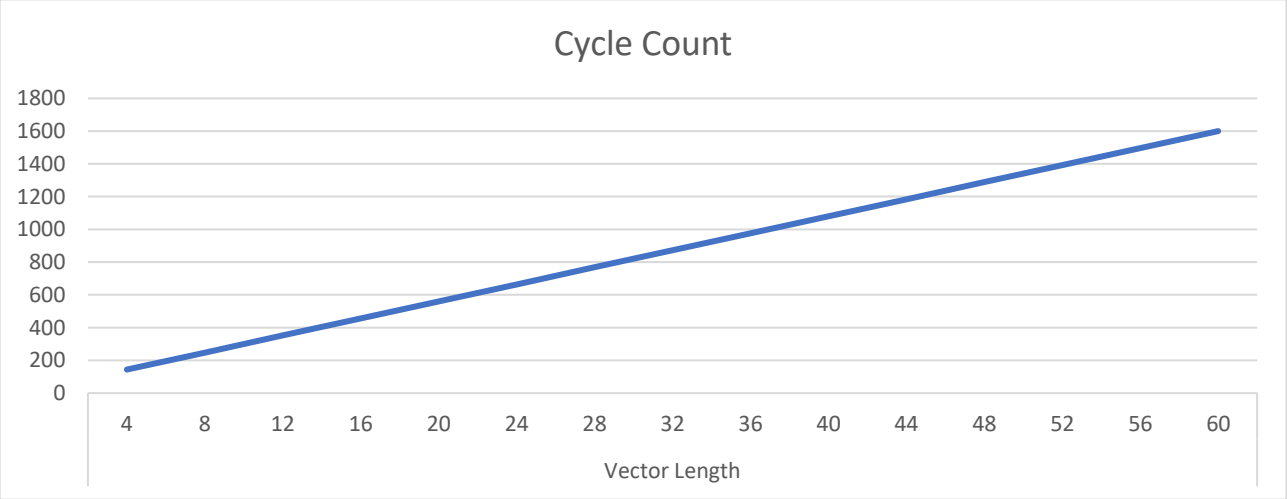


Figure.5.9. Number of cycles taken to perform an arithmetic vector operation without the SPMU

In figure 5.10, the same vector-arithmetic operations were performed with the SPMU with the different data types (8,16,32). However, they were performed using software loops instead of zero-overhead loops (hardware loops). The convolutions were run on the Shared-SPMU scheme

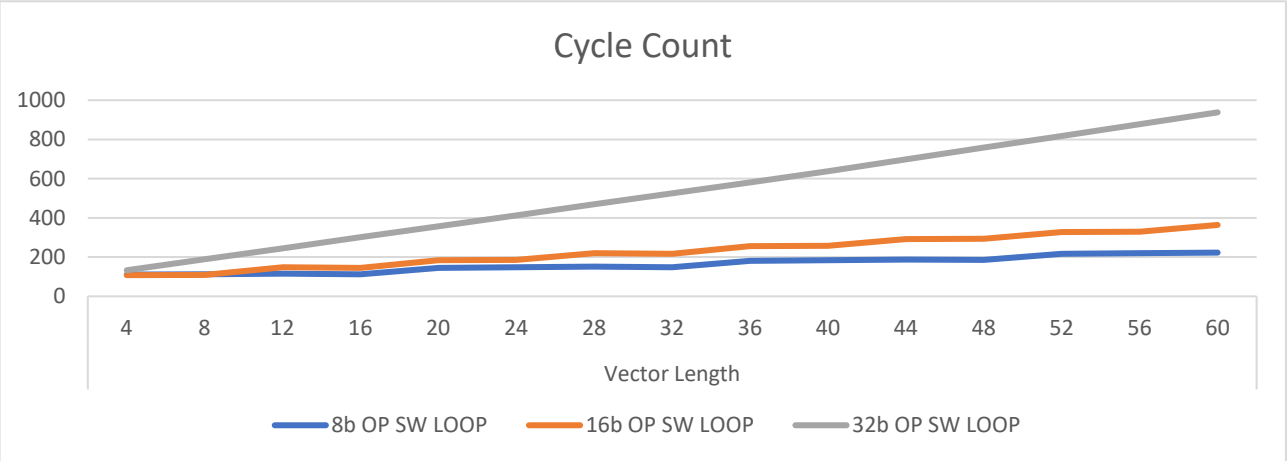


Figure.5.10. Cycle time using the SPMU with SIMD=1 and hardware loops disabled

configured with no data level parallelism (SIMD=1). Figure 5.10, shows the advantage of using the low latency local scratchpad memories.

Comparing figures 5.9 and 5.10 for small vector sizes, the boost was not very evident. However, as the vector sizes grew, the tests that were running on the SPMU using sw-loops and SIMD equal to zero, showed that the cycle time grew with a smaller slope then that of the non-accelerated test. This test clearly outlines the advantage of using low latency scratchpad memories to using the registerfiles. Such that the total number of cycles dropped by more than 40% for vectors of sixty elements.

The reason for the speedup is obvious, since the non-accelerated operations writing to the registerfile will have to push the old data to the stack memory to make way for the new computed results, and then load back the data from the stack when it needs to be read. While when using the SPMU will load the input data once from the main memory with a burst load instruction. Then, stores the final results at the end of the operation with a burst store back to the main memory.

Smaller data width such as 16, and 8-bit performed even better since they are more parallel than the 32-bit operations even though the SIMD of the SPMU is set to one. The nature of SPMU using partial functional units and replicating the non-partial ones will show this very good performance with the tiny slope relative to the 32-bit operations.

Figure 5.11 shows the advantage of using the zero-overhead loops or hardware loops in the SPMU. The hardware loops relieve the core from augmenting the following overhead of instructions:

- Incrementing the address of source operand 1.
- Incrementing the address of source operand 2.
- Decrementing the number of elements left to execute.
- Branching to the beginning of the loop if the number elements is not zero.

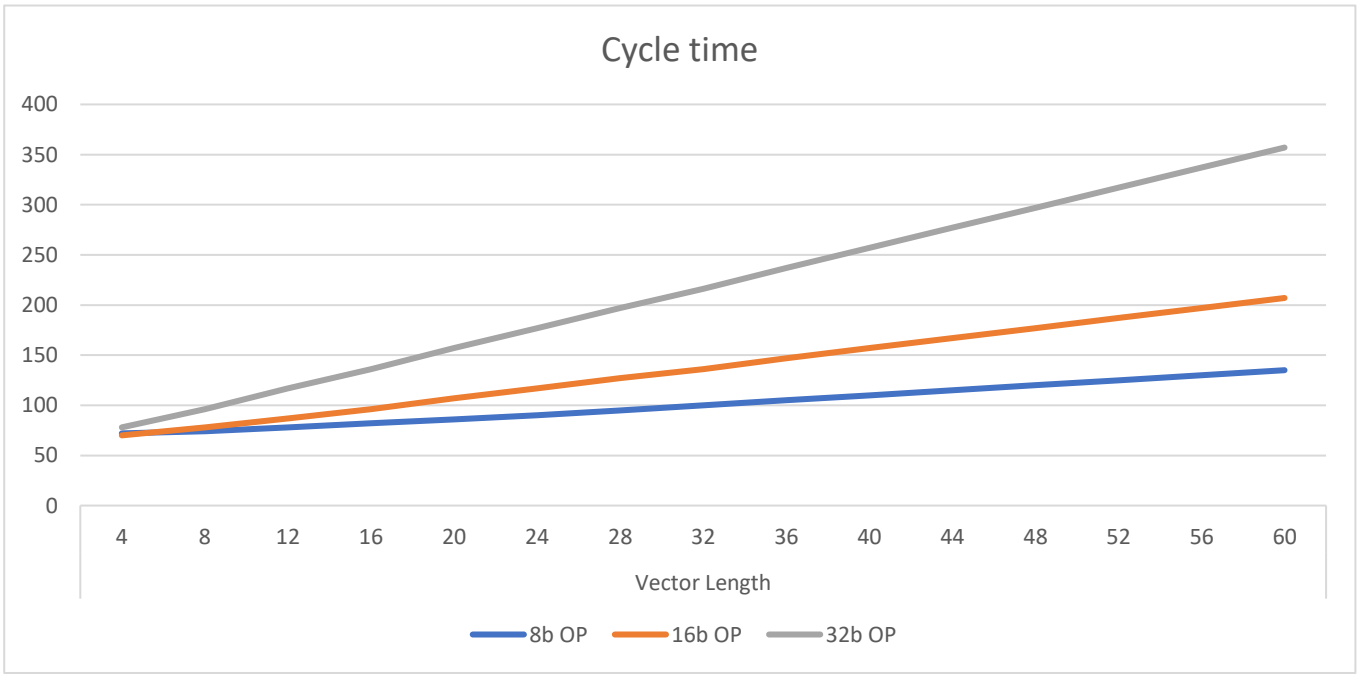


Figure.5.11. Cycle time using the SPMU with SIMD=1 and hardware loops enabled

Enabling the hardware loops in the SPMU, boosted the performance for all vector sizes, such that the speed boost was over 170% for large vectors, and almost 100% for small vectors comparing to the

sw-loop approach. While comparing to the non-accelerated from figure 5.9 approach we can see the speed boost to go over 350% for large vectors.

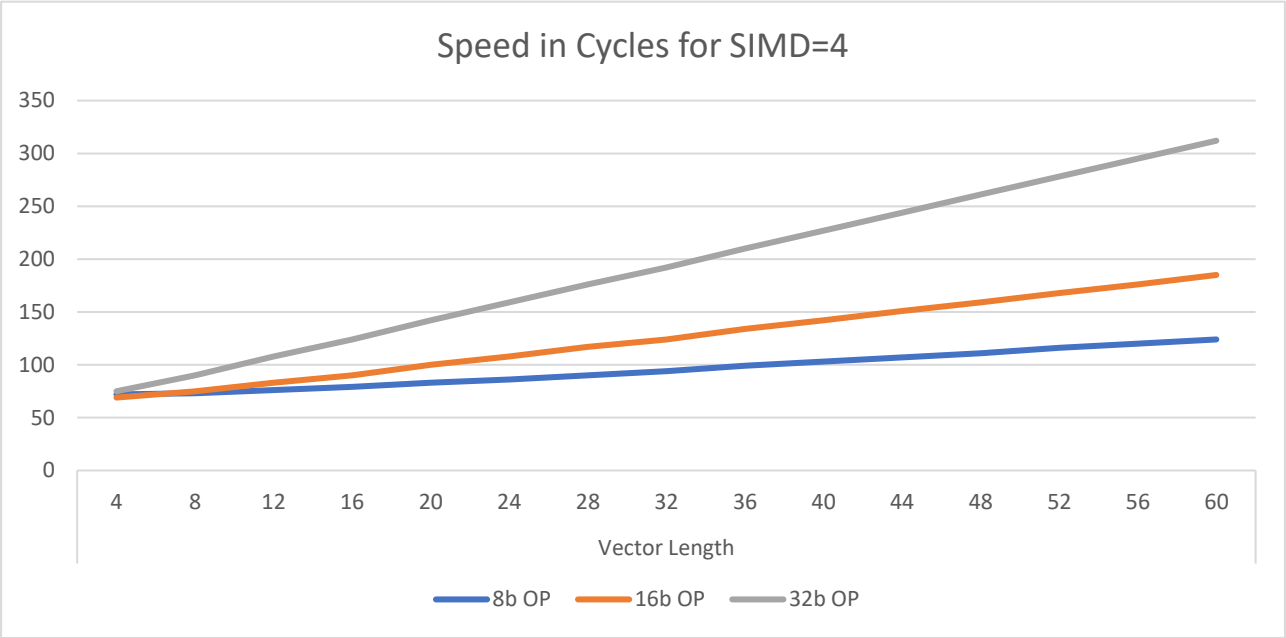


Figure.5.12. Cycle time using the SPMU with SIMD=4 and hardware loops enabled

Finally, figure 5.12 reports the cycle time when executing the same test, however with increasing the data level parallelism by setting the SIMD equal to four.

Boosting the data level parallelism was the least contributor out of all the implementations to the performance boosts. Such that the speed boost was barely visible for small vectors, and for large vectors, the speed boost was about 15% over the previous approach. Not only that, but the area increases from replicating the functional units, and the registers that hold the data in the pipelines of functional units, and the read and write SPM rotators size increase can be regarded as considerably large for such small performance contributions.

More reports regarding the area utilization will be discussed in the section 5.6.

5.5.2. Routine Level Testing

Libraries have been made using the SPMU instructions in order to perform matrix convolutions. Details about the implementation of the convolutions are included in chapter 6. The matrix convolutions included different square matrix sizes, typically 4x4, 8x8, 16x16, and 32x32. The data types used were only 32-bit integers. That is because the neural network test used, uses these data types as well. The convolution tests have been run on the hardware schemes introduced in section 5.4. Each hardware scheme was configured with different SIMD configurations (1, 2, 4 and 8) to show the contribution of the data level parallelism in each. Table 5.2 reports the cycle time for each matrix convolution on each SPMU hardware scheme as well as the non-accelerated versions of the T13 and the native PULPino Riscy cores.

Now as we delve in the evaluation of the different hardware schemes from section 5.4. I will be using some terminology to refer to the schemes in order to be brief:

- *DLP approach:* means increasing the data level parallelism in the Shared-SPMU such that we go from SIMD-1 to SIMD-8.

- *TLP approach:* means that we go from the Shared-SPMU SIMD-1 scheme to the Dedicated-SPMU SIMD-1 or Dedicated-SPI_Shared-SPE SIMD-1 schemes that exploit thread level parallelism.

- *Hybrid Approach:* means that we go from the Shared-SPMU SIMD-1 scheme to the Dedicated-SPMU SIMD-8 or Dedicated-SPI_Shared-SPE SIMD-8 schemes that exploit both data level parallelism and thread level parallelism.

The evaluation begins as follows starting from small matrix convolutions. Looking at table 5.2, small matrix convolutions (4x4) performed by the different SPMU configurations gave approximately 2-3 times the speed-up relative to performing the convolutions on the non-accelerated T13 core (No_ACCL_RV32IM), and more than 2 times the speed-up when being compared to the Riscy core itself and 4-7 times comparing it to the Zeroriscy core. Riscy achieves a low cycle count as it exploits the hardware loops and custom DSP extensions, thus there instruction count decreases as much of the software overhead is performed in hardware.

Table.5.2. Cycle number to execute a set of convolutions for different SPMU configurations

Core		SIMD	Cycle Count			
			4x4	8x8	16x16	32x32
Klessydra T13	Shared SPMU	1	1105	3060	9727	34201
		2	895	2245	6261	20374
		4	824	1768	4607	13444
		8	824	1613	3692	10069
	Dedicated SPMU	1	626	1493	3887	13536
		2	629	1190	3123	8681
		4	560	1190	2543	7148
		8	560	1152	2543	6006
	Dedicated SPI Shared SPE	1	663	1521	4153	13565
		2	638	1274	3280	9167
		4	573	1213	2688	7473
		8	573	1079	2580	6285
	NO_ACCL (RV32IM)	NA	1819	5737	20714	79230
	NO_ACCL_(RV32EM)	NA	2355	7821	28927	111891
	NO_ACCL (RV32I)	NA	4883	17877	69087	272394
	NO_ACCL_ (RV32E)	NA	5568	20707	80478	318084
RISCY		NA	1377	4247	15088	57020
ZeroRiscy		NA	2510	8111	29583	113793
ZeroRiscy (no RV32M)		NA	6406	23601	91233	360081
MicroRiscy		NA	7380	27385	106271	419618

Comparing the SPMU schemes to the T13 cores that did not use the accelerator, acceleration became more evident with bigger convolutions such that 32x32 convolutions achieved up to 5-7 times the speed-up using the DLP or TLP approach alone. Hybrid approaches exploiting both DLP and TLP gained up to 16 times the speed-up. While comparing the results to the PULPino Riscy cores we have even a larger speed-up on bigger convolutions such that hybrid SPMU approaches had up to 12 times the speed-up relative to the Riscy core, and 10 times the speedup comparing to Zeroriscy.

Moving on to comparing the SPMU schemes with themselves in the bigger matrix convolutions, using the DLP approach alone we saw more than 3.4 times the speed-up, while using the TLP approach alone gave approximately 2.5 times the speed-up. Exploiting both DLP and TLP we saw 5.7 times the speed-boost. In bigger matrix convolutions not only did the TLP and DLP approaches

gave higher speed-ups than the smaller matrix convolutions, however the rate of the improvement of the DLP was faster than the rate of the improvement in the TLP such that in bigger matrix it appeared better to use the DLP approach of the TLP approach.

Many other important notes can also be taken from table 2. First, the Dedicated-SPI_Shared-SPE approach when being compared to the Dedicated-SPMU approach has achieved from a minimum of 94% to a maximum of 99% the speed boost when compared to the Dedicated-SPMU. This showed that in fact sharing the resources impacts the speed only a tiny bit as far as matrix convolutions are concerned.

Second, the speed-up in both approaches exploiting TLP *(Dedicated-SPMU, and Dedicated-SPI_Shared-SPE)* can show how much pipeline stalls had an effect on the speed when comparing to the Shared-SPMU.

Third, the embedded approaches (RV32E implementations) that were aimed at decreasing the registerfile footprint in the IMT architectures had somewhat discouraging performance results. such that comparing the NO_ACCL_RV32EM to NO_ACCL_RV32IM showed a speed degradation of 30% in small matrix convolutions and the degradation went up to 41% in the large convolutions, this nonlinear degradation obtained from bigger convolutions is mostly due to the increase in the memory transfers to the stack section of the data memory since the registerfile in the RV32E extension has very little space allocated for saved registers as opposed to the normal registerfile in the RV32I.

Figure 5.13 shows the contribution of the boost from exploiting the DLP, TLP, and the Hybrid approach were both DLP and TLP are exploited. Obviously, the Hybrid had the biggest boost in the cycle time, however, comparing the DLP and TLP alone. We saw that for small vectors TLP was better at giving higher performances and the matrices grew larger (i.e. beyond 16x16) we saw that TLP boost remained the same, and the DLP boost then became better than the boost from the TLP.

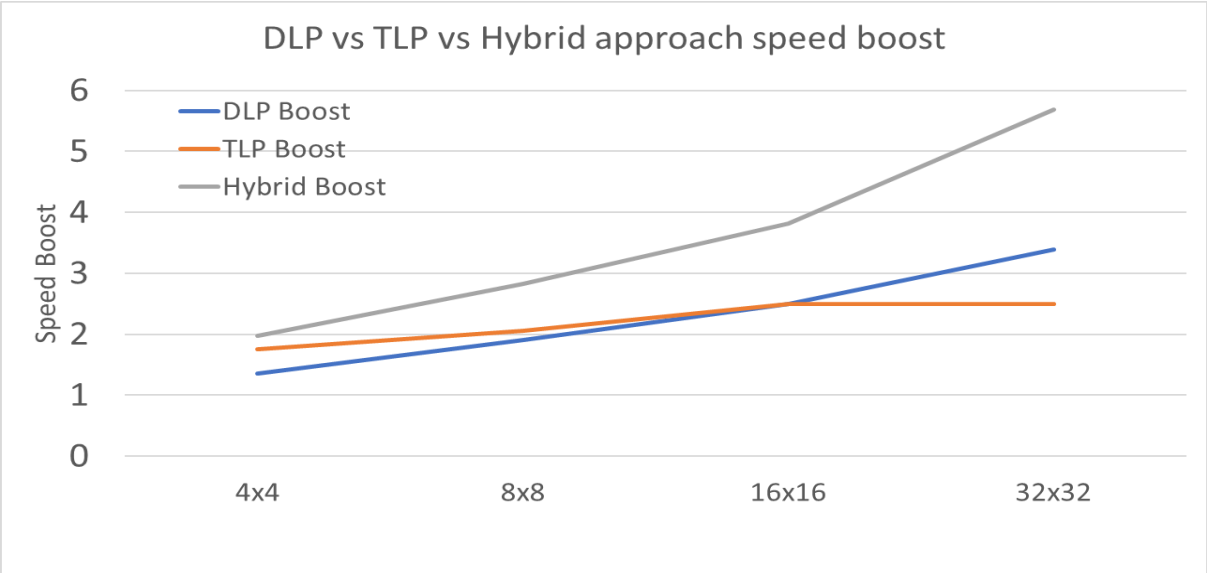


Figure.5.13. Speed boost from exploiting the DLP, TLP, and both together (Hybrid)

The reason behind not seeing much speed-ups due to DLP in small vectors is that:

- The nature of the SPMU being already superscalar with the other non-SPMU execution units does well in hiding the latencies of its instructions.

- The size of the vectors is small such that doubling the functional units can save only a few cycles and not much more.

Table 5.3 shows the top frequency of the T13, and the PULPino Riscy cores after a post-synthesis implementation. The timing constraint used in the synthesis was 1ns, which is a tight constraint that compels Vivado to synthesize the fastest layouts possible.

Table.5.14. Top frequency for each T13 configuration and Riscy Cores

Core		SIMD	Top Frequency (MHz)
Klessydra T13	Shared SPMU	1	165.29
		2	151.17
		4	141.16
		8	129.99
	Dedicated SPMU	1	156.35
		2	130.58
		4	111.51
		8	108.35
	Dedicated SPI Shared SPE	1	140.06
		2	131.04
		4	116.80
		8	102.31
	NO_ACCL (RV32IM)	NA	206.31
	NO_ACCL_(RV32EM)	NA	209.60
	NO_ACCL (RV32I)	NA	185.53
	NO_ACCL_ (RV32E)	NA	216.64
RISCY		NA	91.36
ZeroRiscy		NA	117.23
ZeroRiscy (no RV32M)		NA	133.08
MicroRiscy		NA	146.11

Vivado was able to generate fast layouts for all the hardware schemes for SIMD configurations 1 and 2. However, the top speed witnessed a sharper drop as the DLP grew larger (SIMD 4 and SIMD 8) especially for the hybrid schemes exploiting both TLP and DLP. For the dedicated SPMU approach, the area overhead became large enough so that the FPGA slices were being placed farther away from each other, thus increasing the net delay between the FPGA slices themselves.

While the Dedicated-SPI-Shared-SPE approach witnessed even a larger drop in the top frequency for large SIMD configurations. Looking at the timing report from Vivado, we saw that the crossbar that maps the Dedicated-SPI input data buses to the shared SPE functional units became the critical path in the SPMU for both SIMD 4 and 8 implementations. One approach to make this scheme faster is to pipeline the crossbar, and divide the critical path. However, we will see in the next why this is not a very favorable approach.

Figure 5.14 shows the execution time it takes to run the convolutions on all the schemes from table 5.3 when operating at the maximum frequency. The figure was separated into two margins left side being the SPMU hardware schemes while the right side being the non-accelerated implementations of T13 and Riscy cores. The reason they were separated was so that very high cycle count on the right side does not saturate the improvements of the TLP and DLP in the SPMU schemes on the left side.

Beginning with our evaluations, increasing the DLP in bigger convolutions such as 16x16 and 32x32 did actually provide a decrease in the cycle time for all the SPMU schemes. Smaller convolutions actually got slower when increasing the DLP, that is because of the sharp drop in the top frequency seen from table 5.3 when increasing the DLP was bigger than the boost in the cycle time.

One conclusion can be made here, that although increasing the DLP does multiply the processor's ability to process data in parallel and thus decrease the cycle count, however, your processor might in turn perform slightly worse especially when the vectors being worked on are smaller (figure 5.14 convolution 4x4). Comparing the T13 non accelerated schemes to the Riscy cores.

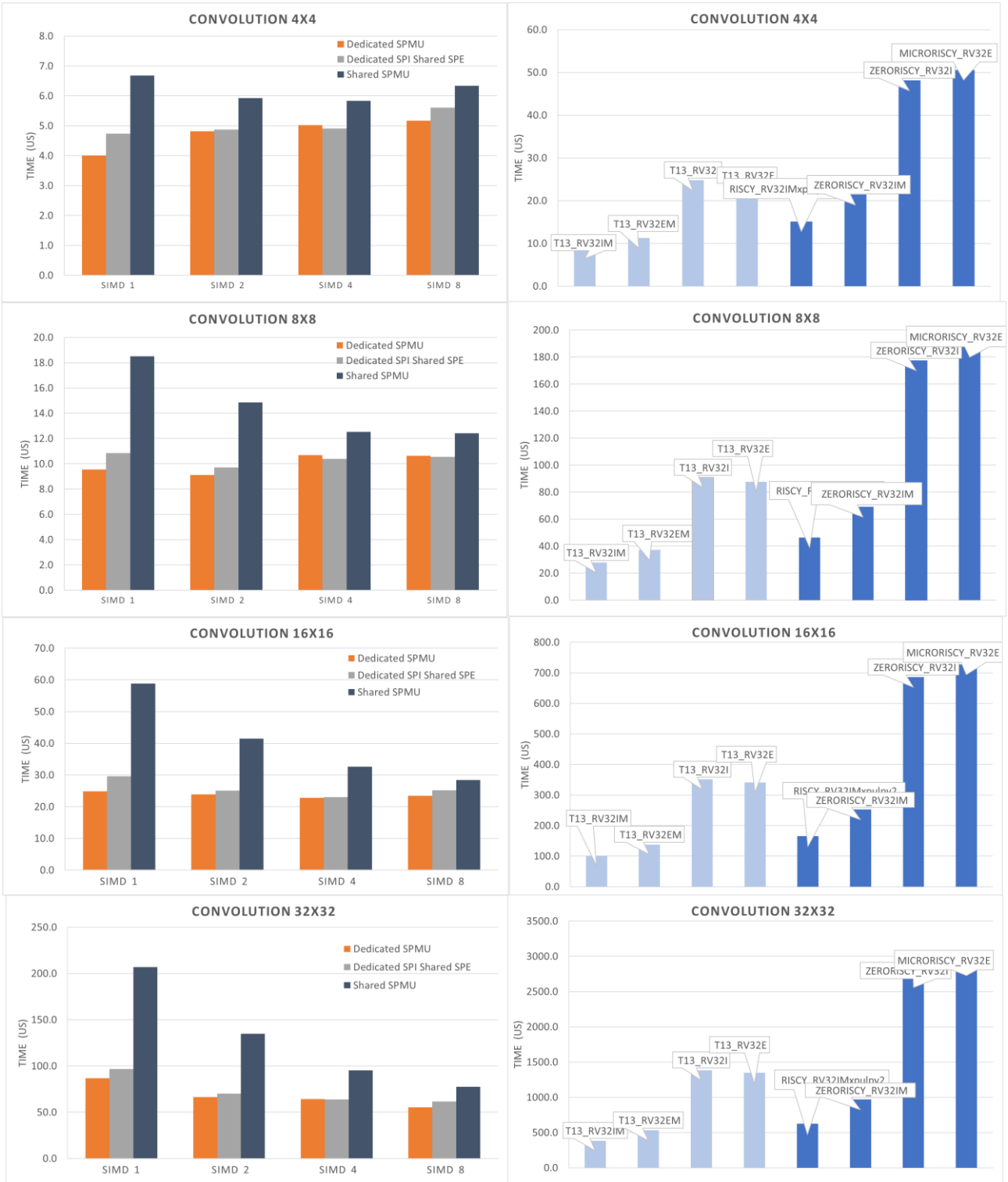


Figure.5.15. Total execution time to perform convolutions when running at the maximum attainable frequency for accelerated and non-accelerated implementations

The T13 cores highly outperformed the Riscy cores since not only do they have a good cycle count, but also attain a very high top frequency in comparison with the other cores.

- The higher cycle count comes as a result of T13 cores having zero data dependency pipeline stalls, and zero pipeline flushing, and low latency multiplication instruction.

- The high frequency is attained from pipelining and hardware simplicity.

Showing how the non-accelerated implementations of T13 outperformed the PULPino Riscy cores makes us certain that as far as CNN accelerators are concerned, it is better to use an IMT architecture over and in-order execution processor.

One final note is that also again, implementations using the embedded extension RV32E had somewhat discouraging results, which did not convince us that migrating towards an IMT architecture with a smaller set of registerfiles is better than using the normal registerfile size as defined in the RV32I ISA.

5.5.3. VGG16 Deep Convolutional Neural Networking Application

In order to further evaluate our SPMU accelerator when executing neural networking applications, we had to make the SPMU execute an entire CNN. For that, we have chosen the famous VGG16 DCNN [40]. The VGG16 test is a very successful DCNN that can achieve accuracies of up to 92.7%. It is used in many classifications [41][42][43]. The layers of the VGG16 test are showed in figure 5.15. In order to fully support the convolution layers of the VGG16, the matrix convolutions from the previous sections were combined with other libraries that performed: pre-scaling, post-scaling, add-bias, and ReLu, as well as a set of libraries for the fully-connected layers. The remaining parts of the network did not undergo acceleration (e.g. *softmax, maxpool*). After having built a unique VGG16 test to run for the various implementations of the SPMU. We have run a particular set of tests to evaluate the performance of the T13 IMT architecture. The layers in the network are shown in the image below.

Two tests are shown in figures 5.16 and 5.17. The first shows the difference in performance when running the VGG16 using one hart only, and when dividing the workload over all the harts in the core. The other compares the IMT full active harts Dedicated-SPMU versus an in-order architecture “Zeroriscy”.

The difference between the single-thread test (1 hart active), and the multi-thread test (all harts active) outlines one very important aspects in IMT architectures. First of all, both implementations interleave three harts in the core. However, the single-thread implementation shows how poorly an IMT core performs when the other harts are Idle. When all the harts become active, and the workload becomes divided among the harts, we will see a large drop in the cycle count that is evident in figure 5.16.

From the results back in the previous sub-section we chose the Dedicated-SPMU SIMD-2 as a very fast and yet most balanced option to be compared with an in-order architecture such as Zeroriscy. A few layers were developed to execute on that version of the SPMU, and they were compared with the Zeroriscy cores as show in figure 5.17.

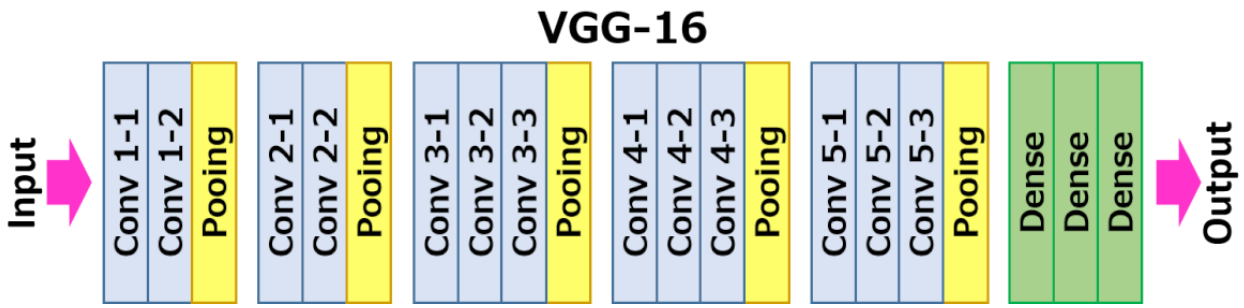


Figure.5.16. Layers of the VGG16 deep convolutional neural network

From figure 5.16 we can still affirm that when running real life applications as the VGG16 the SPMU accelerator indeed maintains it's fast trend results that were displayed back in figure 5.13.

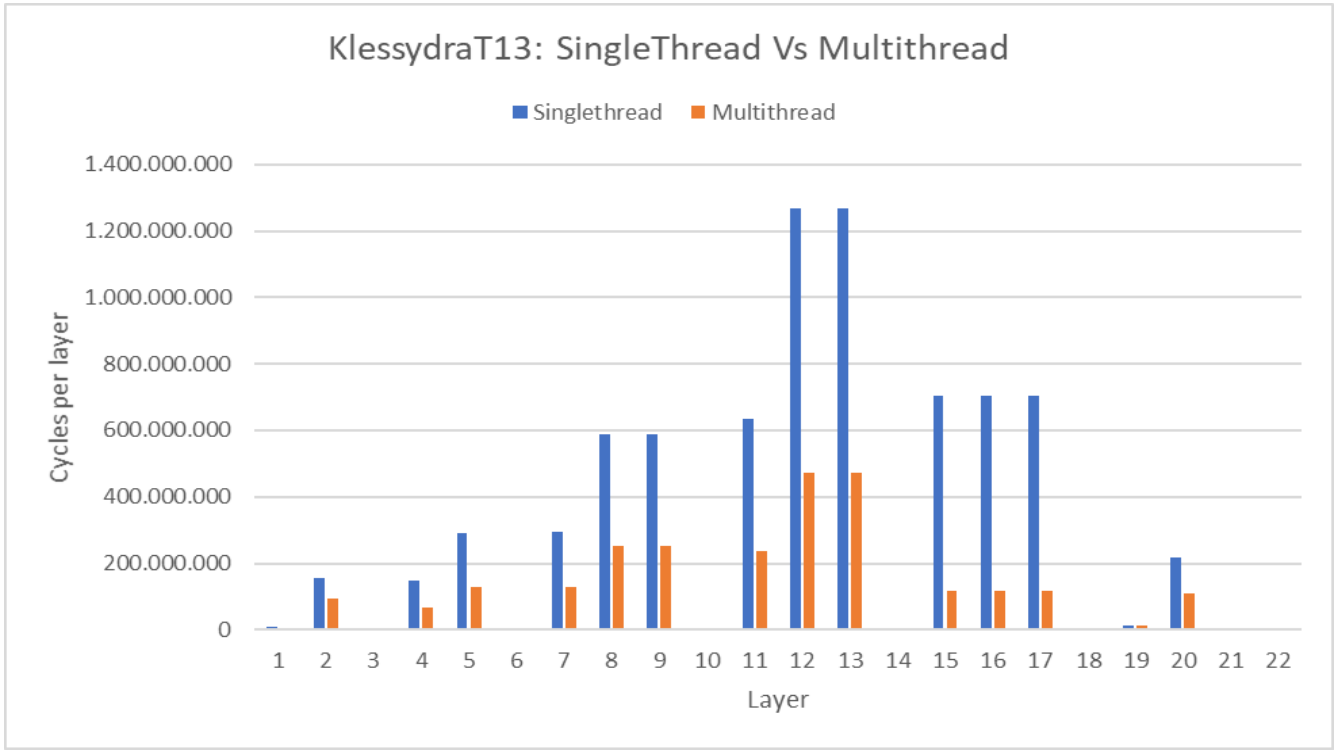


Figure.5.17. KlessydraT13 Shared-SPMU, Single Thread Vs Multithread cycle count per layer for VGG16

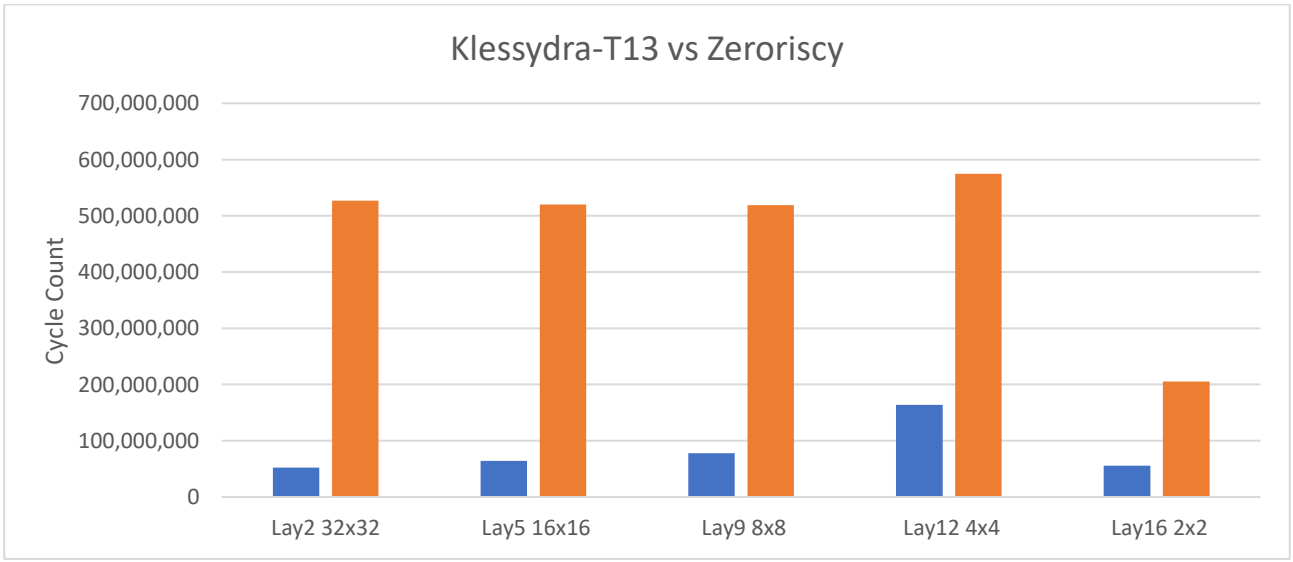


Figure.5.18. KlessydraT13 Dedicated-SPMU SIMD-2, vs Zeroriscy cycle count per layer for VGG16 execution

As a conclusion for the performance evaluation we saw the difference between an IMT core and an in-order processor. An IMT processor certainly performed better when the applications were decoupled. Synthesis results showed that IMT processors had very high top frequencies. Attaching the different SPMU schemes showed the contribution of each SPMU to the performance, and showed how DLP and TLP differently exploit the processor with small and big vector computations. Not to mention a layer of the VGG neural network were run, and they showed how the SPMU accelerator faired in real life applications.

5.6. Area, Power, and Energy Reports

5.6.1. Area Utilization

Table 5.4 reports the area utilization on the FPGA when synthesizing on the Genesys2 board [29]. We can see clearly that the area increase due to the DLP was really impacting especially in the Hybrid approaches exploiting both DLP and TLP. One small conclusion can be made here, that the speed-boost from the DLP showed in the previous section was on average smaller than the TLP speed boost, and yet the DLP exploiting schemes (Shared-SPMU SIMD-8) consumed a higher area than the TLP exploiting schemes (Dedicated-SPMU SIMD-1 and Dedicated-SPI_Shared-SPE).

An additional important note to take from these results as well is that the crossbar in the Dedicated-SPI-Shared-SPE version is large enough, such that the number of LUT utilization is very similar to that in the Dedicated-SPMU version, and that the reduction in element utilization was only in the FFs and the DSP slice count. Pipelining the crossbar to get a higher top frequency is possible, however it will increase the FF utilization in the Dedicated-SPI-Shared-SPE, and hence the FF count saved from sharing FUs will be utilized in pipelining the crossbar rendering this approach to be somewhat useless, relative to the Dedicated-SPMU approach. But still this approach can be considered as seen from the results, we save a huge number in the DSP slice count when sharing the functional units in the Dedicated-SPI-Shared-SPE approach.

Table.5.3. T13 Area Utilization on FPGA for all SPMU Configurations

Core		SIMD	Element Utilization			
			FF	LUT	BRAM	DSP
Klessydra T13	Shared SPMU	1	6552	10655	6	8
		2	6907	12835	6	12
		4	7587	15807	6	20
		8	9064	21423	12	36
	Dedicated SPMU	1	7782	14344	18	16
		2	8875	13017	18	28
		4	10903	28309	18	52
		8	15223	46861	36	100
	Dedicated SPI Shared SPE	1	7234	14229	18	9
		2	8009	18803	18	12
		4	9167	27150	18	20
		8	11460	48081	36	36
	NO_ACCL (RV32IM)	NA	5639	7975	0	4
	NO_ACCL_(RV32EM)	NA	4165	8120	0	4
	NO_ACCL (RV32I)	NA	5424	7674	0	0
	NO_ACCL_ (RV32E)	NA	3890	7414	0	0
RISCY		NA	2527	7674	0	6
ZeroRiscy		NA	1933	3275	0	1
ZeroRiscy (no RV32M)		NA	1791	2832	0	0
MicroRiscy		NA	1279	2434	0	0

Making the Comparison between Riscy, Zeroriscy cores and the T13 non accelerated cores. We definitely see a larger area occupation in the T13 non accelerated cores. Thar is for the obvious reason that in order for the T13 core to be an IMT architecture, we had to replicate the registerfile, the CSR unit and the program counter. One thing to consider in order to decrease overhead that IMT architectures have, is by disabling the performance counters in the CSR unit. Doing that saved us approximately 1200 LUTs from the LUT count listed above. The other thing is to use the embedded extension RV32E which halves the size of the registerfile. However, we saw how that terribly affected the performance, and thus the tradeoff of the registerfile area with performance is a favorable step in this case.

5.6.2. Dynamic Power Consumption and Energy Efficiency

The average dynamic power consumption is reported in figure 5.18 for running the convolutions on each hardware scheme. Obviously, the power consumption increases as the area gets bigger, but the curve rises up very sharply for the SIMD 8 configurations. Deep SIMD configurations proved to be less power efficient in this manner (especially in FPGA synthesis) as they consume a lot of power particularly in the hardware schemes exploiting the TLP. SIMD 2 configurations for all hardware schemes showed only a slight increase in dynamic power consumption in one hand, and a greater increase in performance on the other hand, making it desirable to be considered as a balanced approach.

Other than the small area footprint of the Riscy cores, they also all consumed less dynamic power than the T13 non accelerated cores. The RV32E extensions seemed to have larger drops in the dynamic power consumption as well.

The static power was not mentioned, since for FPGAs the static power does not change based on the area utilization of the FPGA, but rather it depends on the technology of the FPGA itself.

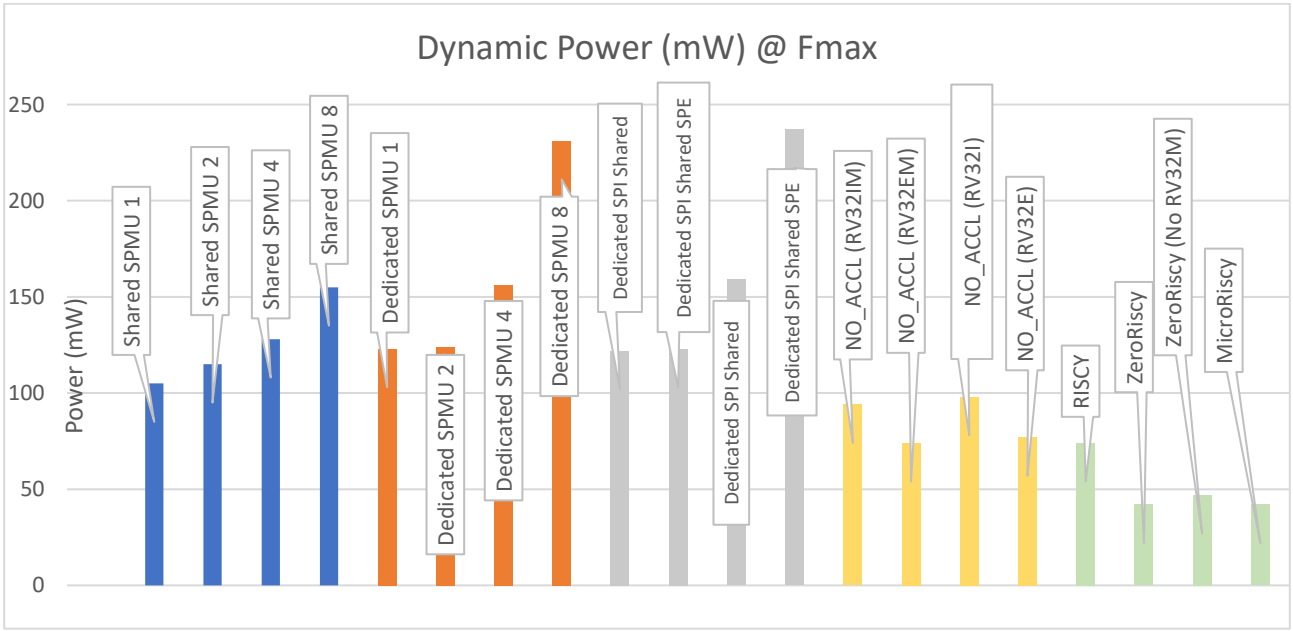


Figure.5.19. Dynamic Power Consumption of the T13 core running 32x32 convolutions

Figure 5.19 shows the total energy consumption for running the different convolutions. They were again divided into two sides. The left sides for the accelerators, and the right side for the non-accelerators. They were separated in since the non-accelerated had very high energy consumption compared to the accelerated counterparts, and thus if placed together, the non-accelerated energy results would have saturated the improvements between the different schemes in the accelerated results.

Figure.5.20. Energy Consumption for running each implementation at the top frequency on the different convolution sizes

Many conclusions can be made from these results. First, we show that not only using the SPMU accelerator generates high speed results, but it is also more energy efficient, than not using the SPMU accelerator.

Second, compare the SPMU accelerators, we can see that the Shared-SPMU has the worst results, and that both the TLP exploiting approaches gave much better results than the Shared-SPMU.

Third, the results comparing the Dedicated-SPMU to the Dedicated-SPI_Shared-SPE approach showed almost an overlap in the energy consumption just like the overlap they in the performance. This is very good since we showed that very little trade-off in the performance and energy consumption can be substituted with a large chunk of area and that is by sharing the SIMD functional units.

Finally comparing the non-accelerated implementations together, we see that the T13 slightly less energy efficient then both Riscy and Zeroriscy. Zeroriscy has a very low dynamic power count, while Riscy has a low cycle count, both contributed heavily to the energy efficiency.

5.7. Further Evaluations (memory test, GCC optimizations)

A few additional tests were performed to see the consistency of the performance using GCC optimization flag "-O2". Figure 5.23 shows the cycle count to perform vector addition when compiling the C tests without enabling any GCC optimizations. While figure 5.20 shows the same results but with GCC optimizations enabled.

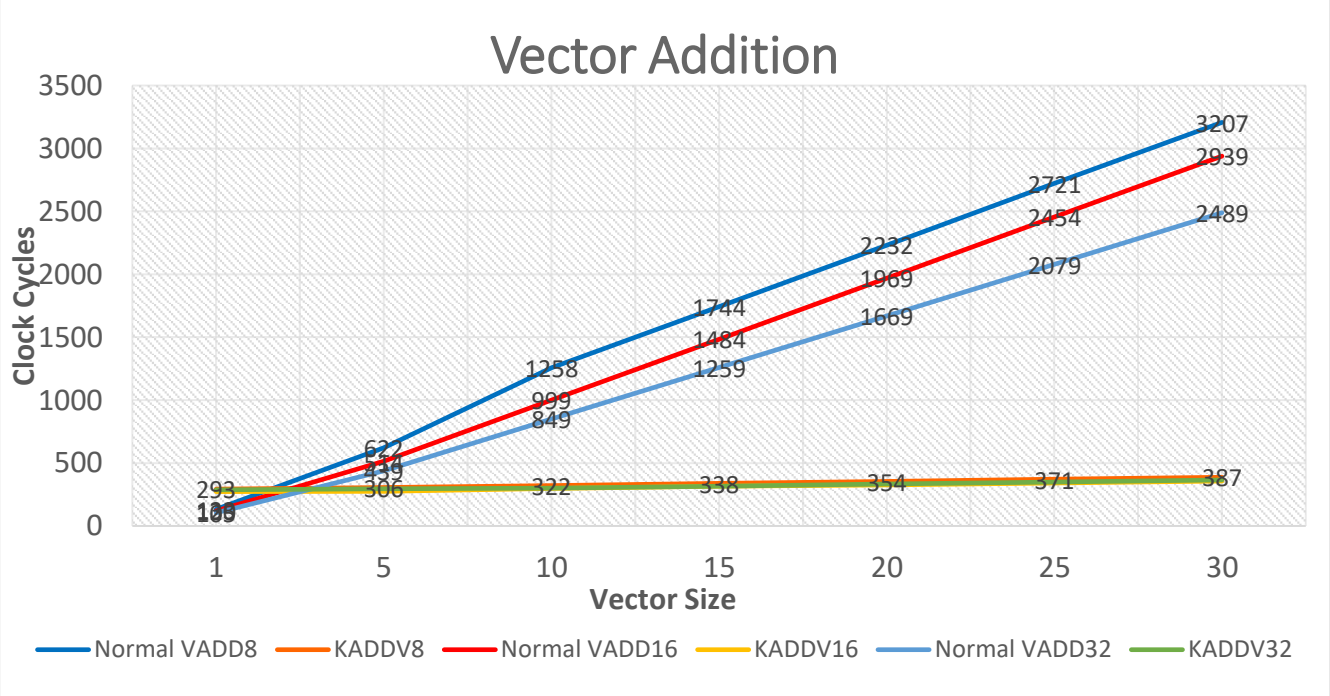


Figure.5.21. Vector addition C test performed with GCC optimizations disabled

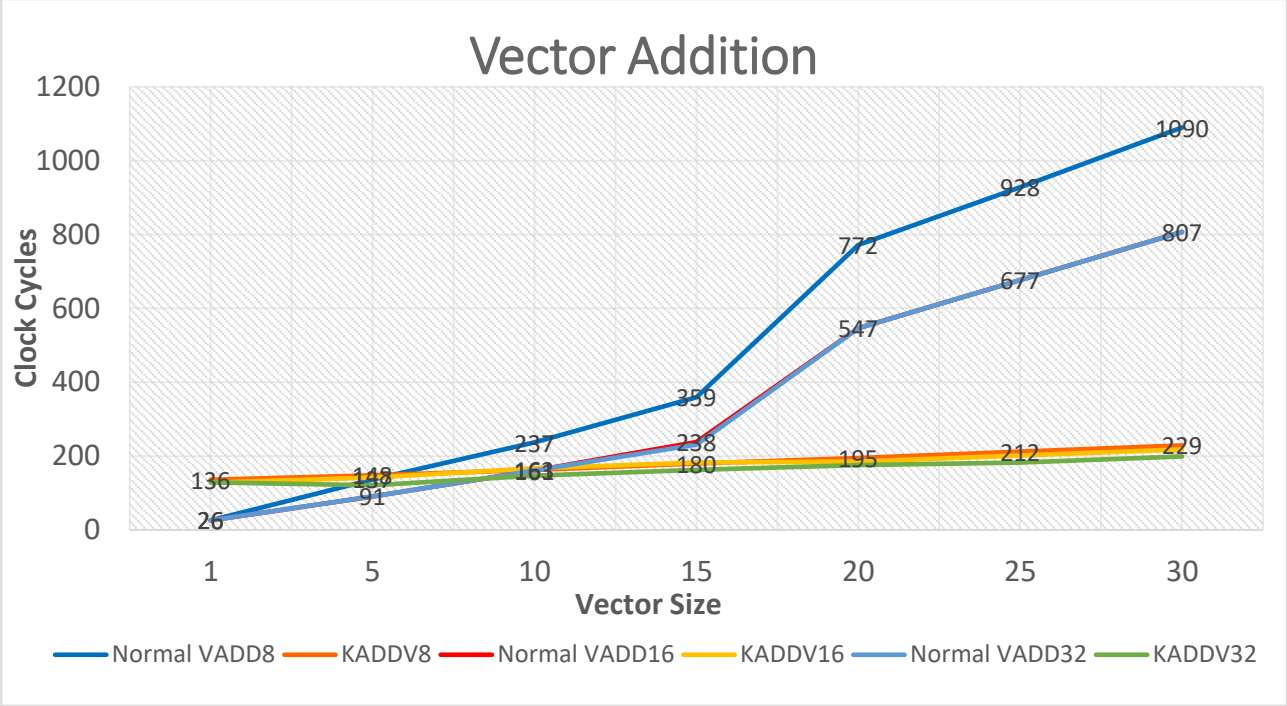


Figure.5.22. Vector addition C test performed with GCC optimizations enabled

From the results above it shows that disabling the GCC optimizations affected performance in both operations. However, for the operations using the accelerator, we have a cycle count increase that is a constant offset, while in the non-accelerated vector addition operation, the cycle count increment was a variable offset such that when the vector size, grows, the offset grows linearly as well.

Another evaluation was made to show the memory impact of doing two equal operations (table 5.5). The first operation does not use the SPMU accelerator. The second performs the same operation, but using the SPMU. In the operations using the SPMU, there are two memory tests that were made, the first one does all the SPMU operations in a single function call, while the other one does the same operations in a multi-function call.

Table.5.4. Size in Bytes of the program memory and data memory for different tests

Size (Bytes)												
Vector Size	Normal Addition Test				SPMU Single Funct Call Test				SPMU Multi Funct Call Test			
	With GCC Optimization		Without GCC Optimization		With GCC Optimization		Without GCC Optimization		With GCC Optimization		Without GCC Optimization	
	Program mem size	Program mem size	Data mem size	Program mem size	Data mem size	Program mem size	Data mem size	Program mem size	Data mem size	Program mem size	Data mem size	Program mem size
1	1326	3059	1300	3533	1378	3477	1352	3705	1378	3230	1352	3591
10	2730	3211	2704	3572	2782	3477	2756	3705	2782	3230	2756	3591
20	4290	3211	4264	3572	4342	3477	4316	3705	4342	3230	4316	3591

The results from the memory tests, shows that also using the SPMU does not impact the memory size, the results are similar to the non-SPMU test. For the data memory, the only impact on the memory size was from increasing the vector size, but regardless whether we use the SPMU or not.

Chapter 6 C Language Software Suite

This chapter shows the implementation of the software suite used in benchmarking the T13 microprocessor. All the tests were written in C and compiled by a patched RISCV-GCC compiler. The first section shows the instruction level testing of the custom SPMU instructions. The second section shows how the custom instructions were used to make convolutions. The third section mentions the additional libraries needed in order to accelerate the convolution and fully-connected layers of the VGG16 DCNN application.

6.1. Instruction level testing:

For every custom instruction in the SPMU, a C test has been made to detect whether the SPMU executes its instructions correctly. All the tests check whether the SPMU outputs match the non-SPMU, and benchmark the performance of the SPMU for all data types (8, 16, and 32).

The example test shown in the code below takes the number of elements inside each vector, and the time variable, and tries to randomize the data with the *rand* function. The test sets the MVTYPE and then calls a C function that uses all the harts in the core to load the vectors and compute the results. The cycle count to perform the arithmetic operation is counted, and saved. The output results are checked to be correct, and then performance is compared to the non-accelerated tests.

The code below shows how vector addition instruction KADDV is tested for 32-bit data types. Other data types and instructions are not shown because of the repetitiveness of the code sequence. There implementation can be inferred just by looking at this one.

```
1  /* ------------------------------------------------------------ KADDV Test ------------------------------------------------------------*/
2  #define NumOfThreads 3
3  #define NumOfElements 50
4  #define TIME 10
5
6  int32_t vect32_1[NumOfElements], vect32_2[NumOfElements];
7  int32_t testres32[NumOfElements];
8  int32_t *res32;
9  int32_t result32[NumOfElements];
10 int   size32=NumOfElements*sizeof(int);
11 int   testperf, perf32[NumOfThreads];
12
13 int main() {
14         srand(TIME);
15         for (int i=0; i<NumOfElements; i++) {
16                 vect32_1[i] = rand()  % (0x80000000 - 0x1) +1;
17                 vect32_2[i] = rand()  % (0x80000000 - 0x1) +1;
18         }
19         int add_pass = 0;
20         int perf = 0;
21         int* ptr_perf = &perf;
22
23         /* 32-bit KADDV here */
24         VECT_ADD_32:
25         sync_barrier();
26         // ENABLE COUNTING ------------------------------------------------------------------------
27         __asm__("csrrw zero, 0x7A0, 0x00000001");
28         //-------------------------------------------------------------------------------------
29         // SET MVTYPE -----------------------------------------------------------------------
30         __asm__("csrrw zero, mvtype, 0x00000002"); // set the data type to 32-bits
31         //-------------------------------------------------------------------------------------
```

```
32
33          // TEST KADDV(32)-------------------------------------------------------------------
34          /* call the function that perfroms the KADDV operation
35          res32=kless_vector_addition_mth((void*) result32, (void*) vect32_1, (void*) vect32_2, size32);
36          //-------------------------------------------------------------------------------------
37          // DISABLE COUNTING AND SAVE MCYCLE OF EACH THREAD ------------
38          __asm__("csrrw zero, 0x7A0, 0x00000000;"
39                          "csrrw %[perf], mcycle, zero;"
40                          "sw %[perf], 0(%[ptr_perf]);"
41                          :
42                          :[perf] "r" (perf), [ptr_perf] "r" (ptr_perf)
43                          );
44          if (Klessydra_get_coreID()==0) perf32[0]=perf;  // store the cycle count of thread 2
45          if (Klessydra_get_coreID()==1) perf32[1]=perf;  // store the cycle count of thread 1
46          if (Klessydra_get_coreID()==2) perf32[2]=perf;  // store the cycle count of thread 0
47          //-------------------------------------------------------------------------------------
48
49          // Test 32-bit addition result ----------------------------------------------------------
50          if (Klessydra_get_coreID()==1){
51                  __asm__( "csrrw zero, 0x7A0, 0x00000001;");  // enable counting
52                  for (int i=0; i<NumOfElements; i++){
53                          testres32[i] = vect32_1[i]+vect32_2[i]; // perform the addition without acceleration
54                  }
55                  __asm__("csrrw zero, 0x7A0, 0x00000000;"  // disable counting and save the cycle count
56                          "csrrw %[perf], mcycle, zero;"
57                          "sw %[perf], 0(%[ptr_perf]);"
58                          :
59                          :[perf] "r" (perf), [ptr_perf] "r" (ptr_perf)
60                          );
61                  testperf = perf;
62                  for (int i=0; i<NumOfElements; i++){
63                          if (res32[i]==testres32[i]) // check every element{
64                                  add_pass++;
65                          }
66                          else {
67                                  goto FAIL_VECT_ADD_32; // if an error is encountered goto the error label
68                          }
69                  }
70                  if (add_pass==NumOfElements){
71                          printf("\nPASSED KADDV32 32-bit vector addition"); // all outputs are correct print pass
72                  }
73          }
74          if (Klessydra_get_coreID()==1){
75                  printf("\n\nNumber of Elements:%d\n",NumOfElements);
76                  for(int i=0;i<3;i++){
77                          printf("Th%d KADDV32 Speed: %d Cycles\n",i, perf32[i]);  // print cycle count of SPMU
78                  }
79                  printf("ADDV32 Speed: %d Cycles\n", testperf);  // print the cycle count and end the program
80                  return 0;
81          }
82          __asm__("csrrw zero, mstatus, 8;" "wfi;");  // stall the harts that finish
83          // ----- Fail Section ----------------------------------------------------------------
84          FAIL_VECT_ADD_32:  // error label
85          printf("\nFAILED KADDV32 32-bit vector addition\n"); // print fail
86          return 1;
```

The function "*kless_vector_addition_mth"* performs the KADDV using all the harts in the T13 core, as seen in the code below. The first thread that enters does a vector load *vs1* atomically, and then exits the routine. The second hart atomically loads the second vector *vs2* to the SPMs and exits

the function. The third hart performs the vector addition, stores the result back in main mem, then exits the function.

```
1  void* kless_vector_addition_mth(void *result, void* src1, void* src2, int size){
2          int SPMADDRA = spmaddrA; // base address of spmA
3          int SPMADDRB = spmaddrB; // base address of spmB
4          int SPMADDRC = spmaddrC; // base address of spmC
5          int key = 1; // the key locks some routines from being executed
6          static int section1 = 0;
7          static int section2 = 0;
8          int* psection1 = &section1;
9          int* psection2 = &section2;
10         asm volatile(
11                 "amoswap.w.aq %[key], %[key], (%[psection1]);"
12                 "bnez %[key], SCP_copyin_vect_2;"
13                 "SCP_copyin_vect_1:"
14                 "          kmemld %[SPMADDRA], %[srcA], %[sz];" // load vector vs1
15                 "          j END;"
16                 "SCP_copyin_vect_2:"
17                 "          amoswap.w.aq %[key], %[key], (%[psection2]);"
18                 "          bnez %[key], END;"
19                 "          kmemld %[SPMADDRB], %[srcB], %[sz];" // load vector vs2
20                 "          csrw 0xBF0, %[sz]; " // set the vector size
21                 "          kaddv %[SPMADDRC], %[SPMADDRA], %[SPMADDRB];" // KADDV operation
22                 "          kmemstr %[result], %[SPMADDRC], %[sz];" // store back the result in memory
23                 "END:"
24                 :
25                 :[key] "r" (key),[psection1] "r" (psection1),
26                 [psection2] "r" (psection2),  [sz] "r" (size),
27                 [SPMADDRA] "r" (SPMADDRA), [srcA] "r" (src1),
28                 [SPMADDRB] "r" (SPMADDRB), [srcB] "r" (src2),
29                 [SPMADDRC] "r" (SPMADDRC), [result] "r" (result)
30         );
31         return result;
32 }
```

Another function that does the above routine with a single thread only is shown below.

```
1
1  void* kless_vector_addition_sth(void *result, void* src1, void* src2, int size){
2          int SPMADDRA = spmaddrA; // base address of spmA
3          int SPMADDRB = spmaddrB; // base address of spmB
4          int SPMADDRC = spmaddrC; // base address of spmC
5          asm volatile(
6                  "          kmemld %[SPMADDRA], %[srcA], %[sz];" // load vector vs1
7                  "          kmemld %[SPMADDRB], %[srcB], %[sz];" // load vector vs2
8                  "          csrw 0xBF0, %[sz]; " // set the vector size
9                  "          kaddv %[SPMADDRC], %[SPMADDRA], %[SPMADDRB];" // KADDV operation
10                 "          kmemstr %[result], %[SPMADDRC], %[sz];" // store back the result in memory
11                 "END:"
12                 :
13                 :[key] "r" (key), [sz] "r" (size),
14                  [SPMADDRA] "r" (SPMADDRA), [srcA] "r" (src1),
15                  [SPMADDRB] "r" (SPMADDRB), [srcB] "r" (src2),
16                  [SPMADDRC] "r" (SPMADDRC), [result] "r" (result)
17         );
18         return result;
19 }
```

An additional function in the SPMU libraries was created to benchmark the speed of the hardware loops, and that is by executing the SPMU instructions continuously inside a sw-loop (*for loop*), then the output is compared. The body of that function is shown below.

```
1  void* kless_vector_addition_sth_sw_loop(void *result, void* src1, void* src2, int size, int SIMD_BYTES){
2          int SPMADDRA = spmaddrA; // base address of spmA
3          int SPMADDRB = spmaddrB; // base address of spmB
4          int SPMADDRC = spmaddrC; // base address of spmC
5          int size_temp = size;
6          asm volatile(
7                  "       kmemld %[SPMADDRA], %[srcA], %[size_temp];"   // load vector vs1
8                  "       kmemld %[SPMADDRB], %[srcB], %[size_temp];"   //  load vector vs2
9                  "       csrw 0xBF0, %[SIMD_BYTES];"  // set the vector size
10                 :[size_temp] "r" (size_temp), [SIMD_BYTES] "r" (SIMD_BYTES),
11                  [SPMADDRA] "r" (SPMADDRA), [srcA] "r" (src1),
12                  [SPMADDRB] "r" (SPMADDRB), [srcB] "r" (src2)
13         );
14         for (int i=0; i<size; i=i+SIMD_BYTES){ // loop through the vector elements
15                 if (size-i >= SIMD_BYTES){
16                         size = size-i;  // decrement the vector size
17                         asm volatile(
18                            "       kaddv %[SPMADDRC], %[SPMADDRA], %[SPMADDRB];"// KADDV operation
19                            : [SPMADDRA] "r" (SPMADDRA),
20                              [SPMADDRB] "r" (SPMADDRB),
21                              [SPMADDRC] "r" (SPMADDRC)
22                         );
23                         SPMADDRA+=SIMD_BYTES;  // increment source A pointer
24                         SPMADDRB+=SIMD_BYTES;  // increment source B pointer
25                         SPMADDRC+=SIMD_BYTES;  // increment the destination pointer
26                 }
27                 else {
28                 /* if there is no need to loop anymore, then re-write the vector size and execute the last SPM line */
29                         size = i;
30                         asm volatile(
31                                 "       csrw 0xBF0, %[size];"
32                                 "       kaddv %[SPMADDRC], %[SPMADDRA], %[SPMADDRB];"
33                                 : [SPMADDRA] "r" (SPMADDRA) ,
34                                   [SPMADDRB] "r" (SPMADDRB) ,
35                                   [SPMADDRC] "r" (SPMADDRC) ,
36                                   [size] "r" (size)
37                         );
38                 }
39         }
40         SPMADDRC=spmaddrC;
41         asm volatile(
42                 "       kmemstr %[result], %[SPMADDRC], %[size_temp];"
43                 :[size_temp] "r" (size_temp), [SIMD_BYTES] "r" (SIMD_BYTES),
44                 [SPMADDRC] "r" (SPMADDRC), [result] "r" (result)
45         );
46         return result;
47 }
```

6.2. Convolution tests:

The convolution test comes with a set of functions called convolution2D (*conv2D* for short). In order to fully explain the algorithm of the conv2D functions we will demonstrate how a convolution is performed the conventional way, and how the algorithm was transformed to fit on the SPMs.

6.2.1. Convolutions (traditional method)

The convolutions in neural networks are performed by sliding the kernel map from its central point over the entire pixels of the feature map otherwise known as the input matrix. The kernel maps in our convolutions have their dimensions set to 3x3 (i.e. like VGG16 filters). Consider the convolution of this kernel with a 4x4 feature map as shown in figure 6.1.

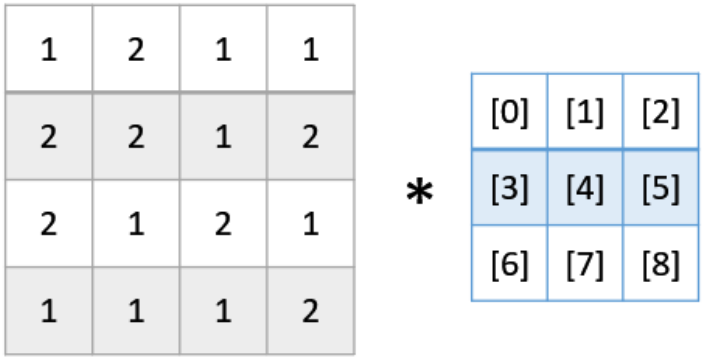


Figure.6.1. Convolution of feature map on the left and kernel map on the right

When the kernel map starts sliding over the feature map starting from the top left corner. There will be elements of the kernel map not overlapping any elements of the feature map. In order to overcome this, feature map is padded with zeros around its entire parameter such that when the kernel map slides, its elements will either be overlapping the feature map or the padded-zeros as seen in figure 6.2.

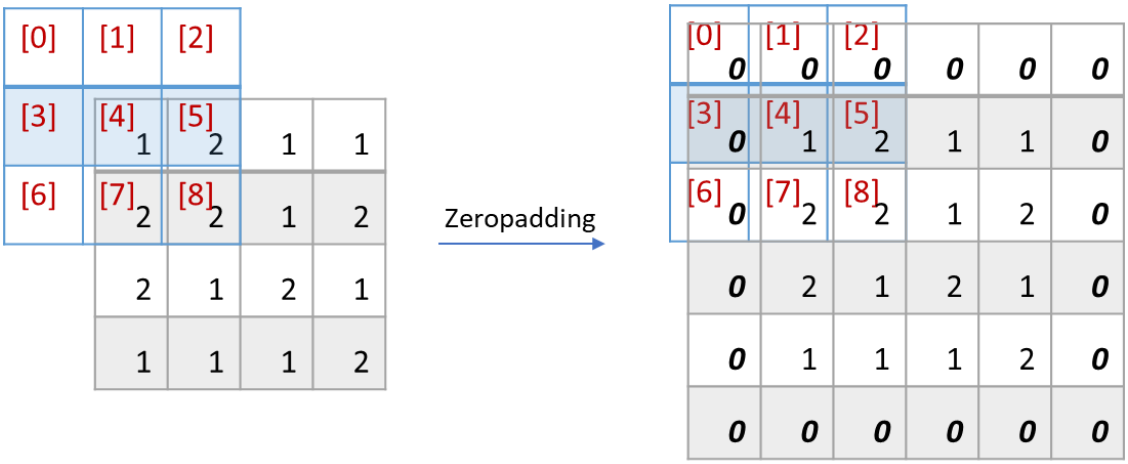


Figure.6.2. Convolution of feature map on the left and kernel map on the right

One convolution gives one output pixel result for the output map. When the kernel has passed over the entire feature map and produced all the output pixels, the convolution2D would be considered at this point done.

6.2.2. Convolutions (sub-kernel method):

One drawback of the traditional method of performing a matrix convolution was the augmentation of zero-paddings to the whole parameter of the feature maps. It presented a few challenges for doing that method, such as:

- High memory consumption, for example a 32x32 matrix of integers that will be zero-padded cannot fit on a 4KB scratchpad memory, it needs an extra 528 Bytes of memory space to fit, which is about 12.5% the size of the original matrix.

- We also have slower memory for ASIC implementations, since FPGAs have fixed size BRAMs [44] so this might only affect the FF or LUT based memories. However, for ASIC zero-padding will require an 8KB memory for a 4KB feature map, and a 4KB memory for 2KB feature map and etc. Bigger memories are usually slower than smaller memories, or have higher latencies.

There was the need to re-write the conv2D function in order to avoid zero-padding. The key idea was to divide the conv2D function into separate functions each would perform a set of convolutions with sub-kernels on the different regions of the feature map as shown in figure 6.3. We will demonstrate how the convolution with sub-kernel F was performed. Other sub-kernel implementations follow a similar pattern and thus will not be elaborated.

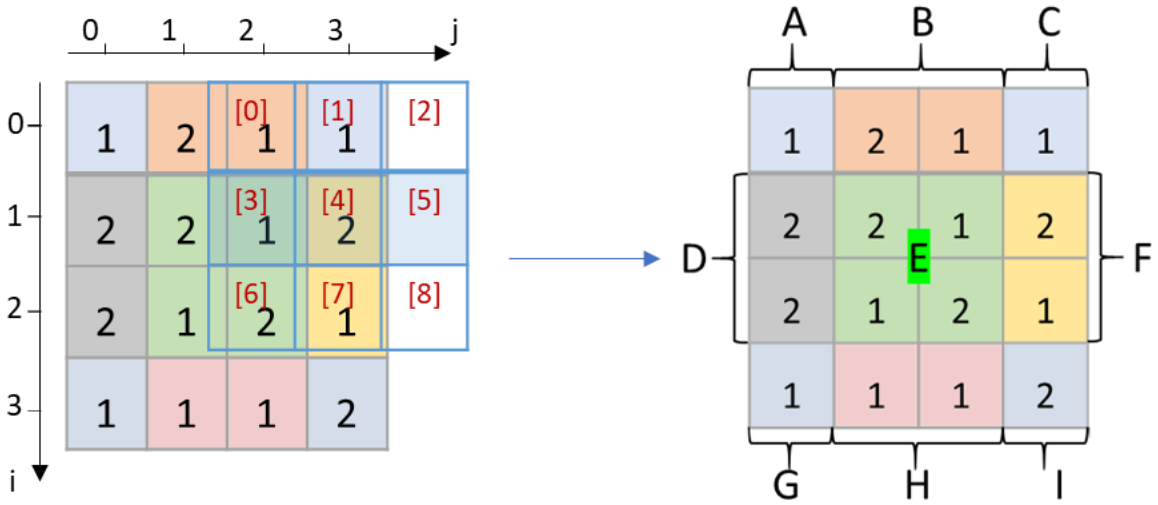


Figure.6.3. Division of the sub-kernels. On the left shows the overlap with sub-kernel F

The sub-kernels include only the overlapping parts between the kernels and the feature maps. In the above figure with the 4x4 matrix, we can see some regions. Each region has a different part of the kernel map overlapping. Thus, performing a convolution would require calling nine functions each performing the routine with a different sub-kernel.

The functions are divided into four groups. The first group is when the kernel centroid lands on the edges, we perform the A-C-G-I routines. Sliding the centroid in between the corners on the first and last row uses the B-H group. Likewise, sliding in between columns we use D-F groups. When the sub-kernel is fully overlapping the feature map, the operations will belong to group E, and the convolution will be the default case.

Considering the convolution with sub-kernel F, the output pixel is calculated as follows:

$$\boldsymbol{output\ pixel} \mathrel{+}= [\mathbf{0}] * \mathbf{1} + [\mathbf{1}] * \mathbf{1} + [\mathbf{3}] * \mathbf{1} + [\mathbf{4}] * \mathbf{2} + [\mathbf{6}] * \mathbf{2} + [\mathbf{7}] * \mathbf{1}$$

The presence of the "+= "sign is because the convolutions always accumulate the output pixel. In addition, since our convolutions are performed with a fixed-point implementation, the outputs need to be post scaled. Hence the equation would actually look like this.

$$\boldsymbol{output\ pixel} \mathrel{+}= ([\mathbf{0}] * \mathbf{1}) \gg \boldsymbol{ps} + ([\mathbf{1}] * \mathbf{1}) \gg \boldsymbol{ps} + ([\mathbf{3}] * \mathbf{1}) \gg \boldsymbol{ps} + ([\mathbf{4}] * \mathbf{2}) \gg \boldsymbol{ps} + ([\mathbf{6}] * \mathbf{2}) \gg \boldsymbol{ps} + ([\mathbf{7}] * \mathbf{1}) \gg \boldsymbol{ps}$$

The snippet of the code in figure 6.4 shows how to perform the convolution with sub-kernel F using the SPMU instructions.

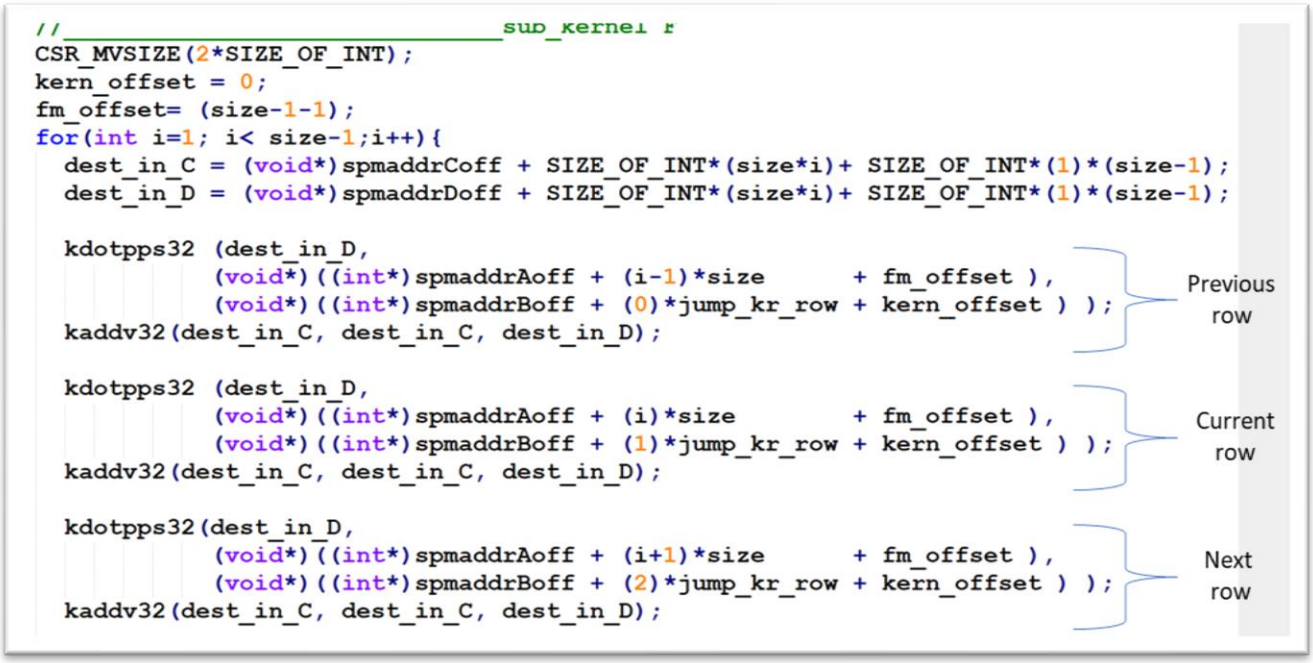


Figure.6.4. Sub-Kernel F executed in the SPMU

As figure 6.3 suggests, when the centroids overlap the element (1,3), three different rows of two integers are highlighted, hence the vector length of 2.

6.2.3. Choosing the best convolutions algorithm:

Although the sub-kernel method had the memory advantage over the zero padded method. However, it suffered in the cycle time as it did not actually exploit the SIMD nature of the SPMU very well. While the zero-padded implementation while still consuming bigger memory, but nonetheless exploited very well the SIMD implementation in the SPMU, but it suffered with the memory loads as it was loading a bunch of zeros.

So instead of doing one burst load for the entire matrix with a "*kmemld*" instruction, we found that the optimal solution was to use the zero-padded method with a set of burst loads that loads the discrete data lines in the matrix without the padded zeros. This in turn will relieve the overhead of doing unnecessary memory transfers of zeros. The data lines will be separated by the offset of zeros that separate them. Figure 6.5 shows how it is done.

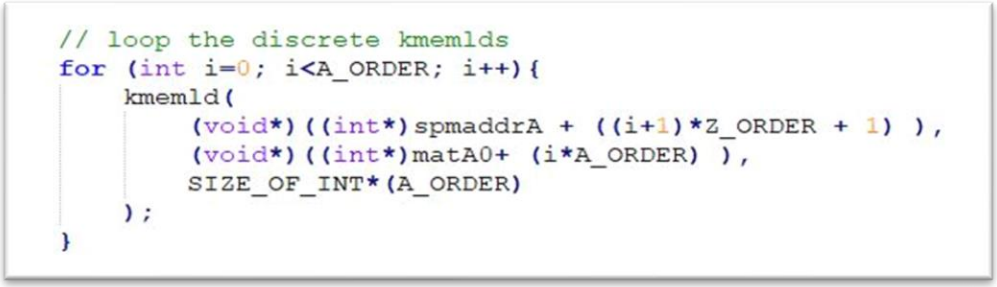


Figure.6.5. Discrete Kmemlds for zeropadded implementations

Figure 6.6 shows how the zero-padded convolutions are done using the SPMU instructions.

```
for(int i=1; i< size-1;i++)
{
    k_element=0;
    for (int rw_pt=-1; rw_pt<2; rw_pt++)
    //rw_pt is an index i use to point to the correct row, regarding this loop that is executed three time
    //instead of making 9 different ksvmulrf
    {
    ksvmulsc((void*)(   (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
              (void*)   ( (int*)spmaddrAoff + (i+rw_pt)*size    +0 ),
              (void*)   ( (int*)spmaddrBoff+k_element++) );

    ksrav((void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (int*)conv2D_out_scal);

    kaddv ((void*)( (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT);

    ksvmulsc((void*)(   (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
              (void*)   ( (int*)spmaddrAoff + (i+rw_pt)*size    +1 ),
              (void*)   ( (int*)spmaddrBoff+k_element++) );

    ksrav((void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (int*)conv2D_out_scal);

    kaddv ((void*)( (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT);

    ksvmulsc((void*)(   (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
              (void*)   ( (int*)spmaddrAoff + (i+rw_pt)*size    +2 ),
              (void*)   ( (int*)spmaddrBoff+k_element++) );

    ksrav((void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (int*)conv2D_out_scal);

    kaddv ((void*)( (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrCoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT,
          (void*)(  (int*)  (spmaddrDoff)   ) + SIZE_OF_INT*(size*i)+1*SIZE_OF_INT);
    }
}
// CSR_MVSIZE(size*size*SIZE_OF_INT);
ksvmulrf((void*)spmaddrDoff,(void*)spmaddrDoff,(void*)zero);
```

Figure.6.6. Zero-Padded Convolution method using the SPMU instructions

6.3. Supplementary VGG16 libraries

Having built libraries capable of doing the matrix convolutions, there was still the need to supplement the VGG16 libraries with a few more functions in order to have it ready to accelerate the network.

First, AddBias and ReLu operations are functions were made Adding the bias to the output matrix is done with the following function:

"*ksvaddsc_v2 (dest, source1, source2, size);"*

The function above sets the MVSIZE CSR to be equal to *size,* and calls the *ksvaddsc* SPMU instruction which adds the vector *source1* with the scalar *source2,* and stores the result in *dest*.

The operation is followed by calling a ReLu function that rectifies all the negative values.

"krelu((void)dest, (void*)source);"*

The function above rectifies the source vector in *source*, and places the output vector in *dest*.

What remains after this is the fully connected layer which can be simply implemented by one instruction called '*kdotpps'*.

Operations in VGG16 not handled by the SPMU are the following:

- Maxpool layer haves the sizes of its input matrices by pooling the maximum value in 2x2 filter that slides vertically and horizontally across the input matrix.

- As for the last part, layer_22 is implemented using the *softmax()* function, which implements the *non-linear* function *softmax* for producing the probability distribution of all the possible outcomes.

With this, the libraries have become complete and can be used to accelerate the VGG16. The performance results were already reported in chapter 5.

Conclusions

In this thesis we introduce the Klessydra-T branch of the Klessydra family of microprocessors. The Klessydra cores fully support RISC-V instruction set in 32-bit. The Klessydra-T cores support the base integer instructions “I”, the atomic extensions “A”, the multiplication/division extension “M”. The T1 sub-branch of the Klessydra-T further appends to the native RISC-V ISA a set custom specialized instruction for accelerating convolutional neural networking applications. The motivation behind forming the Klessydra-T branch was to exploit IoT embedded systems in order to obtain higher energy efficiency and performance, and the motivation behind adding a hardware accelerator in the T1 was in order to allow an easy migration of CNN towards embedded systems.

Our study started by determining the optimal pipeline organization in interleaved multithreaded processors by performing and experimental assessment , and we showcased that pipelining the core has consistently improved the performance, while interleaved multithreading maintained the core in having zero delay slots, thus improving both the overall performance, and the energy efficiency required to execute a single instruction.

We further described in an analytical assessment that deeper pipelines between registerfile read and write ports are unfavorable (e.g. T04, T05, etc.), since the critical path would improve only slightly in soft core implementations due to the growth of the net delay between FPGA elements. While the area would still continue to grow linearly with every new hart. Also, we mentioned that the cycle count would become worse when executing practical applications in these deeply pipelined IMT architectures, such that overall performance will degrade in the sequential single hart applications, or in parallel tightly coupled applications that require constant thread synchronizations.

Also, in another analytical assessment, we saw that introducing pipelines before the registerfile read ports does not increase the performance, but rather degrades it, since it will require that the IMT core implements instruction flushing logic in which it was not needed previously. Thus, re-introducing the branch delay slots.

The spectrum of target applications covered in our earlier assessments, showed that the number of applications that can be exploited by the IMT approach were only a small portion of the entire spectrum. So, we attempted to develop an IMT processor coupled with a hardware accelerator that can exploit more target applications. And since neural networks were becoming a hot topic in embedded systems. This in hand drove us to develop a neural network accelerator called the SPMU.

In our basic evaluations of the SPMU, we saw the significance of the performance contributions in cycle count of both the low latency scratchpad memories, and the hardware loops (zero overhead loops) to the overall performance of the SPMU when executing different vector sizes. Further evaluations continued to test the cycle count improvement in increasing the data level parallelism for small and large vectors. We determined that data level parallelism can improve the cycle count greatly in large vectors and only slightly in small vector because of the T13 core’s ability to hide the latency of the SPMU instructions almost completely when the vectors are small, and only moderately if the vectors were large.

Two more complex SPMU hardware schemes were employed. These two schemes exploited the instruction level parallelism through increasing the thread level parallelism. The first scheme sets dedicated memories subsystems (SPI) for every hart, and dedicated functional units (SPE) as well. While the other scheme employs dedicated memories (SPI) for every hart, but a shared set of functional units (SPE) to be used by all the harts. Both approaches decreased the cycle count even

further then the basic Shared-SPMU approach. The Dedicated-SPMU scheme got slower because of the large area overhead and the increase in the net delay, while the scheme containing shared functional units suffered in the top operating frequency because the crossbar connecting the SPI memories to the shared SPE functional units was very large. The speed drop becomes highly more obvious in higher level SIMD configurations. However, the Dedicated-SPI_Shared-SPE approach showed an overlap in the overall performance with the Dedicated-SPMU which was a good sign that a tiny performance trade-off was made with a large chunk of area.

The Dedicated-SPMU were further evaluated with a more practical test, and that is by executing the layers of the VGG16 deep convolutional neural network algorithm. The first test showcased the performance of the T13 IMT architecture when having one active hart only, and when having all the harts active. We further evaluated the performance of the Dedicated-SPMU versus the Zero-riscy cores showing the performance in executing the layers of the VGG16 test with both large vectors, and small vectors, the Dedicated-SPMU continued to show performance superiority even in these real-life applications.

Area evaluations were made and we showed how much the DLP impacts the area, versus the TLP, Also, we saw how big the cross-bar was in the Dedicated-SPI_Shared-SPE scheme. Finally, we saw how much overhead does the T13 IMT core have over the in-order Riscy and Zeroriscy cores.

Finally, the dynamic power consumption and the energy consumption were shown for all the SPMU configurations. We saw that the dynamic power increased largely especially in SIMD 8 configurations. Also, that the SPMU schemes had a high power consumption. But when the time came to showcase the energy consumption, we saw the Hybrid approach was the most energy efficient such that Dedicated-SPMU SIMD-2 or the Dedicated-SPI_Shared-SPE SIMD-2 had the lowest energy consumption among all the hardware schemes.

Our study of the T13 showed how to easily make a high performance and energy efficient hardware accelerator for a very balanced IMT architecture, that interleaves a moderate number of harts. By simply adding a hardware accelerator that writes to its own dedicated memory, we can allow superscalar execution. This in hand will allow superscalar execution between the instructions that write to different memories without having stalls due to data dependencies, while still maintaining the same thread pool baseline, and not needing to interleave any additional harts to fence between the memory accesses. The study can be generalized to any hardware accelerator for IMT architecture, and not only convolution engines.

Appendix A

Klessydra Technical Manual

Chapter 1
Architecture overview

1.1 Features

The Klessydra processing core family is a set of processors featuring full compliance with the RISC-V instruction set and intended to be placed within the Pulpino microprocessor platform. To date, the Klessydra family includes

- a minimal gate count single-thread core, **Klessydra S0**. The S0 core is not maintained as open-source;
- a class of multi-threaded cores, **Klessydra T0**, available in different implementations called Klessydra T0ab;
- a class of extended versions of the T0 cores, named **Klessydra T1** cores, featuring an SPMU hardware accelerator.
- A class of fault tolerant versions of the T0 cores, featuring fault-tolerant mechanisms for harsh environment applications, named Klessydra F0x.

The Klessydra core family features:

- Full compliance with the RISC-V architecture specification (instruction set, control and status registers, interrupt handling mechanism and calling convention);
- Compliance with the standard RISC-V compilation toolchain;
- Interleaved multi-threaded execution of RISC-V *harts* (hardware threads);
- Easy and standardized multi-threading programming interface;
- Core synthesis on FPGA (presently, Xilinx Series 7 implementations have been tested);
- Hardware compliance with the Pulpino microprocessor platform, as pin-to-pin compatible alternative of the Pulpino RI5CY core;
- Software compliance with the Pulpino microprocessor platform, as compatible I/O memory map, interrupt handler memory map, program/data memory map;
- Extends the software test suite of Pulpino with custom tests designed specifically for the klessydra cores.

1.2 Naming convention

The different cores available in the Klessydra family follow the naming convention depicted in Fig. 1.1.

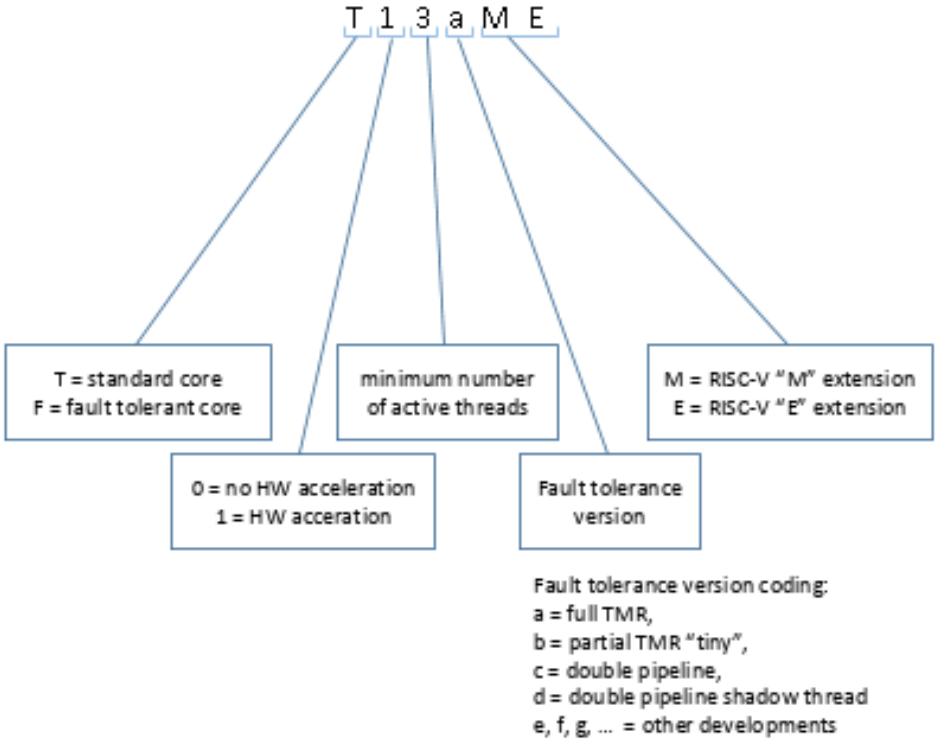


Fig. 1.1. Naming convention

1.3 Supported Instruction Set

To date, all the Klessydra cores implement the 32-bit integer RISC-V machine mode instruction set, namely user-level RV32I base integer instruction set version 2.1 and M-mode privileged instruction set version 1.1. T0 and T1 cores support the RV32IME set.

The T0 and T1 cores support the atomic instruction AMOSWAP.W from the RVA atomic instruction extension.

The T1 core extends the instruction set with non-native **custom** vector instructions for memory to scratchpad transfers and vector arithmetic operations. Vector instructions come in three different variants supporting different data width “8-bit, 16-bit, 32-bits” e.g.

Only M-mode operation is supported, so that no operating system support is implemented. Yet, the Klessydra family comes with a baseline runtime system software layer that implements part of the interrupt handling features and part of the multi-threaded programming model.

1.4 Multi-threading model

Klessydra S0 core supports single thread execution (RISC-V hart) only, with the following features:

- The hart can be interrupted by a trap such as an external interrupt or instruction exception. Software interrupts are supported, although their use is expected to be impractical in a single-thread execution environment. When the trap handling routine ends the core resumes the original execution thread (see Chapter “Exception and Interrupts” for details);
- The core can enter an idle state by means of the WFI instruction; when an external interrupt arrives at the core, the core starts the execution of the interrupt handling routine as the new hart of execution.

- The hart can be halted and resumed by means of the *Fetch_en* core interface signal.

Klessydra T0x, **T1**x and **F0**x cores implement interleaved multi-threading. At each clock cycle, a new instruction is fetched from a different hart (Fig. 1.2).

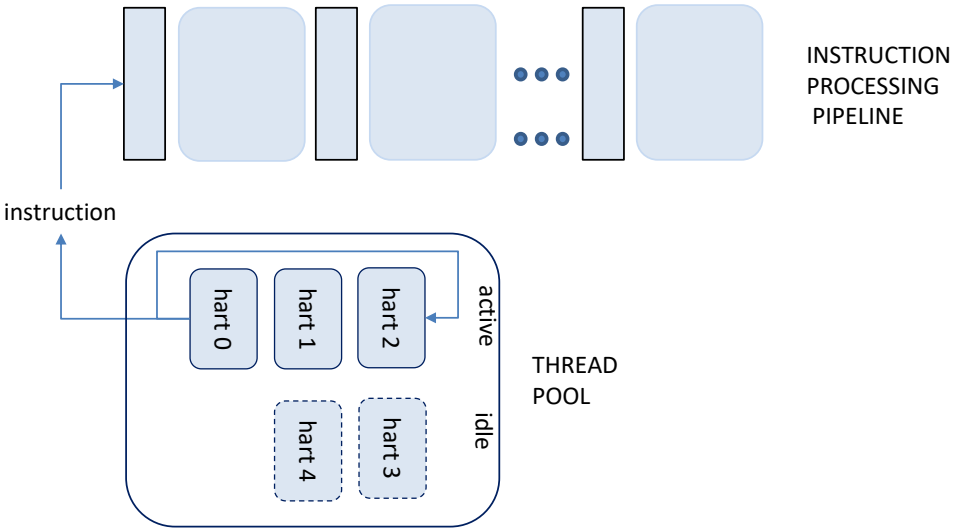


Fig. 1.2. Conceptual view of hardware thread (hart) interleaved execution

The execution has the following features:

- Each hart in the hardware thread pool can be either active or idle.
- An idle hart can be activated by an interrupt request directed to the hart. The core executes the interrupt handling routine within the hart. When the interrupt handling routine ends, the hart becomes idle again (see Chapter “Exception and Interrupts” for details).
- An active hart can be interrupted by instruction exceptions or interrupt requests. When the interrupt/exception handling routine ends and the signal **fetch_enable_i** is high, the core resumes the interrupted execution hart. (see Chapter “Exception and Interrupts” for details);
- An active hart can become idle by executing the WFI instruction;
- The maximum number of active harts is an architecture characteristic parameter called Thread Pool Size.
- Each hart is identified by an integer number ranging from 0 up to Thread Pool Size – 1.
- There is also a minimum number of active harts, needed to avoid data hazards between threads during the pipelined execution, called Thread Pool Baseline. The Thread Pool Baseline value is an architecture characteristic parameter related to the instruction pipeline organization implemented in the hardware microarchitecture of the core.
- When the number of active threads is less than the Thread Pool Baseline, one or more idle hart runs in the pipeline as NOP instructions.

As a general note, a higher Thread Pool Baseline value corresponds to a higher sustainable clock frequency and generally indicates a higher performance when running at full thread pool. For example, a T03 core will significantly outperform a T02 core when executing 4 harts.

1.5 Core Interfaces

The core interface is signal-to-signal compatible with the Pulpino microprocessor platform, and as such it is the same as Pulpino RI5CY cores. The detailed description follows.

Table.1.1 Clock, reset active low, test enable

Name	Direction	Width	Notes
clk_i	In	1	Core clock signal
clock_en_i	In	1	Core clock enable
rst_ni	In	1	Core reset signal, active low
test_en_i	In	1	Core test enable (unused)

Table.1.2 Initialization signals

Name	Direction	Width	Notes
boot_addr_i	In	32	Boot address value
core_id_i	In	4	Core id number
cluster_id_i	In	6	Cluster id number

Table 1.3 Program memory interface

Name	Direction	Width	Notes
instr_req_o	Out	1	Request signal, must stay high until accepted
instr_gnt_i	In	1	Request accepted, address may change in the next cycle
instr_rvalid_i	In	1	Instruction valid, stays high for exactly one cycle.
instr_addr_o	Out	32	Address
instr_rdata_i	In	32	Instruction read from memory

Table 1.4 Data Memory interface

Name	Direction	Width	Notes
data_req_o	Out	1	Request signal, must stay high until accepted
data_gnt_i	In	1	Request accepted, address may change in the next cycle
data_rvalid_i	In	1	Data valid, stays high for exactly one cycle
data_we_o	Out	1	Write enable, high = write, low = read
data_be_o	Out	4	Byte selection
data_addr_o	Out	32	Address
data_wdata_o	Out	32	Data to be written to memory
data_rdata_i	In	32	Data read from memory
data_err_i	In	1	Memory error signal

Table 1.5 Interrupt request / acknowledge

Name	Direction	Width	Notes
irq_i	In	1	Interrupt request signal
irq_id_i	in	5	Interrupt request vector value
irq_ack_o	out	1	Interrupt acknowledge signal
irq_id_o	in	5	Interrupt acknowledge vector value (unused)

Table 1.6 Debug interface

Name	Direction	Width	Notes
debug_req_i	In	1	Debug request
debug_gnt_o	Out	1	Debug request granted
debug_rvalid_o	Out	1	Debug data valid
debug_addr_i	In	15	Debug location address
debug_we_i	In	1	Debug write enable
debug_wdata_i	In	32	Debug data to be written to core
debug_rdata_o	Out	32	Debug data read from core
debug_halted_o	Out	1	Debug halt acknowledge
debug_halt_i	In	1	Debug halt request
debug_resume_i	in	1	Debug resume signal

Table 1.7 Miscellaneous control signals

Name	Direction	Width	Notes
fetch_enable_i	In	1	Fetch enable, stops the core
core_busy_o	Out	1	Core busy signal
ext_perf_counters_i	In	1	External performance counter signal (unused)

Chapter 2
Memory model and protocol

2.1 Instruction Fetch

The instruction fetch stage of the core is called FSM_IF and is able to supply one instruction to the instruction decode stage per cycle, if the program memory is able to serve one instruction per cycle. Instructions are word aligned, meaning that the two least significant bits in the PC are always set to 0, and the PC value is incremented by 4 units at each new fetch when no branch occurs. Compressed instruction format is not supported. No prefetch logic is present.

2.2 Memory Access Protocol

The program and data memory access protocol is pin-to-pin compatible with the Pulpino microprocessor platform, and as such it is the same as RI5CY / Zeroriscy cores'. The protocol that used to access the data memory works as follows. The program memory follows the same protocol except for the absence of write operation support.

The core provides a valid address in *data_addr_o* and sets *data_req_o* high. The memory then answers with *data_gnt_i* set high as soon as it is ready to serve the request. This may happen in the same cycle as the request is sent or any number of cycles later. After a grant is received, the address may be changed in the next cycle by the core. In addition, the *data_wdata_o, data_we_o* and *data_be_o* signals may be changed. After receiving a grant, the memory answers with *data_rvalid_i* set high if *data_rdata_i* is valid. This may happen one or more cycles after the grant has been received. The signal *data_rvalid_i* must also be set when a write operation is performed, although the *data_rdata_i* has no meaning in this case. Figure 2.1, Figure 2.2 and Figure 2.3 shows examples of the protocol timing.

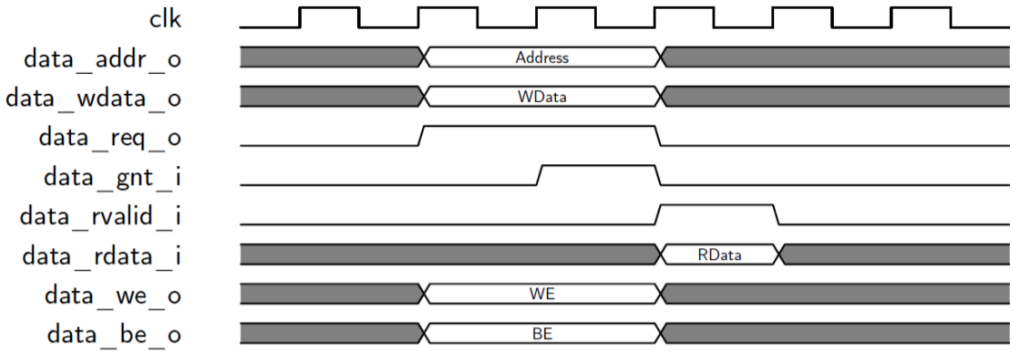


Figure 2.1 Basic Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

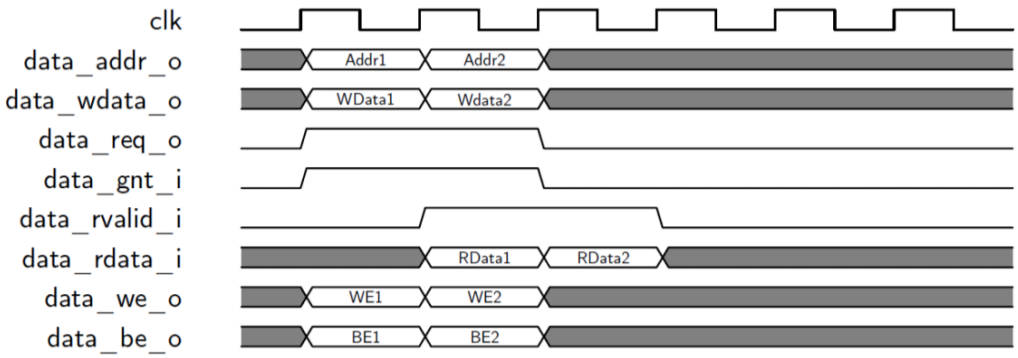


Figure 2.2 Back-to-Back Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

Figure 2.3 Slow Response Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

2.3 Misaligned Accesses

The core hardware does not perform misaligned accesses natively (i.e. accesses that are not aligned on natural word boundaries). If a misaligned memory access is requested by an instruction, the core produces an exception. There is no necessary hardware to realize the misaligned access by multiple aligned access. In compliance with RISC-V specification, misaligned accesses are therefore not guaranteed to be atomic.

2.4 Memory Address Map

Harts (i.e. hardware threads) running on a Klessydra core share the memory map illustrated in Fig. 2.4, which is compliant with the Pulpino SoC platform specification. The MIP CSR, one for each hart, are memory mapped starting at address 0x0000ff00 and allow for inter-thread interrupts, in compliance with the RISC-V specification. (Other CSRs are not memory mapped).

Each hart has its own stack, and the stack size and starting address are customizable at software level in the runtime system startup routine. The remaining memory space is available for inter-thread data communication.

For information about the addresses from 0x00 to 0x90, see the vector table in chapter 5. Address 0x94 is reserved to MTVEC.

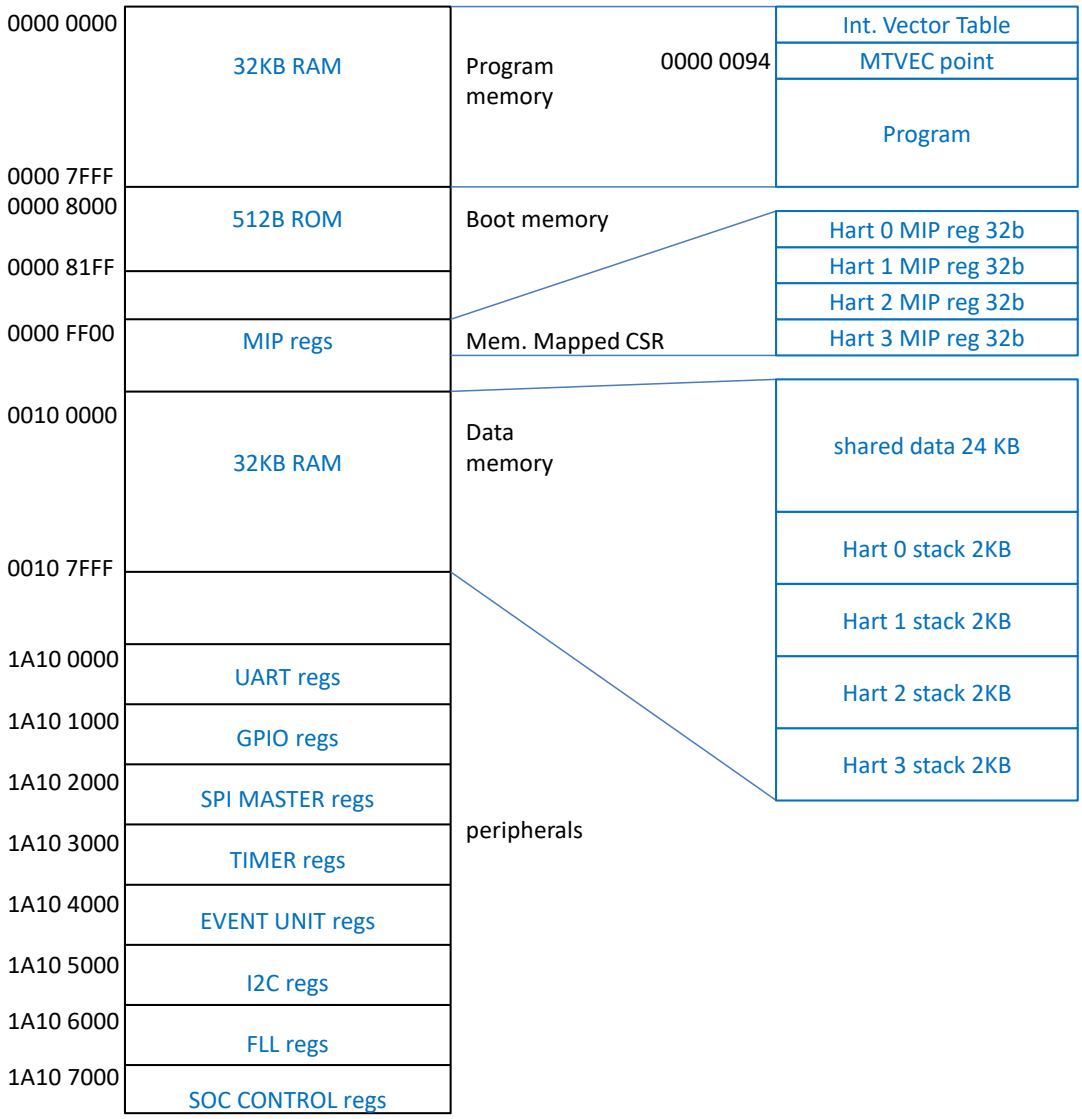


Fig. 2.4 Klessydra Memory Map (assuming 4 Threads, 2 KB stack per thread)

Chapter 3
Architecture Registers

3.1 Register File

Klessydra has 32x32-bit wide registers which form the registers x0 to x31. Register x0 is statically bound to 0 and can only be read. Write on register x0 has no side effect. They can be modified to 16x32 registers if the RV32E embedded extension was enabled.

3.2 Control and Status Registers

Klessydra cores implement a subset of the control and status registers specified in the RISC-V privileged specification, limited to the registers needed for M-mode operation and to the functionalities implemented in the core. Klessydra cores also implement some additional CSRs specifically needed for the core operations and/or for

compliance with the Pulpino microprocessor platform. This extended CSR sub-set is composed of the MIRQ, PCER, PCMR registers. The whole set of CSRs implemented in the Klessydra cores is as follows:

Table 3.1 CSR Registers

Name	CSR Address	Reset Value	R/W	Description
MSTATUS	0x300	0x0000_1808	R/W	Machine Status
MEPC	0x341	0x0000_0000	R/W	Machine Exception Program Counter
MCAUSE	0x342	0x0000_0000	R/W	Machine Trap Cause
PCER	0x7A0	0xFFFF_FFFF	R/W	Performance Counter Enable
MHPMCOUNTER	0xB00,0xB02 0xB03, 0xB06-0xB0A	0x0000_0000	R/W	Machine Performance-Monitoring Counter
MHPMEVENT	0x323, 0x326-0x32A	0x0000_0000	R/W	Machine Performance-Monitoring Event Selector
MCPUID	0xF00	0x0000_0100	R	CPU Description
MIMPID	0xF01	0x0000_8000	R	Implementation ID
MHARTID	0xF10	-	R	Hardware Thread ID
MIP	0x344	-	R/W	Interrupt Pending
MTVEC	0x305	0x0000_0094	R/W	Trap-Handler Base Address
MBADADDR	0x343	0x0000_0000	R/W	Misaligned Address Container
MIRQ	0xFC0	-	R	Interrupt Request
MVSIZE	0xBF0	0x0000_0001	R/W	Set Vector Size unit (T1)
MVTYPE	0xBF8	0x0000_0002	R/W	Set the data type (T1)
MPSCLFAC	0xBE0	0x0000_0000	R/W	Set the post scaling factor (T1)

- **MSTATUS Register bit map**

Table.3.1.1 MSTATUS bits

Bit #	R/W	Description
3	R/W	**Interrupt Enable:** When an exception is encountered, Interrupt Enable will be set to 1'b0, and it's state will be stored in bit '7'. When the *mret* instruction is executed, the original value of Interrupt Enable will be restored from the 7th bit. If you want to enable interrupt handling in your exception handler, set the Interrupt Enable to '1' inside your handler code.

7	R/W	**Interrupt Previous Enable:** Takes the state of the 3rd bit when serving an interrupt, and when an *mret* is served it stays latched to 1. And returns the 3rd bit back to it's original value.

- **MEPC Register**

  When an exception is encountered, the current program counter is saved in MEPC, and the core jumps to the MTVEC address. When an MRET instruction is executed, the value from MEPC replaces the current program counter, unless the return value was a WFI instruction, in this case we return to the instruction in the address after the WFI.

- **MCAUSE Register bit map**

Table.3.1.2 MCAUSE bits

Bit #	R/W	Description
31	R	**Interrupt:** This bit is set when the exception was triggered by an interrupt.
30	R	**WFI:** This bit indicates that the last instruction before entering the subroutine was a *WFI*
4:0	R	**Trap Cause:** "0011" for SW IRQ, "0111" for Timer IRQ, "1011" for External IRQ.

- **PCER Register bit map**

Each bit in the PCER register controls one performance counter. If the bit is 1, the counter is enabled and starts counting events. If it is 0, the counter is disabled and its value won't change.

Table.3.1.3 PCER bits

Bit #	Description
9	Branch Taken Counter Enable
8	Branch Counter Enable
7	Jump Counter Enable
6	Store Counter Enable
5	Load Access Counter Enable
4	Instruction Miss Counter Enable (currently not implemented)
3	Jump Access Stall Counter Enable (currently not implemented)
2	Load/Store Access Stall Counter Enable
1	Instruction Counter Enable
0	Cycle Counter Enable

- **MHPMCOUNTER Registers**

Klessydra Core includes a MCYCLE counter, a MINSTRET counter and others 6 additional event counters, MHPMCOUNTER3, MHPMCOUNTER6-MHPMCOUNTER10 of which only the first eight are used. The names of the registers are compliant to RISC-V but the counters are not divided into 32 lower bits and 32 higher bits. Only MCYCLE and MINSTRET are extended to 64 bits by the registers CYCLEH and MINSTRETH. The counter value is 32 bits unsigned integer.

Table.3.1.4 MHPMCOUNTER bits

Register	Description
MCYCLE	Counts the number of cycles the core was active (not sleeping)
MINSTRET	Counts the number of instructions executed
MHPMCOUNTER3	Number of load/store data hazards
MHPMCOUNTER4	currently not used
MHPMCOUNTER5	currently not used
MHPMCOUNTER6	Number of data memory loads executed
MHPMCOUNTER7	Number of data memory stores executed
MHPMCOUNTER8	Number of unconditional jumps
MHPMCOUNTER9	Number of branches. Counts taken and not taken branches
MHPMCUNTER10	Number of taken branches

- **MHPMEVENT Registers**

  In each MHPMEVENT register all the bits are statically bound to 0 except for the bit related to the counter that must be enabled. If that bit is 1, the counter is active and starts counting events. For instance, if the user wants to enable MHPMCOUNTER3 he will set the bit #2 (the 3th bit) of MHPMEVENT3 to 1. This procedure is equivalent to set PCER (3) to 1. The core includes 6 registers, MHPMEVENT3, MHPMEVENT6-MHPMEVENT10.

Table.3.1.5 MHPMEVENT bits

Register	Not Bound Bit #
MHPMEVENT3	2
MHPMEVENT4 (currently not used)	-
MHPMEVENT5 (currently not used)	-
MHPMEVENT6	5
MHPMEVENT7	6
MHPMEVENT8	7
MHPMEVENT9	8
MHPMEVENT10	9

- **MCPUID Register**

  The value of this register is fixed to 256 and cannot be changed. By using the CPUID opcode, software can determinate processor type and the presence of features.

- **MIMPID Register**

  The value of this register is fixed to 32768 and cannot be changed. MIMPID provides a unique encoding of the version of the processor implementation.

- **MHARTID Register**

  This register contains the integer ID of the hardware thread running the code. His value depends on Cluster and Core external signals and can only be read.

Table.3.1.6 MHARTID bits

Bit #	Description
9:4	ID of the Cluster
3:0	ID of the core within the cluster

- **MIP Register**

  The MIP register contains information about the type of pending interrupts. Bits #11 and #7 are enabled according to the external interrupt bits while bit #3 is settled to 1 to activate the SW interrupt routine.

Table.3.1.7 MIP bits

Bit #	R/W	Interrupt Type
11	R	External Interrupt
7	R	Time Interrupt
3	R/W	Software Interrupt

- **MTVEC Register**

  When an exception or an interrupt occurs, PC is loaded with the value of this register. MTVEC is the standard RISC-V base trap vector.

- **MIRQ Register**

  This register saves which interrupt has been called. The value of this register is four times the number of the interrupt's bit enabled. For instance, if irq_i(3) is set, MIRQ will be loaded with 12. If no interrupt is set, MIRQ value is 65535, that is just an arbitrary number.

- **BADADDR Register**

  When an instruction-fetch, load or store address-misaligned or access exception occurs, MBADADDR is written with the faulting address.

- **MVSIZE Register**

  Setting this register will set the vector size to be used by the mathematical unit. The biggest size should not exceed the SPM size, since overflow bits will be ignored.

- **MPSCLFAC Register**

  Contains the post scaling factor that determines the shift amount in KDOTPPS custom Klessydra instruction.

Chapter 4
Pipeline Organization

4.1 General concepts

Klessydra cores implement pipelined instruction processing. The number of pipeline stages differs among the cores as reported below. In the following, **F** indicates instruction fetch, **D** indicates operand read from register file and instruction decoding, **E** indicates operation execution, **W** indicates result writeback to the register file.
In all cores, the **F** stage latency is equal to the latency of program memory access, and variable latency program memory is supported (as for the case of instruction cache memory). The **F** stage latency is 1 in case of single-cycle-access program memory.
For other pipeline stages, the latency may be fixed or depend on external events (e.g. data memory latency, contention on CSR updating in case of interrupt requests). When a stage latency takes more than 1 cycle, the hardware stalls the preceding stage by local handshake signals. Similarly, each stage locally signals the succeeding stage when a new item is ready.
The generic microarchitecture for T0 cores is depicted in Fig. 4.1.
Each-thread is identified by a positive integer number *harc* (hardware context). The *harc* counter changes the *harc* value at each new instruction fetch, and the *harc* value associated to an instruction is passed through the pipeline stages. Most of the logic in the pipeline control section is replicated on a per-thread basis, and the *harc* value is used to properly index the logic units. Conversely, all the logic in the processing pipeline is not per-thread replicated with the only exception of the data register file.
In the S0 core, per-thread replication and the *harc*-related logic are natively absent.

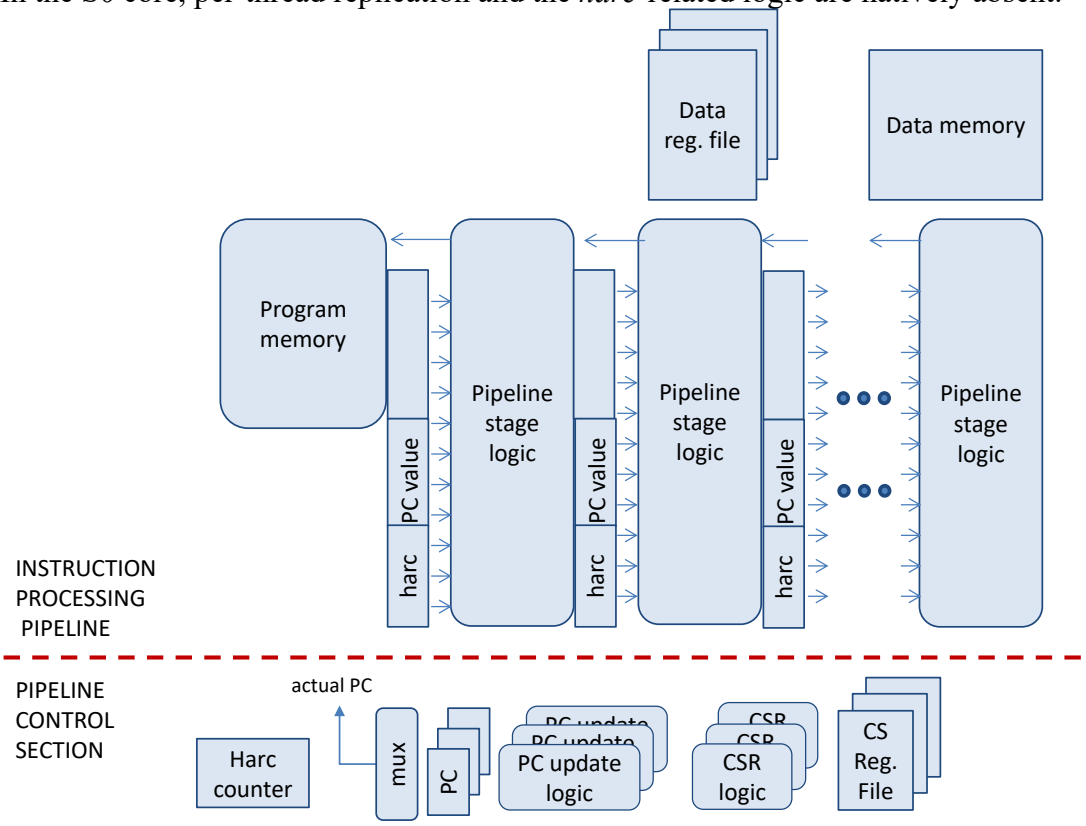


Fig. 4.1 – Generic pipeline microarchitecture scheme implemented in Klessydra T0 cores.

The specialize microarchitecture of T1 cores is represented in Fig 4.2.

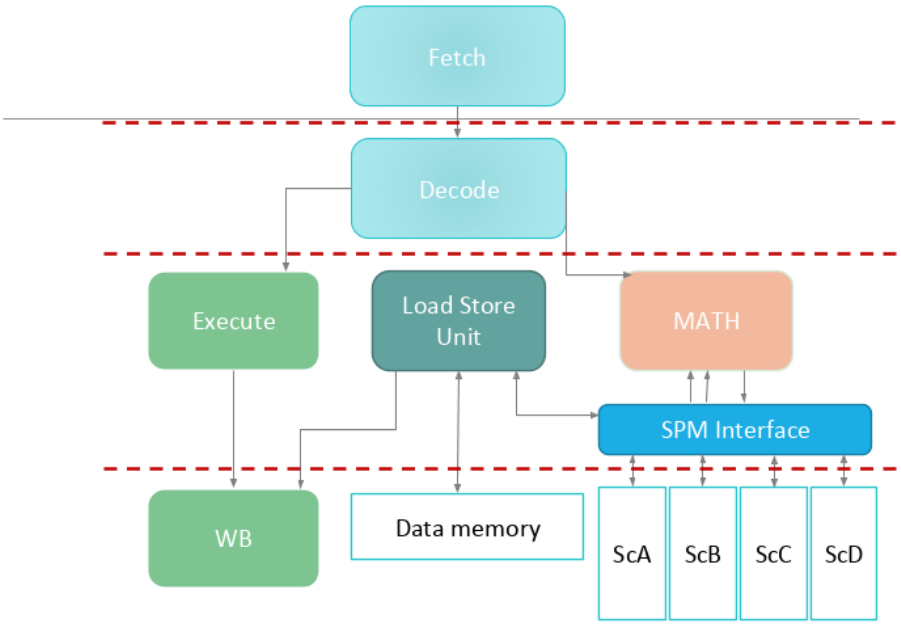


Fig. 4.2 – Datapath sketch of T1 cores.

T1 cores feature an execution stage that is split into a mathematical acceleration unit, scratchpad memory unit and a regular execution unit.

4.2 S0 core pipeline

The Klessydra S0 core implements a 2-stage pipeline according to the model **F / DEW**. The latency scheme is as follows:

	F	DEW
Load and store instructions	≥ 1	≥ 2
CSR instructions	≥ 1	≥ 2
All other instructions	≥ 1	1

Branch instructions are predicted as not-taken and are executed with a delay slot of 1 cycle; in case of taken branch the hardware flushes any wrongly fetched instruction from the pipeline.
Data hazards never occur.

4.3 T02*x* core pipeline

The Klessydra T0*x* cores implement a 3-stage pipeline according to the model F / D / EW. The latency scheme is as follows:

	F	D	EW
Load and store instructions	≥ 1	1	≥ 2
CSR instructions	≥ 1	1	≥ 2
Atomic memory operations	≥ 1	1	≥ 4
All other instructions	≥ 1	1	1

Branch instructions are predicted as not-taken and are executed with a delay slot of 2 cycles; in case of taken branch the hardware flushes any wrongly fetched instruction, belonging to the branching thread, from the pipeline. No pipeline flush occurs if at least 3 threads are interleaved in the pipeline.
Data hazards never occur, provided that at least 2 threads (Thread Pool Baseline) are interleaved in the pipeline.

4.4 T03*x* / T13x / Fxxx core pipeline

The Klessydra T03*x*/T13x cores implement a 4-stage pipeline according to the model F / D / E / W. The latency scheme is as follows:

	F	D	E	W
Load and store instructions	≥ 1	1	≥ 2	0
CSR instructions	≥ 1	1	≥ 2	0
Atomic memory operations	≥ 1	1	≥ 4	0
All other instructions	≥ 1	1	1	1
Specialized vector instructions	≥ 1	1	≥ 2	0

Branch instructions are predicted as not-taken and are executed with a delay slot of 3 cycles; in case of taken branch the hardware flushes any wrongly fetched instruction, belonging to the branching thread, from the pipeline. No pipeline flush occurs if at least 3 threads are interleaved in the pipeline.
Data hazards never occur, provided that at least 2 threads (Thread Pool Baseline) are interleaved in the pipeline.

Chapter 5
Exceptions and Interrupts

Klessydra cores implement exceptions on illegal instructions, on load and store instructions to invalid addresses, on misaligned memory accesses, and on ECALL instruction execution.
Klessydra cores implement vectorized interrupts, specifically supporting 32 separate interrupt service routines. There are three types of interrupt:

- Software Interrupt
- External Interrupt
- Timer Interrupt

The interrupt/exception vector table supported by Klessydra cores is compliant with the Pulpino platform interrupt vector table, as follows:

Table.5.1 Interrupt Handler address map

0x00-0x7C	Interrupts 0-31
0x80	Reset
0x84	Illegal Instruction
0x88	ECALL Instruction Executed
0x8C	LSU Error (Invalid Memory Access)
0x90	Software Interrupt

Except Code	Exception
0x0000_0002	ILLEGAL_INSN_EXCEPT_CODE
0x0000_0005	LOAD_ERROR_EXCEPT_CODE
0x0000_0007	STORE_ERROR_EXCEPT_CODE
0x0000_000B	ECALL ECALL_EXCEPT_CODE
0x0000_0004	LOAD_MISALIGNED_EXCEPT_CODE
0x0000_0006	STORE_MISALIGNED_EXCEPT_CODE
0x0000_0100	ILLEGAL_VECTOR_SIZE_EXCEPT_CODE
0x0000_0101	ILLEGAL_ADDRESS_EXCEPT_CODE
0x0000_0102	SCRATCHPAD_OVERFLOW_EXCEPT_CODE

Interrupt handling is accomplished in the core hardware by jumping to the address contained in MTVEC, in compliance with RISC-V specification; the pre-compiled startup software routine located at MTVEC address implements the interrupt vector table as it is shown above, jumping to the right handler routine address. The interrupt handler are to be written by the final user according to the target application.
Interrupts can be enabled/disabled on a global basis through the MSTATUS register; they cannot be individually enabled/disabled. Exceptions cannot be disabled.
When entering an interrupt routine, the core saves the current value of MIE (3^{rd}-bit) to the MPIE (7^{th}-bit) in the MSTATUS register; the state of MIE will be restored after returning from interrupt service routine.
If multiple interrupt requests arrive at the same cycle, the order of service is external interrupt first, then software interrupt, timer interrupt and exceptions (compliance to RISC-V specification).
In T0 cores, external interrupts are always re-directed to hart number 0. Software interrupts can be directed from any active hart, to any active or idle hart. Software interrupts allow inter-hart service requests.

In T0 cores and in T1 cores, as all status registers are replicated on a per-thread basis, the interrupt/exception handling mechanism is implemented referring to the status registers of the interrupted thread.
T1 cores introduce five more exceptions regarding the scratchpad handling. Exceptions will be raised if the Math Accelerator unit operands are from non-scratchpad addresses, or if writing or reading will result in a request from an overflown scratchpad address, or if we have dual writes or dual reads from the same scratchpad such as in the case of the LSU and Math Accelerator unit working simultaneously

Chapter 6
Scratchpads and mathematical unit (T1 version only)

6.1 Scratchpad memory subsystem

Klessydra T1 cores include scratchpad memories, with configurable number of scratchpads, banks, and scratchpad size and address mapping. The configurations can be modified in the PKG file of the synthesizable Klessydra suite. Each scratchpad memory (SPM) is composed of a set of memory banks; the number of banks available in each SPM is defined by on the "SIMD" parameter value set in the PKG file.

Each bank address holds a 32-bit word. An SPM data line is composed of as many words as the "SIMD" value. As addresses remain byte-aligned, the address distance between SPM data lines is + SIMD*4. Each word on the line has its own address and can be independently accessed (with 4-byte aligned address).

Any SPM bank can be read or written to. For read access, any bank that is not bank0 will cause the data read in SIMD fashion to be rotated as if it was coming from bank0 by a read rotator. While for write access, any write to a bank different from bank0 will cause the data to be rotated to its correct destination bank by a write rotator.The rotators were made to align the two input source operands

Each scratchpad has one read port, and one write port. Each port has size SIMD*32-bits (e.g. for SIMD=4, we have 4*32 = 128 bits).

The SPMs can be accessed by the SPMU or the LSU. When a dual read (or dual write) access is requested to the same SPM on the same port by two different units (SPMU and LSU), priority will be given to the unit that requested the access first and the other unit will be halted until the operation is finished. Due to the in-order single-issue pipeline of the Klessydra cores it is not possible that the two units request access to the SPM in the same cycle).

All transfers to/from the scratchpads go through an interface called **SPI**. Both SPM read and writes happen through this SPI wrapper. The LSU and SPMU are the only units that interact with the scratchpad memories, always through the SCI.

6.2 Mathematical accelerator unit

The Mathematical unit was designed to execute custom Klessydra instructions targeting vector, DSP-like and CNN-inference-like operations.

The Mathematical unit interfaces to the SPI unit by means of two read ports for the operands coming from the scratchpad memory interface, and one write port to the scratchpad memory interface. The read and write port width are dependent on the SIMD parameter value set in the PKG file.

The custom instruction set executed in the Mathematical unit are listed in table 7.1. It executes different instructions many of which have different variants. Table.7.1 also shows the SIMD capability of the Mathematical unit. A composition of partial functional units has been adopted to enhance the SIMD execution and to optimize the area consumption mathematical unit. Addition instructions use a combination of 8-bit adders to make 8-bit, 16-bit, and 32-bit additions. Multiplication instructions use a combination of 16-bit multipliers to perform 8-bit, 16-bit and 32-bit multiplications[1].

[1] 16-bit multipliers were chosen over 8-bit multipliers since doing 32-bit multiplication using 8-bit multipliers would be inefficient, and also 16-bit are the optimal choice needed for utilizing DSP blocks on presently available FPGAs

There are no 32-bit arithmetic units in the mathematical except for the 32-bit shifters, that can be configured to do 8,16,32-bit right arithmetic or logical shifts, accumulators (32-bit adders, and 16-bit adders for both 8, and 16-bit), and Rectify Linear Unit (RELU).
When working on vectors, the Mathematical unit exploits built-in hardware loop (zero overhead loops), executing the following steps in hardware:

a. Increment the source and destination vector pointers to fetch the next element chunk;
b. Decrement the remaining number of elements to process;
c. Evaluate a conditional branch to check whether the number of remaining elements reached zero.

The Mathematical unit can operate in parallel with respect to the other execution units. Since the custom Klessydra instructions never have dependencies with the standard RISCV instructions, the IE unit and LSU can work in parallel with the Mathematical unit.
The Mathematical unit recovers its state when a halt occurs due to dual read/write access.

Chapter 7
Fault Tolerance Support (F0x versions only)

Klessydra core versions F0x support several mechanisms of fault tolerance targeting aerospace and safety critical application. Most of the mechanisms implemented address tolerance to single event upset (SEU) in memory elements (registers and memories).

7.1 Basic mechanisms

Supported standard FT mechanisms are Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR), both based on repetition on functional modules and comparison of outputs through a voting system.

DMR uses two replicas of combinational or sequential logic, it can only detect errors and has a low area occupation and power consumption but high time of implementation.

Basic-TMR is a triple repetition of combinational or sequential logic and a majority voter; it has the same time of implementation of DMR and also area and power consumption increase.

Full-TMR adds a triple redundancy to both logic and registers at cost of area and power consumption and it uses cross voters to guarantee high error correction capability.

Global-TMR is based on full-TMR but it can be automated through synthesis tools.

7.2 F03a: Fully TMR – Partial TMR Design

The protection of control and state registers is a priority because they contain vital information about core operation and they are written only once at first run, so a TMR must be used.

Counter registers are less critical because they are constantly refreshed. Each core has a dedicated counter with many 32-bit registers, so it's suggested to use alternative techniques:

- MSB-TMR: triple redundancy only of N most significant bits, reducing area impact.
- DMR: detection of an error trigger a trap identified by a code and it's managed by the software.
- Software protection: no hardware protection, the software periodically reads and compares counters.

Pipeline robustness is fundamental in TMR because redundancy does not protect registers from loading wrong values that irredeemably corrupt code execution. So redundancy has to be applied to all registers between pipeline stages and state registers of state machines.

Registers file are dedicated for each thread so a TMR has the highest impact in terms of area.

The voting system also lengthens critical path that lowers the maximum clock frequency.

Program counter unit has a dedicated 32-bit register for each thread and some flip-flop to store events and conditions that must be resolved from the unit. Flip-Flop corruption doesn't lead to loss of control because PC update is defined by signals coming from pipeline.

An error on exception service Flip-Flop is more critical because it requires a response from CSR.

Due to the small area occupation on few registers and Flip-Flop it's suggested to protect with TMR the PC unit.

7.3 F03b: Double Pipeline Design with Check&Restore

In this version, a protection technique is used that does not allow error correction, but the area occupied by the TMR version is reduced by a third, without losing reliability. This design implies a change of the internal architecture of the core. The structure is based on a new Processing Unit composed of two Pipelines, the CRU and a new CSR unit derived from the TMR version. In this architecture, a checkpoint is created before critical portions of code or periodically. Checkpoint control is managed by the Check Restore Unit (CRU) and the CSR.
The two pipelines are the same as the Klessydra T03 version. The input and output signals of the pipelines are controlled by the CRU, which can drive them exclusively to start checkpoint or restore procedures, not natively implemented in the Pipeline. To allow CRU functioning and error check, there is an internal register, called CRSTATUS. This critical register is protected by TMR redundancy and can be read (but not written via) CSR instructions. The management of the check and restore system requires instructions for management and control. These instructions are:

- Chepoint start instruction
- Instruction to activate thread dependency
- Instruction to restart the restore manually
- Instruction to deactivate protected mode

The double pipeline structure adds two operating modes to the system:

- the normal mode - it allows to deactivate the clock of the not used pipeline in order to reduce the dynamic consumption of the core;
- the "single pipeline" mode - allows you to increase core's life in critical environments. This mode is integrated and supported by the hardware but requires that the code is written ad properly.

A portion of the software is used to check the correct functioning of the hardware: if a pipeline is damaged, it can be disabled. The core is then used with a single unprotected pipeline. The robustness of the processing must be granted to the software, which will be executed in a redundant manner, sacrificing the processing speed.
The CRU is the heart of the DoublePipe architecture protection system. The system is based on the comparison of the pipeline outputs. In case of output's discrepancy, the CRU activates a flag that indicates the presence of an error. In the next execution phase of the thread with an active flag, the CRU takes control of the outputs, simulating an illegal instruction with a specific cause code. At this point, a software routine takes care of recovering the values of the register files previously saved in memory. At the end of the illegal instruction routine, the PC unit loads the program counter with the value saved during checkpoint creation. The return to a checkpoint does not deactivate the protected mode or eliminate the checkpoint. The unit manages part of the dedicated instructions of this architecture and the internal control register. The control register contains information on the configuration of the CRU, on the execution status of the core and on any hanging errors. This register can be read (but not written) by the user.
The DuoblePipe CSR, unlike the original version, includes a specific register which is used to back up the PC. The register is not addressable, and its writing is managed by the CSR during the execution of the pseudo instructions developed for this architecture. Other registers are instead extended in use and functionality compared to the RISC-V standard: the writing with particular values of some registers will be interpreted as an instruction. The CSR, together with the CRU, takes care of serving the instructions for starting a checkpoint and restoring it in the event of an error.
The DoublePipe program counter unit is substantially identical to the original version in terms of functionality. Since the activation system of a checkpoint is based on the start of a particular software interrupt, it is necessary to add a condition in the PC unit that allows the service of this type of interrupt despite the interrupts being disabled.

7.4 F03c: Shadow Thread Double Pipe

The F03c architecture is based on the possibility to correct errors, by using a double pipeline and a redundant execution of the thread instructions. To obtain corrections of the errors, 3 copies of the same result are necessary:

- two contemporary copies obtained through pipeline redundancy;
- a third copy obtained through a temporally out of phase processing.

A Shadow control unit (SCU) handle the pipelines and the architecture synchronization. During the shadow processing, the SCU handles two different instructions. These are executed in the pipelines. To solve latency problem between instructions, the SCU can put the pipeline on hold in order to complete the execute phase in a synchronized way. The register file management is left to SRU unit, that constantly communicates with the SCU. The SCU provides information about processing in the pipeline, indicating to the SRU whether the instructions require access to the registers. In case of error, the SRU performs a memory access and retrieves the value of the register. If the error is detected in conjunction with an instruction that requires reading the same file, the SRU sends a signal to the SCU which blocks the pipelines. Once received the data from the memory, a triple comparison is made, and the correct data is sent to the pipelines. When this phase is completed, the SCU unlocks the pipelines. The writing in memory is started in conjunction with the WB phase. The copy of the regfile has priority over a possible access instruction in memory, which is put on hold, locking the pipelines.

The CSR ST does not contain dedicated registers but is equipped with additional input signals. The operation of the unit depends on:

- main processing: the CSR executes the commands received from the pipeline processing, previously controlled by the SCU. It also reacts to any hardware routines to serve exceptions and interrupts or following a return instruction.
- Shadow processing: the CSR simulates the execution of the access instructions by supplying the values contained in the registers. The values contained in the registers are not modified unless explicitly commanded by the SCU. No interrupt or exception affects the CSR at this step. An eventual interrupt event is served at the next main processing.

Writing to the internal registers can be disabled at any time by the SCU, which keeps a constant check on the CSR. This architectural difference allows the value sent to the registers to be blocked at any time. Loading incorrect values (in the TMR registers) is always prevented. The triple redundancy technique completely loses its effectiveness in the event of an error in the logic that sends the data.
The architecture of the PC ST unit differs from the original version of the core as it must guarantee the functioning of the shadow structure that requires up to two PCs simultaneously. The portion that manages the PC during the fetching phase of the shadow processing is located inside the SCU. The PC ST unit, in addition to supplying the correct PC value to the SCU, must manage the correct updating of the internal PC registers. This is done by receiving information on the location of the shadow thread within the Pipeline. Thanks to this information the unit can execute or block the updating of the PC registers. In the event of interrupts, exceptions or jumps during the execution phase of the Shadow processing, the unit locks the PC update system waiting for the main processing phase. If the conditions that triggered an interrupt, an exception or any request to change the program flow remain, the PC will update and start the procedure. This allows to avoid serving the same interrupt twice. In this unit there is an input signal that allows the SCU to stop updating the PC at any time in case of error. In this case indeed, the TMR protection of the inside registers is not able to correct the loading of an incorrect value.

Chapter 8
Debug Support

Klessydra core supports common baseline debug features: halting the program flow, reading data register file, reading the PC value and enabling a single step execution. Software breakpoints are implemented by the RISC-V instruction EBREAK.
The debug operations are intended at core level and not per-thread. When entering debug mode, the whole core (i.e. with all its threads) enters debug mode. The internal debug unit accesses information related to the thread whose instruction is in the execution stage of the core pipeline in the current clock cycle.
The debug hardware interface is the same as the memory interface, but on separate buses. Every access to debug facilities is done by an access to debug registers.
To halt the core, external debug unit has to set DBG_CTRL[0] bit. If DBG_CTRL[0] is set, the core is in single step mode, so clearing the DGB_HIT[0] bit enable execution of a single instruction.
Debug registers are always accessible. Program counter and register file are accessible only when the core is halted. Which register of register file external debug unit requires is specified in [6:2] bit of the address.

Table.8.1 Debug Registers

Address	Name	Description
0x00	DBG_CTRL	Debug Control
0x04	DBG_HIT	Debug Hit
0x2000	DBG_PPC	Next PC
0x2004	DBG_NPC	Previous PC
0x400-0x47C	GPR(x0-x31)	General Purpose Registers

Table.8.2 Debug Control register bit map

Bit #	R/W	Description
16	R/W	HALT bit: When set to ‘1’, the core enters debug mode, when reset to ‘0’, the core exits debug mode.
0	R/W	SSTE bit: Single-step enable bit.

Table.8.3 Debug Hit register bit map

Bit #	R/W	Description

0	R/W	SSTH: Single-step hit, sticky bit that must be cleared by external debugger in order to execute next instruction.

Table.8.4 Debug Next Program Counter register bit map

Bit #	R/W	Description
31:0	R/W	NPC: Next PC to be executed

Table.8.5 Debug Previous Program Counter register bit map

Bit #	R/W	Description
31:0	R/W	PPC: Previous PC, already executed

Chapter 9
Instruction Set

9.1 Integer Register-Immediate operations

Table.9.1 Register-Immediate operations

Name	Binary format type	Assembly syntax
ADDI – add immediate	I	ADDI rd, rs1, imm
SLTI - set if less immediate	I	SLTI rd, rs1, imm
SLTIU - set if less imm. uns.	I	SLTIU rd, rs1, imm
ANDI - and immediate	I	ANDI rd, rs1, imm
ORI - or immediate	I	ORI rd, rs1, imm
XORI – excl. or immediate	I	XORI rd, rs1, imm
SLLI – shift left logical imm.	I	SLLI rd, rs1, shamt
SRLI– shift right logical imm.	I	SRLI rd, rs1, shamt
SRAI – shift right arithm. imm.	I	SRAI rd, rs1, shamt
LUI - load upper immediate	U	LUI rd, imm
AUIPC - add upper imm. to pc	I	AUIPC rd, imm

- ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low 32 bits of the result. ADDI *rd, rs1, 0* can be used to implement a register move operation.
- SLTI places the value 1 in register *rd* if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd. SLTIU is similar but compares the values as unsigned numbers.
- ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Notably, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1*.
- SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field.
- LUI is used to build 32-bit constants. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.
- AUIPC is used to build PC-relative addresses. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the PC, then places the result in register *rd*.

9.2 Integer Register-Register Operations

Table.9.2 Register-Register Operations

Name	Binary format type	Assembly syntax
ADD - add	R	ADD rd, rs1, rs2
SLT - set if less	R	SLT rd, rs1, rs2
SLTU – set if less unsigned	R	SLTU rd, rs1, rs2
AND - and	R	AND rd, rs1, rs2
OR - or	R	OR rd, rs1, rs2
XOR - exclusive or	R	XOR rd, rs1, rs2

SLL – shift left logical	R	SLL rd, rs1, rs2
SRL – shift right logical	R	SRL rd, rs1, rs2
SUB – subtract	R	SUB rd, rs1, rs2
SRA - shift right arithmetic	R	SRA rd, rs1, rs2

- ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low 32 bits of results are written to the destination.
- SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd, x0, rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudo-op SNEZ *rd, rs*).
- AND, OR, and XOR perform bitwise logical operations.
- SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

9.3 Unconditional Jumps

Table.9.3 Unconditional Jumps

Name	Binary format type	Assembly syntax
JAL - jump and link	UJ	JAL rd, imm
JALR – jump to reg and link	UJ	JALR rd, rs1, imm

- The jump and link (JAL) instruction uses the J-immediate to encode a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (PC+4) into register *rd*. Plain unconditional jumps are encoded as a JAL with *rd = x0*.
- The indirect jump instruction JALR (jump and link register) obtains the target address by adding the 12-bit signed I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (PC+4) is written to register rd. Register *x0* can be used as the destination if the result is not required.
- The JAL and JALR instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.

9.4 Conditional Branches

Table.9.4 Branches

Name	Binary format type	Assembly syntax
BEQ – branch if equal	SB	BEQ rs1, rs2,imm
BNE - branch if not eq.	SB	BNE rs1, rs2,imm
BLT – branch if less	SB	BLT rs1, rs2,imm
BGE– branch if greater	SB	BGE rs1, rs2,imm
BLTU – branch if less	SB	BLTU rs1, rs2,imm
BGEU – branch if greater	SB	BGEU rs1, rs2,imm

- BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively.
- BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively.
- BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively.

- All branch instructions use the 12-bit B-immediate to encode signed offsets in multiples of 2, and add the offset to the current PC to give the target address. The conditional branch range is ±4 KiB.

9.5 Memory access Instructions

Table.9.5 Load-Store Instructions

Name	Binary format type	Assembly syntax
LB - load byte	I	LB rd, rs1, imm
LH - load half word	I	LH rd, rs1, imm
LW - load word	I	LW rd, rs1, imm
LBU - load byte unsigned	I	LBU rd, rs1, imm
LHU - load half word unsig.	I	LHU rd, rs1, imm
SB - store byte		SB rs1,rs2,imm
SH - store half word		SH rs1,rs2,imm
SW - store word		SW rs1,rs2,imm

- Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.
- The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory

9.6 CSR Instructions (Read-Set-Clear)

Table.9.6 CSR Instructions

Name	Binary format type	Assembly syntax
CSRRW - csr read/write		CSRRW rd, csr, rs1
CSRRS - csr read & set		CSRRS rd, csr, rs1
CRSSC - csr read & clear		CSRRC rd, csr, rs1
CSRRWI - csr rd/wr. Imm.		CSRRWI rd, csr, imm
CSRRSI - csr rd & set imm		CSRRSI rd, csr, imm
CSRRCI - csr rd & clr imm		CSRRCI rd, csr, imm

- The CSRRW instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to 32 bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If *rd=x0*, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.
- The CSRRS instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).
- The CSRRC instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit

that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

- For both CSRRS and CSRRC, if *rs1*=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if rs1 specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.
- The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an 32-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if *rd=x0*, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

9.7 CSR Privileged Instructions

Table.9.7 Privileged Instructions

Name	Binary format type	Assembly syntax
ECALL – environment call		ECALL
EBREAK – break to envir.		EBREAK
WFI – wait for IRQ		WFI
MRET – machine return		MRET

- The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.
- The EBREAK instruction is presently implemented in the S0 core only (future update in T0 cores and T1 cores).
- The WFI is a wait for interrupt instruction, that latches the thread in an idle state until an interrupt arrives.
- The MRET updates the program counter with the address of the instruction being executed before entering the trap handling routine. Unless the instruction was a WFI, we return to the address after it.

9.8 Atomic Instructions

Table.9.8 Atomic Instructions

Name	Binary format type	Assembly syntax
AMOSWAP.W.AQ	R	AMOSWAP.W.AQ rd,rs1,rs2
AMOSWAP.W.RL	R	AMOSWAP.W.RL rd,rs1,rs2

- The atomic memory operations AMOSWAP.W atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a swap between the loaded value and the original value in *rs2*, then store the swapped value to the address in *rs1*.
  The implementation follows “release consistency”. The AMOSWAP.W.AQ instruction implements a read-modify-write operation suited to lock acquiring, while the AMOSWAP.W.AQ instruction implements a read-modify-write operation suited to lock releasing.

The S0 core does not support Atomic Instructions.

9.9 Klessydra Custom Extensions (T1 version only)

Table.9.9 Klessydra custom extensions

Name	Binary format type	Assembly syntax
KMEMLD	R	kmemld rd, rs1, rs2
KMEMSTR	R	kmemstr rd, rs1, rs2
KADDV	R	kaddv rd, rs1, rs2
KSUBV	R	ksubv rd, rs1, rs2
KVMUL	R	kvmul rd, rs1, rs2
KVRED	R	kvred rd, rs1, rs2
KDOTP	R	kdotp rd, rs1, rs2
KSVADDSC	R	ksvaddsc rd, rs1, rs2
KSVADDRF	R	ksvaddrf rd, rs1, rs2
KSVMULSC	R	ksvmulsc rd, rs1, rs2
KSVMULRF	R	ksvmulrf rd, rs1, rs2
KDOTP	R	kdotp rd, rs1, rs2
KDOTPPS	R	kdotpps rd, rs1, rs2
KSRLV	R	ksrlv rd, rs1, rs2
KSRAV	R	ksrav rd, rs1, rs2
KRELU	R	krelu rd, rs1, rs2
KBCAST	R	kbcast rd, rs1
KVCP	R	kvcp rd, rs1

- KMEMLD: loads the number of bytes specified by ‘rs2’ in the scratchpad memory at address ‘rd’, from the address ‘rs1’ in the main memory.
- KMEMSTR: loads the number of bytes specified by ‘rs2’ in the main memory at address ‘rs1’, from the address ‘rd’ in the scratchpad memory.
- KADDV: adds the operands in the scratchpad at addresses in ‘rs1’ and in ‘rs2’ and stores the result as a vector at the address ‘rd’ in the scratchpad memory.
- KSUBV: subtracts the operands in the scratchpad at addresses in ‘rs1’ and in ‘rs2’ and stores the result as a vector at the address ‘rd’ in the scratchpad memory.
- KVMUL: multiplies the vector elements of rs1 and rs2 and stores the result in rd.
- KVRED: performs vector reduction between the elements at addresses ‘rs1’ and ‘rs2’, and stores the scalar in ‘rd’.
- KDOTP: multiplies the operands at addresses in ‘rs1’ and in ‘rs2’, the multiply intermediate results are accumulated, and the final results are stored as a scalar in the address in ‘rd’.
- KDOTPPS: performs post scaling dot product on the elements at addresses in ‘rs1’ and ‘rs2’ and puts the result in ‘rd’. The multiplication result is shifted by the value set the CSR register ‘MPSCLFAC’.
- KSVADDSC/RF: adds the scalar operand in the register file or scratchpad address in ‘rs1’ with a scalar value that is in ‘rs2’. The result is stored as a vector at address in ‘rd’. (A faster alternative to using KBCAST).
- KSVMULRF/SC multiplies the scalar operand in the register file / scratchpad in ‘rs1’ with a scalar value that is in ‘rs2’. The result is stored as a vector in the address in ‘rd’. (A faster alternative to using KBCAST).
- KSRLV/KSRAV: does right logical/arithmetic shifts on the vector at the address in ‘rs1’ by the shift amount in ‘rs2’ and stores the vector results at the address in ‘rd’.
- KRELU: does linear rectification on the negative values of the vector at the address in ‘rs1’ and puts the rectified vector at the address in ‘rd’.

- KBCAST: does a vector broadcast of the scalar value contained in scalar register ‘rs1’ to the vector at the address in ‘rd’.
- KVCP: copies the vectors starting at the address in ‘rs1’ to the address in ‘rd’. Both addresses are in scratchpad memory space.

All logical-arithmetic vector instructions should all be used in conjunction with the CSR register ‘MVSIZE’ in order to specify the size of the vector to be processed by the operation.

Appendix B

T13 VHDL Code

This appendix includes some of the main RTL files of the Klessydra T13, not all files have been included in order to keep this thesis more compact. The language is VHDL_2008

Also, one important note, the term DSP refers to the SPMU. Earlier implementations of the unit were designed to make a DSP, however, the term was later changed to SPMU

Another note: SC is the earlier abbreviation of scratchpad memory, which is now known as SPM.

The sources included are the package file, the SPE, SPI, and SPM entities, all sources can be found at Github [31][32][33].

1. Package file Parameters

```
library ieee;
use ieee.math_real.all;
use ieee.std_logic_1164.all;

package thread_parameters_klessydra is

  type array_2d     is array (integer range<>) of std_logic_vector;
  type array_3d     is array (integer range<>) of array_2d;
  type array_2d_int is array (integer range<>) of integer;

  constant THREAD_ID_SIZE   : integer := 4;

  constant THREAD_POOL_SIZE : integer := 3;  -- Changing the TPS to less than "number of pipeline stages-1" is not allowed. And making it bigger than "pipeline stages-1" is okay but not recommended
  constant NOP_POOL_SIZE    : integer := 2;  -- should be static and not touched, unless the number of pipeline stages changes; presently unused

  constant BRANCHING_DELAY_SLOT    : integer := 3;  -- should be static and not touched, unless the number of pipeline stages change

  constant HARC_SIZE : integer := THREAD_POOL_SIZE; -- for the moment we do not implement "nop" threads
  subtype harc_range is integer range THREAD_POOL_SIZE - 1 downto 0;  -- will be used replicated units in the core


  ------------------------------------------------------------------------------
  --   ######  #####  ###   ## ####### #####  ######   ######  --
  --  ##       #     # ## #  ## ##         #   ##        ##       --
  --  ##       #     # ## # ## #####      #   ## ####   #####    --
  --  ##       #     # ##  # ## ##         #   ##    ##       ##  --
  --   ######  #####  ##   ### ##       #####  ######  ######   --
  ------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  constant RF_SIZE              : natural := 32;  -- Regfile size, Can be set to 32 for RV32I or 16 for RV32E
  constant RV32M                : natural := 0;   -- Enable the M-extension of the risc-v instruction set
  constant accl_en              : natural := 0;   -- Enable the generation of the special purpose accelerator
  constant replicate_accl_en    : natural := 0;   -- Set to 1 to replicate the accelerator for every thread
  constant multithreaded_accl_en : natural := 0;   -- Set to 1 to let the replicated accelerator share the functional units (note: replicate_accl_en must be set to '1')
  constant SPM_NUM                              : natural := 4;   -- The number of scratchpads available "Minimum allowed is two"
  constant Addr_Width           : natural := 14;  -- This address is for scratchpads. Setting this will make the size of the spm to be: "2^Addr_Width -1"
  constant SPM_STRT_ADDR        : std_logic_vector(31 downto 0) := x"1000_0000";  -- This is starting address of the spms, it shouldn't be bigger than 2^32, and shouldn't overlap any sections in the memory map
  constant SIMD                 : natural := 1;   -- Changing the SIMD, would change the number of the functional units in the dsp, and the number of banks in the spms (can be power of 2 only e.g. 1,2,4,8)
  constant MCYCLE_EN            : natural := 0;   -- Can be set to 1 or 0 only. Setting to zero will disable MCYCLE and MCYCLEH
  constant MINSTRET_EN          : natural := 0;   -- Can be set to 1 or 0 only. Setting to zero will disable MINSTRET and MINSTRETH
```

```
  constant MHPMCOUNTER_EN       : natural := 0;   -- Can be set to 1 or 0 only. Setting to zero will disable all program counters except
"MCYCLE/H" and "MINSTRET/H"
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------

  constant RF_CEIL          : natural := integer(ceil(log2(real(RF_SIZE))));
  constant TPS_CEIL         : natural := integer(ceil(log2(real(THREAD_POOL_SIZE))));
  constant TPS_BUF_CEIL     : natural := integer(ceil(log2(real(THREAD_POOL_SIZE-1))));
  constant SPM_ADDR_WID     : natural := integer(ceil(log2(real(SPM_NUM+1))));
  constant SIMD_BITS        : natural := integer(ceil(log2(real(SIMD))));
  constant Data_Width       : natural := 32;
  constant SIMD_Width       : natural := SIMD*Data_Width;
  --constant XLEN    : natural := 32;  -- aaa use this instead of Data_Width, the name is shorter and more convenient

  constant ACCL_NUM : natural := (THREAD_POOL_SIZE - (THREAD_POOL_SIZE-1)*(1-replicate_accl_en));
  constant FU_NUM   : natural := (ACCL_NUM - (ACCL_NUM-1)*(multithreaded_accl_en));

  subtype accl_range is integer range ACCL_NUM - 1 downto 0;  -- will be used replicated accelerators in the core
  subtype fu_range   is integer range FU_NUM - 1 downto 0; -- will be used replicated accelerators in the core

  type fsm_IE_states is (sleep, reset, normal, csr_instr_wait_state, debug);
  type mul_states is (mult, accum);
  type div_states is (init, divide);
  type fsm_LS_states is (normal , data_valid_waiting);

  constant dsp_init        : std_logic_vector(1 downto 0) := "00";
  constant dsp_halt_hart   : std_logic_vector(1 downto 0) := "01";
  constant dsp_exec        : std_logic_vector(1 downto 0) := "10";
```

2. SPE Unit

```
-- ieee packages ------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.numeric_std.all;
use std.textio.all;

-- local packages ------------
use work.riscv_klessydra.all;
use work.thread_parameters_klessydra.all;

-- DSP  pinout ---------------------
entity DSP_Unit is
  port (
              -- Core Signals
    clk_i, rst_ni                : in std_logic;
    -- Processing Pipeline Signals
    rs1_to_sc                    : in  std_logic_vector(SPM_ADDR_WID-1 downto 0);
    rs2_to_sc                    : in  std_logic_vector(SPM_ADDR_WID-1 downto 0);
    rd_to_sc                     : in  std_logic_vector(SPM_ADDR_WID-1 downto 0);
             -- CSR Signals
    MVSIZE                       : in  array_2d(harc_range)(Addr_Width downto 0);
    MVTYPE                       : in  array_2d(harc_range)(3 downto 0);
    MPSCLFAC                     : in  array_2d(harc_range)(4 downto 0);
    dsp_except_data              : out array_2d(accl_range)(31 downto 0);
     -- Program Counter Signals
     dsp_taken_branch            : out std_logic_vector(accl_range);
     dsp_except_condition        : out std_logic_vector(accl_range);
    -- ID_Stage Signals
    decoded_instruction_DSP      : in  std_logic_vector(DSP_UNIT_INSTR_SET_SIZE-1 downto 0);
    harc_EXEC                    : in  harc_range;
    pc_IE                        : in  std_logic_vector(31 downto 0);
    RS1_Data_IE                  : in  std_logic_vector(31 downto 0);
    RS2_Data_IE                  : in  std_logic_vector(31 downto 0);
    RD_Data_IE                   : in  std_logic_vector(Addr_Width -1 downto 0);
    dsp_instr_req                : in  std_logic_vector(accl_range);
    spm_rs1                      : in  std_logic;
    spm_rs2                      : in  std_logic;
    vec_read_rs1_ID              : in  std_logic;
    vec_read_rs2_ID              : in  std_logic;
    vec_write_rd_ID              : in  std_logic;
    busy_dsp                     : out std_logic_vector(accl_range);
    -- Scratchpad Interface Signals
```

```
    dsp_data_gnt_i                : in  std_logic_vector(accl_range);
    dsp_sci_wr_gnt                : in  std_logic_vector(accl_range);
    dsp_sc_data_read              : in  array_3d(accl_range)(1 downto 0)(SIMD_Width-1 downto 0);
    dsp_we_word                   : out array_2d(accl_range)(SIMD-1 downto 0);
    dsp_sc_read_addr              : out array_3d(accl_range)(1 downto 0)(Addr_Width-1 downto 0);
    dsp_to_sc                     : out array_3d(accl_range)(SPM_NUM-1 downto 0)(1 downto 0);
    dsp_sc_data_write_wire        : out array_2d(accl_range)(SIMD_Width-1 downto 0);
    dsp_sc_write_addr             : out array_2d(accl_range)(Addr_Width-1 downto 0);
    dsp_sci_we                    : out array_2d(accl_range)(SPM_NUM-1 downto 0);
    dsp_sci_req                 : out array_2d(accl_range)(SPM_NUM-1 downto 0);
    -- tracer signals
    state_DSP                 : out array_2d(accl_range)(1 downto 0)

            );
end entity;  ----------------------------------------


architecture DSP of DSP_Unit is

  signal nextstate_DSP : array_2d(accl_range)(1 downto 0);

  -- Virtual Parallelism Signals
  signal relu_en                    : std_logic_vector(accl_range);  -- enables the use of the shifters
  signal shift_en                   : std_logic_vector(accl_range);  -- enables the use of the shifters
  signal add_en                     : std_logic_vector(accl_range);  -- enables the use of the adders
  signal mul_en                     : std_logic_vector(accl_range);  -- enables the use of the multipliers
  signal accum_en                   : std_logic_vector(accl_range);  -- enables the use of the accumulator
  signal relu_en_wire               : std_logic_vector(accl_range);  -- enables the use of the shifters
  signal shift_en_wire              : std_logic_vector(accl_range);  -- enables the use of the shifters
  signal add_en_wire                : std_logic_vector(accl_range);  -- enables the use of the adders
  signal mul_en_wire                : std_logic_vector(accl_range);  -- enables the use of the multipliers
  signal accum_en_wire              : std_logic_vector(accl_range);  -- enables the use of the accumulatorss
  signal add_en_pending_wire          : std_logic_vector(accl_range);  -- signal to preserve the request to access the adder "multhithreaded mode" only
  signal shift_en_pending_wire        : std_logic_vector(accl_range);  -- signal to preserve the request to access the shifter "multhithreaded mode"
only
  signal mul_en_pending_wire          : std_logic_vector(accl_range);  -- signal to preserve the request to access the multiplier "multhithreaded mode"
only
  signal accum_en_pending_wire        : std_logic_vector(accl_range);  -- signal to preserve the request to access the accumulator "multhithreaded
mode" only
  signal relu_en_pending_wire         : std_logic_vector(accl_range);  -- signal to preserve the request to access the ReLU "multhithreaded mode"
only
  signal add_en_pending             : std_logic_vector(accl_range);  -- signal to preserve the request to access the adder "multhithreaded mode" only
  signal shift_en_pending           : std_logic_vector(accl_range);  -- signal to preserve the request to access the shifter "multhithreaded mode" only
  signal mul_en_pending             : std_logic_vector(accl_range);  -- signal to preserve the request to access the multiplier "multhithreaded mode"
only
  signal accum_en_pending           : std_logic_vector(accl_range);  -- signal to preserve the request to access the accumulator "multhithreaded
mode" only
  signal relu_en_pending            : std_logic_vector(accl_range);  -- signal to preserve the request to access the ReLU "multhithreaded mode" only
  signal busy_add                 : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_mul                 : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_shf                 : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_acc                 : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_rel                 : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_add_wire              : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_mul_wire              : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_shf_wire              : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_acc_wire              : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal busy_rel_wire              : std_logic;  -- busy signal active only when the FU is shared and currently in use
  signal halt_hart                : std_logic_vector(accl_range); -- halts the thread when the requested functional unit is in use
  signal fu_req                   : array_2D(accl_range)(4 downto 0); -- Each threa has request bits equal to the total number of FUs
  signal fu_gnt                   : array_2D(accl_range)(4 downto 0); -- Each threa has grant bits equal to the total number of FUs
  signal fu_gnt_wire                : array_2D(accl_range)(4 downto 0); -- Each threa has grant bits equal to the total number of FUs
  signal fu_gnt_en                  : array_2D(accl_range)(4 downto 0); -- Enable the giving of the grant to the thread pointed at by the issue buffer
  signal fu_rd_ptr                : array_2D(4 downto 0)(TPS_BUF_CEIL-1 downto 0); -- five rd pointers each has a number of bits equal to
ceil(log2(THREAD_POOL_SIZE-1))
  signal fu_wr_ptr                  : array_2D(4 downto 0)(TPS_BUF_CEIL-1 downto 0); -- five rd pointers each has a number of bits equal to
ceil(log2(THREAD_POOL_SIZE-1))
  -- five buffers for each FU times the "TPS-1" and not "TPS" since there is always one thread active, and not needing a buffer. Each buffer hold the
thread_ID "TPS_CEIL"
  signal fu_issue_buffer              : array_3D(4 downto 0)(THREAD_POOL_SIZE-2 downto 0)(TPS_CEIL-1 downto 0);

  -- Functional Unit Ports ---
  --signal dsp_in_sign_bits             : array_2d(accl_range)(4*SIMD-1 downto 0);             -- vivado unsynthesizable, but more efficient alternative
  signal dsp_in_shifter_operand         : array_2d(fu_range)(SIMD_Width -1 downto 0);
  signal dsp_in_shifter_operand_lat     : array_2d(fu_range)(SIMD_Width -1 downto 0);          -- 15 bits because i only want to latch the signed bits
  signal dsp_in_shifter_operand_lat_wire : array_2d(fu_range)(SIMD_Width -1 downto 0);
  signal dsp_int_shifter_operand        : array_2d(fu_range)(SIMD_Width -1 downto 0);
  signal dsp_out_shifter_results        : array_2d(fu_range)(SIMD_Width -1 downto 0);
```

```
  signal dsp_in_relu_operands          : array_2d(fu_range)(SIMD_Width-1 downto 0);
  signal dsp_in_mul_operands           : array_3d(fu_range)(1 downto 0)(SIMD_Width-1 downto 0);
  signal dsp_out_mul_results           : array_2d(fu_range)(SIMD_Width-1 downto 0);
  signal dsp_out_relu_results          : array_2d(fu_range)(SIMD_Width-1 downto 0);
  signal dsp_in_accum_operands         : array_2d(fu_range)(SIMD_Width-1 downto 0);
  signal dsp_out_accum_results         : array_2d(fu_range)(31 downto 0);
  signal dsp_in_adder_operands         : array_3d(fu_range)(1 downto 0)(SIMD_Width-1 downto 0);
  signal dsp_in_adder_operands_lat     : array_3d(fu_range)(1 downto 0)(SIMD_Width/2 -1 downto 0); -- data_Width devided by the number of
pipeline stages
  signal dsp_out_adder_results         : array_2d(fu_range)(SIMD_Width-1 downto 0);

  signal carry_8_wire                : array_2d(fu_range)(SIMD-1 downto 0);  -- carry-out bit of the "dsp_add_8_0" signal
  signal carry_16_wire               : array_2d(fu_range)(SIMD-1 downto 0);  -- carry-out bit of the "dsp_add_16_8" signal
  signal carry_16                  : array_2d(fu_range)(SIMD-1 downto 0);  -- carry-out bit of the "dsp_add_16_8" signal
  signal carry_24_wire               : array_2d(fu_range)(SIMD-1 downto 0);  -- carry-out bit of the "dsp_add_24_16" signal
  signal dsp_add_8_0                 : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits, contains the results of 8-bit adders
  signal dsp_add_16_8                : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits  contains the results of 8-bit adders
  signal dsp_add_8_0_wire            : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits, contains the results of 8-bit adders
  signal dsp_add_16_8_wire           : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits  contains the results of 8-bit adders
  signal dsp_add_24_16_wire          : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits  contains the results of 8-bit adders
  signal dsp_add_32_24_wire          : array_3d(fu_range)(SIMD-1 downto 0)(8 downto 0); -- 9-bits, this should be 8 if we choose to discard the
overflow of the addition of the upper byte
  signal mul_tmp_a                   : array_3d(fu_range)(SIMD-1 downto 0)(Data_Width-1 downto 0);
  signal mul_tmp_b                   : array_3d(fu_range)(SIMD-1 downto 0)(Data_Width-1 downto 0);
  signal mul_tmp_c                   : array_3d(fu_range)(SIMD-1 downto 0)(Data_Width-1 downto 0);
  signal mul_tmp_d                   : array_3d(fu_range)(SIMD-1 downto 0)(Data_Width-1 downto 0);
  signal dsp_mul_a                   : array_2d(fu_range)(SIMD_Width -1 downto 0); --  Contains the results of the 16-bit multipliers
  signal dsp_mul_b                   : array_2d(fu_range)(SIMD_Width -1 downto 0); --  Contains the results of the 16-bit multipliers
  signal dsp_mul_c                   : array_2d(fu_range)(SIMD_Width -1 downto 0); --  Contains the results of the 16-bit multipliers
  signal dsp_mul_d                   : array_2d(fu_range)(SIMD_Width -1 downto 0); --  Contains the results of the 16-bit multipliers

  signal carry_pass                  : array_2d(accl_range)(2 downto 0);  -- carry enable signal, depending on it's configuration, we can do KADDV8,
KADDV16, KADDV32
  signal FUNCT_SELECT_MASK             : array_2d(accl_range)(31 downto 0); -- when the mask is set to "FFFFFFFF" we enable KDOTP32
execution using the 16-bit muls
  signal twos_complement                 : array_2d(accl_range)(31 downto 0);
  signal dsp_shift_enabler             : array_2d(accl_range)(15 downto 0);
  signal dsp_in_shift_amount           : array_2d(accl_range)(4 downto 0);

  signal dsp_sc_data_write_wire_int    : array_2d(accl_range)(SIMD_Width-1 downto 0);
  signal dsp_sc_data_write_int         : array_2d(accl_range)(SIMD_Width-1 downto 0);

  signal MVTYPE_DSP                    : array_2d(accl_range)(1 downto 0);
  signal vec_write_rd_DSP              : std_logic_vector(accl_range);  -- Indicates whether the result being written is a vector or a scalar
  signal vec_read_rs1_DSP              : std_logic_vector(accl_range);  -- Indicates whether the operand being read is a vector or a scalar
  signal vec_read_rs2_DSP              : std_logic_vector(accl_range);  -- Indicates whether the operand being read is a vector or a scalar
  signal dotp                        : std_logic_vector(accl_range);  -- indicator used in the pipeline handler to switch functional units
  signal dotpps                      : std_logic_vector(accl_range);  -- indicator used in the pipeline handler to switch functional units
  signal wb_ready                    : std_logic_vector(accl_range);
  signal halt_dsp                    : std_logic_vector(accl_range);
  signal halt_dsp_lat                : std_logic_vector(accl_range);
  signal recover_state               : std_logic_vector(accl_range);
  signal recover_state_wires           : std_logic_vector(accl_range);
  signal dsp_data_gnt_i_lat            : std_logic_vector(accl_range);
  signal shifter_stage_1_en            : std_logic_vector(accl_range);
  signal shifter_stage_2_en            : std_logic_vector(accl_range);
  signal shifter_stage_3_en            : std_logic_vector(accl_range);
  signal adder_stage_1_en              : std_logic_vector(accl_range);
  signal adder_stage_2_en              : std_logic_vector(accl_range);
  signal adder_stage_3_en              : std_logic_vector(accl_range);
  signal mul_stage_1_en                : std_logic_vector(accl_range);
  signal mul_stage_2_en                : std_logic_vector(accl_range);
  signal mul_stage_3_en                : std_logic_vector(accl_range);
  signal relu_stage_1_en               : std_logic_vector(accl_range);
  signal relu_stage_2_en               : std_logic_vector(accl_range);
  signal accum_stage_1_en              : std_logic_vector(accl_range);
  signal accum_stage_2_en              : std_logic_vector(accl_range);
  signal accum_stage_3_en              : std_logic_vector(accl_range);
  signal dsp_except_data_wire          : array_2d(accl_range)(31 downto 0);

  signal decoded_instruction_DSP_lat   : array_2d(accl_range)(DSP_UNIT_INSTR_SET_SIZE -1 downto 0);
  signal overflow_rs1_sc               : array_2d(accl_range)(Addr_Width downto 0);
  signal overflow_rs2_sc               : array_2d(accl_range)(Addr_Width downto 0);
  signal overflow_rd_sc                : array_2d(accl_range)(Addr_Width downto 0);
  signal dsp_rs1_to_sc                 : array_2d(accl_range)(SPM_ADDR_WID-1 downto 0);
  signal dsp_rs2_to_sc                 : array_2d(accl_range)(SPM_ADDR_WID-1 downto 0);
  signal dsp_rd_to_sc                  : array_2d(accl_range)(SPM_ADDR_WID-1 downto 0);
  signal dsp_sc_data_read_mask         : array_2d(accl_range)(SIMD_Width-1 downto 0);
```

```
  signal RS1_Data_IE_lat              : array_2d(accl_range)(31 downto 0);
  signal RS2_Data_IE_lat              : array_2d(accl_range)(31 downto 0);
  signal RD_Data_IE_lat               : array_2d(accl_range)(Addr_Width -1 downto 0);
  signal MVSIZE_READ                    : array_2d(accl_range)(Addr_Width downto 0);  -- Bytes remaining to read
  signal MVSIZE_READ_MASK                 : array_2d(accl_range)(Addr_Width downto 0);  -- Bytes remaining to read
  signal MVSIZE_WRITE                   : array_2d(accl_range)(Addr_Width downto 0);  -- Bytes remaining to write
  signal MPSCLFAC_DSP                   : array_2d(accl_range)(4 downto 0);
  signal busy_dsp_internal            : std_logic_vector(accl_range);
  signal busy_DSP_internal_lat          : std_logic_vector(accl_range);
  signal rf_rs2                       : std_logic_vector(accl_range);
  signal SIMD_RD_BYTES_wire             : array_2d_int(accl_range);
  signal SIMD_RD_BYTES                  : array_2d_int(accl_range);

  component ACCUMULATOR
  port(
     clk_i                        : in  std_logic;
     rst_ni                       : in  std_logic;
     MVTYPE_DSP                     : in  array_2d(accl_range)(1 downto 0);
     accum_stage_1_en               : in  std_logic_vector(accl_range);
     accum_stage_2_en               : in  std_logic_vector(accl_range);
     recover_state_wires            : in  std_logic_vector(accl_range);
     halt_dsp_lat                   : in  std_logic_vector(accl_range);
     state_DSP                      : in  array_2d(accl_range)(1 downto 0);
     decoded_instruction_DSP_lat      : in  array_2d(accl_range)(DSP_UNIT_INSTR_SET_SIZE -1 downto 0);
     dsp_in_accum_operands          : in  array_2d(fu_range)(SIMD_Width-1 downto 0);
     dsp_out_accum_results          : out array_2d(fu_range)(31 downto 0)
               );
  end component;

-------------------------------------------------------------------------------------------
---------------------------------- DSP BEGIN ----------------------------------------------
begin


  busy_dsp          <= busy_dsp_internal;

  DSP_replicated : for h in accl_range generate


  ------------ Sequential Stage of DSP Unit ---------------------------------------------------------------
  DSP_Exec_Unit : process(clk_i, rst_ni)  -- single cycle unit, fully synchronous

  begin
    if rst_ni = '0' then
      rf_rs2(h)   <= '0';
      dotpps(h)   <= '0';
      dotp(h)     <= '0';
      recover_state(h) <= '0';
    elsif rising_edge(clk_i) then
      if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then

        case state_DSP(h) is

          when dsp_init =>

            ------------------------------------------------------------------------------------------
            -- ##### ###   ## ##### ########   ######  ######   #######  --
            --    #    ## #  ##    #        ##       ##     # ##         ##       # --
            --    #    ## #  ##    #        ##       ##     #  #####    #######  --
            --    #    ##  # ##    #        ##       ##     #        ##  ##         --
            -- ##### ##   ### #####      ##       ######  ######   ##          --
            ------------------------------------------------------------------------------------------
            FUNCT_SELECT_MASK(h) <= (others => '0');
            twos_complement(h)   <= (others => '0');
            rf_rs2(h)   <= '0';
            dotpps(h)   <= '0';
            dotp(h)     <= '0';
            -- Set signals to enable correct virtual parallelism operation
            if (decoded_instruction_DSP(KADDV_bit_position)    = '1'  or
               decoded_instruction_DSP(KSVADDSC_bit_position) = '1') and
               MVTYPE(h)(3 downto 2) = "10" then
              carry_pass(h) <= "111";  -- pass all carry_outs
            elsif decoded_instruction_DSP(KSVADDRF_bit_position) = '1' and
                 MVTYPE(h)(3 downto 2) = "10" then
              carry_pass(h) <= "111";  -- pass all carry_outs
              rf_rs2(h) <= '1';
            elsif (decoded_instruction_DSP(KADDV_bit_position)    = '1'  or
                 decoded_instruction_DSP(KSVADDSC_bit_position) = '1') and
```

```
          MVTYPE(h)(3 downto 2) = "01" then
        carry_pass(h) <= "101";  -- pass carrries 9, and 25
      elsif decoded_instruction_DSP(KSVADDRF_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "01" then
        carry_pass(h) <= "101";  -- pass carrries 9, and 25
        rf_rs2(h) <= '1';
      elsif (decoded_instruction_DSP(KADDV_bit_position)     = '1'  or
           decoded_instruction_DSP(KSVADDSC_bit_position)  = '1') and
           MVTYPE(h)(3 downto 2) = "00" then
        carry_pass(h) <= "000";  -- don't pass carry_outs and keep addition 8-bit
      elsif decoded_instruction_DSP(KSVADDRF_bit_position)  = '1' and
          MVTYPE(h)(3 downto 2) = "00" then
        carry_pass(h) <= "000";  -- don't pass carry_outs and keep addition 8-bit
        rf_rs2(h) <= '1';
      elsif decoded_instruction_DSP(KSUBV_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "10" then
        carry_pass(h) <= "111";  -- pass all carry_outs
        twos_complement(h) <= "00010001000100010001000100010001";
      elsif decoded_instruction_DSP(KSUBV_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "01" then
        carry_pass(h) <= "101";  -- pass carrries 9, and 25
        twos_complement(h) <= "01010101010101010101010101010101";
      elsif decoded_instruction_DSP(KSUBV_bit_position)  = '1' and
          MVTYPE(h)(3 downto 2) = "00" then
        carry_pass(h) <= "000";  -- don't pass carry_outs and keep addition 8-bit
        twos_complement(h) <= "11111111111111111111111111111111";
      elsif decoded_instruction_DSP(KDOTP_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "10" then
        -- KDOTP32 does not use the adders of KADDV instructions but rather adds the mul_acc results using it's own adders
        FUNCT_SELECT_MASK(h) <= (others => '1');  -- This enables 32-bit multiplication with the 16-bit multipliers
        dotp(h) <= '1';
      elsif decoded_instruction_DSP(KDOTP_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "01" then
        dotp(h) <= '1';
      elsif decoded_instruction_DSP(KDOTP_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "00" then
        dotp(h) <= '1';
      elsif decoded_instruction_DSP(KDOTPPS_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "10" then
        FUNCT_SELECT_MASK(h) <= (others => '1');  -- This enables 32-bit multiplication with the 16-bit multipliers
        dotpps(h) <= '1';
      elsif decoded_instruction_DSP(KDOTPPS_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "01" then
        dotpps(h) <= '1';
      elsif decoded_instruction_DSP(KDOTPPS_bit_position)  = '1' and
          MVTYPE(h)(3 downto 2) = "00" then
        dotpps(h) <= '1';
      elsif decoded_instruction_DSP(KSVMULRF_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "10" then
        FUNCT_SELECT_MASK(h) <= (others => '1');
        rf_rs2(h) <= '1';
      elsif decoded_instruction_DSP(KSVMULRF_bit_position) = '1' and
          MVTYPE(h)(3 downto 2) = "01" then
        rf_rs2(h) <= '1';
      elsif decoded_instruction_DSP(KSVMULRF_bit_position)  = '1' and
          MVTYPE(h)(3 downto 2) = "00" then
        rf_rs2(h)  <= '1';
      elsif (decoded_instruction_DSP(KVMUL_bit_position)    = '1'  or
                                              decoded_instruction_DSP(KSVMULSC_bit_position) = '1') and
          MVTYPE(h)(3 downto 2) = "10" then
        FUNCT_SELECT_MASK(h) <= (others => '1');
      end if;

     -- We backup data from decode stage since they will get updated

      MVSIZE_READ_MASK(h) <= MVSIZE(harc_EXEC);
      MVSIZE_WRITE(h) <= MVSIZE(harc_EXEC);
      MPSCLFAC_DSP(h) <= MPSCLFAC(harc_EXEC);
      MVTYPE_DSP(h) <= MVTYPE(harc_EXEC)(3 downto 2);
      decoded_instruction_DSP_lat(h)  <= decoded_instruction_DSP;
      vec_write_rd_DSP(h) <= vec_write_rd_ID;
      vec_read_rs1_DSP(h) <= vec_read_rs1_ID;
      vec_read_rs2_DSP(h) <= vec_read_rs2_ID;
      dsp_rs1_to_sc(h) <= rs1_to_sc;
      dsp_rs2_to_sc(h) <= rs2_to_sc;
                dsp_rd_to_sc(h)  <= rd_to_sc;
      RD_Data_IE_lat(h) <= RD_Data_IE;
      -- Increment the read addresses
```

```
356             if dsp_data_gnt_i(h) = '1' then
357               if vec_read_rs1_ID = '1' then
358                 RS1_Data_IE_lat(h) <= std_logic_vector(unsigned(RS1_Data_IE) + SIMD_RD_BYTES_wire(h));   -- source 1 address increment
359               else
360                 RS1_Data_IE_lat(h) <= RS1_Data_IE;
361               end if;
362               if vec_read_rs2_ID = '1' then
363                 RS2_Data_IE_lat(h) <= std_logic_vector(unsigned(RS2_Data_IE) + SIMD_RD_BYTES_wire(h)); -- source 2 address increment
364               else
365                 RS2_Data_IE_lat(h) <= RS2_Data_IE;
366               end if;
367               -- Decrement the vector elements that have already been operated on
368               if unsigned(MVSIZE(harc_EXEC)) >= SIMD_RD_BYTES_wire(h) then
369                 MVSIZE_READ(h) <= std_logic_vector(unsigned(MVSIZE(harc_EXEC)) - SIMD_RD_BYTES_wire(h));  -- decrement by SIMD_BYTE
370     Execution Capability
371               else
372                 MVSIZE_READ(h) <= (others => '0');                                  -- decrement the remaining bytes
373               end if;
374             else
375               RS1_Data_IE_lat(h) <= RS1_Data_IE;
376               RS2_Data_IE_lat(h) <= RS2_Data_IE;
377               MVSIZE_READ(h) <= MVSIZE(harc_EXEC);
378             end if;
379             ---------------------------------------------------------------------------
380
381           when dsp_exec =>
382             recover_state(h) <= recover_state_wires(h);
383             if halt_dsp(h) = '1' and halt_dsp_lat(h) = '0' then
384               dsp_sc_data_write_int(h) <= dsp_sc_data_write_wire_int(h);
385             end if;
386
387             if halt_dsp(h) = '0' then
388               -- Increment the write address when we have a result as a vector
389                                         if vec_write_rd_DSP(h) = '1' and wb_ready(h) = '1' then
390                 RD_Data_IE_lat(h)  <= std_logic_vector(unsigned(RD_Data_IE_lat(h))  + SIMD_RD_BYTES(h)); -- destination address increment
391                                         end if;
392               if wb_ready(h) = '1' then
393                 if to_integer(unsigned(MVSIZE_WRITE(h))) >= SIMD_RD_BYTES(h) then
394                   MVSIZE_WRITE(h) <= std_logic_vector(unsigned(MVSIZE_WRITE(h)) - SIMD_RD_BYTES(h));       -- decrement by SIMD_BYTE
395     Execution Capability
396                 else
397                   MVSIZE_WRITE(h) <= (others => '0');                               -- decrement the remaining bytes
398                 end if;
399               end if;
400               -- Increment the read addresses
401               if to_integer(unsigned(MVSIZE_READ(h))) >= SIMD_RD_BYTES(h) and dsp_data_gnt_i(h) = '1' then -- Increment the addresses untill all
402     the vector elements are operated fetched
403                 if vec_read_rs1_DSP(h) = '1' then
404                   RS1_Data_IE_lat(h) <= std_logic_vector(unsigned(RS1_Data_IE_lat(h)) + SIMD_RD_BYTES(h));   -- source 1 address increment
405                 end if;
406                 if vec_read_rs2_DSP(h) = '1' then
407                   RS2_Data_IE_lat(h) <= std_logic_vector(unsigned(RS2_Data_IE_lat(h)) + SIMD_RD_BYTES(h)); -- source 2 address increment
408                 end if;
409               end if;
410               -- Decrement the vector elements that have already been operated on
411               if dsp_data_gnt_i(h) = '1' then
412                 if to_integer(unsigned(MVSIZE_READ(h))) >= SIMD_RD_BYTES(h) then
413                   MVSIZE_READ(h) <= std_logic_vector(unsigned(MVSIZE_READ(h)) - SIMD_RD_BYTES(h));        -- decrement by SIMD_BYTE
414     Execution Capability
415                 else
416                   MVSIZE_READ(h) <= (others => '0');                                -- decrement the remaining bytes
417                 end if;
418               end if;
419               dsp_sc_data_read_mask(h) <= (others => '0');
420               if dsp_data_gnt_i_lat(h) = '1' then
421                 if to_integer(unsigned(MVSIZE_READ_MASK(h))) >= SIMD_RD_BYTES(h) then
422                   dsp_sc_data_read_mask(h) <= (others => '1');
423                   MVSIZE_READ_MASK(h) <= std_logic_vector(unsigned(MVSIZE_READ_MASK(h)) - SIMD_RD_BYTES(h));       -- decrement by
424     SIMD_BYTE Execution Capability
425                 else
426                   MVSIZE_READ_MASK(h) <= (others => '0');
427                   dsp_sc_data_read_mask(h)(to_integer(unsigned(MVSIZE_READ_MASK(h)))*8 - 1 downto 0) <= (others => '1');
428                 end if;
429               end if;
430             end if;
431
432           when others =>
433             null;
```

```
        end case;
      end if;
    end if;
  end process;

  ------------ Combinational Stage of DSP Unit ------------------------------------------------------------------------
  DSP_Excpt_Cntrl_Unit_comb : process(all)

  variable busy_DSP_internal_wires : std_logic;
  variable dsp_except_condition_wires : replicated_bit;
  variable dsp_taken_branch_wires : replicated_bit;

  begin

    busy_DSP_internal_wires       := '0';
    dsp_except_condition_wires(h)  := '0';
    dsp_taken_branch_wires(h)     := '0';
    wb_ready(h)                  <= '0';
    halt_dsp(h)                  <= '0';
    nextstate_DSP(h)             <= dsp_init;
    recover_state_wires(h)        <= recover_state(h);
    dsp_except_data_wire(h)       <= dsp_except_data(h);
    overflow_rs1_sc(h)            <= (others => '0');
    overflow_rs2_sc(h)            <= (others => '0');
    overflow_rd_sc(h)             <= (others => '0');
    dsp_we_word(h)                <= (others => '0');
    dsp_sci_req(h)               <= (others => '0');
    dsp_sci_we(h)                <= (others => '0');
    dsp_sc_write_addr(h)          <= (others => '0');
    dsp_sc_read_addr(h)           <= (others => (others => '0'));
    dsp_to_sc(h)                 <= (others => (others => '0'));

    if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
      case state_DSP(h) is

        when dsp_init =>

          overflow_rs1_sc(h) <= std_logic_vector('0' & unsigned(RS1_Data_IE(Addr_Width -1 downto 0)) + unsigned(MVSIZE(harc_EXEC)) -1);
          overflow_rs2_sc(h) <= std_logic_vector('0' & unsigned(RS2_Data_IE(Addr_Width -1 downto 0)) + unsigned(MVSIZE(harc_EXEC)) -1);
          overflow_rd_sc(h)  <= std_logic_vector('0' & unsigned(RD_Data_IE(Addr_Width  -1 downto 0)) + unsigned(MVSIZE(harc_EXEC)) -1);
          if MVSIZE(harc_EXEC) = (0 to Addr_Width => '0') then
            null;
          elsif MVSIZE(harc_EXEC)(1 downto 0) /= "00" and MVTYPE(harc_EXEC)(3 downto 2) = "10" then  -- Set exception if the number of bytes
are not divisible by four
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= ILLEGAL_VECTOR_SIZE_EXCEPT_CODE;
          elsif MVSIZE(harc_EXEC)(0) /= '0' and MVTYPE(harc_EXEC)(3 downto 2) = "01" then           -- Set exception if the number of bytes are not
divisible by two
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= ILLEGAL_VECTOR_SIZE_EXCEPT_CODE;
          elsif (rs1_to_sc  = "100" and vec_read_rs1_ID = '1') or
                                  (rs2_to_sc  = "100" and vec_read_rs2_ID = '1') or
                                   rd_to_sc   = "100" then     -- Set exception for non scratchpad access
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= ILLEGAL_ADDRESS_EXCEPT_CODE;
          elsif rs1_to_sc = rs2_to_sc and vec_read_rs1_ID = '1' and vec_read_rs2_ID = '1' then              -- Set exception for same read access
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= READ_SAME_SCARTCHPAD_EXCEPT_CODE;
          elsif (overflow_rs1_sc(h)(Addr_Width) = '1' and vec_read_rs1_ID = '1') or (overflow_rs2_sc(h)(Addr_Width) = '1' and  vec_read_rs2_ID = '1')
then -- Set exception if reading overflows the scratchpad's address
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= SCRATCHPAD_OVERFLOW_EXCEPT_CODE;
          elsif overflow_rd_sc(h)(Addr_Width) = '1'  and vec_write_rd_ID = '1' then          -- Set exception if reading overflows the scratchpad's address,
scalar writes are excluded
            dsp_except_condition_wires(h) := '1';
            dsp_taken_branch_wires(h)     := '1';
            dsp_except_data_wire(h) <= SCRATCHPAD_OVERFLOW_EXCEPT_CODE;
          else
            if halt_hart(h) = '0' then
              nextstate_DSP(h) <= dsp_exec;
            else
              nextstate_DSP(h) <= dsp_halt_hart;
            end if;
```

```
            busy_DSP_internal_wires := '1';
          end if;

          if rs1_to_sc /= "100" and spm_rs1 = '1' and halt_hart(h) = '0' then
            dsp_sci_req(h)(to_integer(unsigned(rs1_to_sc))) <= '1';
            dsp_to_sc(h)(to_integer(unsigned(rs1_to_sc)))(0) <= '1';
            dsp_sc_read_addr(h)(0) <= RS1_Data_IE(Addr_Width-1 downto 0);
          end if;
          if rs2_to_sc /= "100" and spm_rs2 = '1' and rs1_to_Sc /= rs2_to_sc and halt_hart(h) = '0' then   -- Do not send a read request if the second
operand accesses the same spm as the first,
            dsp_sci_req(h)(to_integer(unsigned(rs2_to_sc))) <= '1';
            dsp_to_sc(h)(to_integer(unsigned(rs2_to_sc)))(1) <= '1';
            dsp_sc_read_addr(h)(1) <= RS2_Data_IE(Addr_Width-1 downto 0);
          end if;

        when dsp_halt_hart =>

          if halt_hart(h) = '0' then
            nextstate_DSP(h) <= dsp_exec;
          else
            nextstate_DSP(h) <= dsp_halt_hart;
          end if;
          busy_DSP_internal_wires := '1';

        when dsp_exec =>

          if (dsp_sci_wr_gnt(h) = '0' and wb_ready(h) = '1') then
            halt_dsp(h) <= '1';
            recover_state_wires(h) <= '1';
          elsif unsigned(MVSIZE_WRITE(h)) <= SIMD_RD_BYTES(h) then
            recover_state_wires(h) <= '0';
          end if;

          if vec_write_rd_DSP(h) = '1' and  dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) = '1' then
            if unsigned(MVSIZE_WRITE(h)) >= (SIMD)*4+1 then  --
              dsp_we_word(h) <= (others => '1');
            elsif  unsigned(MVSIZE_WRITE(h)) >= 1 then
              for i in 0 to SIMD-1 loop
                if i <= to_integer(unsigned(MVSIZE_WRITE(h))-1)/4 then -- Four because of the number of bytes per word
                  if to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))/4 + i) < SIMD then
                    dsp_we_word(h)(to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))/4 + i)) <= '1';
                  elsif to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))/4 + i) >= SIMD then
                    dsp_we_word(h)(to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))/4 + i - SIMD)) <= '1';
                  end if;
                end if;
              end loop;
            end if;
          elsif vec_write_rd_DSP(h) = '0' and  dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) = '1' then
            dsp_we_word(h)(to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))/4)) <= '1';
          end if;

          if decoded_instruction_DSP_lat(h)(KBCAST_bit_position)  = '1' then
            -- KBCAST signals are handeled here
            if MVSIZE_WRITE(h) > (0 to Addr_Width => '0') then
              nextstate_DSP(h) <= dsp_exec;
              busy_DSP_internal_wires := '1';
            end if;
            wb_ready(h) <= '1';
            dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
            dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
          end if;

          if decoded_instruction_DSP_lat(h)(KVCP_bit_position)  = '1' then
            -- KVCP signals are handeled here
            if adder_stage_3_en(h) = '1' then
              wb_ready(h) <= '1';
            elsif recover_state(h) = '1' then
              wb_ready(h) <= '1';
            end if;
            if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
              dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
              dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))  <= '1';
              dsp_sc_read_addr(h)(0) <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
            end if;
            if MVSIZE_WRITE(h) > (0 to Addr_Width => '0') then
              nextstate_DSP(h) <= dsp_exec;
              busy_DSP_internal_wires := '1';
            end if;
```

```
            if wb_ready(h) = '1' then
              dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
              dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
            end if;
          end if;

          if decoded_instruction_DSP_lat(h)(KRELU_bit_position) = '1' then
            -- KRELU signals are handeled here
            if relu_stage_2_en(h) = '1' then
              wb_ready(h) <= '1';
            elsif recover_state(h) = '1' then
              wb_ready(h) <= '1';
            end if;
            if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
              dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
              dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h)))) <= '1';
              dsp_sc_read_addr(h)(0) <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
            end if;
            if MVSIZE_WRITE(h) > (0 to Addr_Width => '0') then
              nextstate_DSP(h) <= dsp_exec;
              busy_DSP_internal_wires := '1';
            end if;
            if wb_ready(h) = '1' then
              dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
              dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
            end if;
          end if;

          if decoded_instruction_DSP_lat(h)(KSRAV_bit_position) = '1' or
            decoded_instruction_DSP_lat(h)(KSRLV_bit_position) = '1' then
            -- KSRAV signals are handeled here
            if shifter_stage_3_en(h) = '1' then
              wb_ready(h) <= '1';
            elsif recover_state(h) = '1' then
              wb_ready(h) <= '1';
            end if;
            if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
              dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
              dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h)))) <= '1';
              dsp_sc_read_addr(h)(0) <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
            end if;
            if MVSIZE_WRITE(h) > (0 to Addr_Width => '0') then
              nextstate_DSP(h) <= dsp_exec;
              busy_DSP_internal_wires := '1';
            end if;
            if wb_ready(h) = '1' then
              dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
              dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
            end if;
          end if;

          if decoded_instruction_DSP_lat(h)(KADDV_bit_position) = '1' or
            decoded_instruction_DSP_lat(h)(KSUBV_bit_position) = '1' then
            -- KADDV and KSUBV signals are handeled here
            if adder_stage_3_en(h) = '1' then
              wb_ready(h) <= '1';
            elsif recover_state(h) = '1' then
              wb_ready(h) <= '1';
            end if;
            if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
              dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
              dsp_to_sc(h)(to_integer(unsigned(dsp_rs2_to_sc(h))))(1) <= '1';
              dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h)))) <= '1';
              dsp_sci_req(h)(to_integer(unsigned(dsp_rs2_to_sc(h)))) <= '1';
              dsp_sc_read_addr(h)(0) <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
              dsp_sc_read_addr(h)(1) <= RS2_Data_IE_lat(h)(Addr_Width - 1 downto 0);
            end if;
            if MVSIZE_WRITE(h) > (0 to Addr_Width => '0') then
              nextstate_DSP(h) <= dsp_exec;
              busy_DSP_internal_wires := '1';
            end if;
            if wb_ready(h) = '1' then
              dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
              dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
            end if;
          end if;

          if decoded_instruction_DSP_lat(h)(KVRED_bit_position) = '1' or
```

```
668               decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1' or
669               decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1' then
670             -- KDOTP signals are handeled here
671             if accum_stage_3_en(h) = '1' then
672               wb_ready(h) <= '1';
673             elsif recover_state(h) = '1' then
674               wb_ready(h) <= '1';
675             end if;
676             if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
677               if vec_read_rs2_DSP(h) = '1' then
678                 dsp_sci_req(h)(to_integer(unsigned(dsp_rs2_to_sc(h)))) <= '1';
679                 dsp_to_sc(h)(to_integer(unsigned(dsp_rs2_to_sc(h))))(1) <= '1';
680                 dsp_sc_read_addr(h)(1)  <= RS2_Data_IE_lat(h)(Addr_Width - 1 downto 0);
681               end if;
682               dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h)))) <= '1';
683               dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
684               dsp_sc_read_addr(h)(0)  <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
685               nextstate_DSP(h) <= dsp_exec;
686               busy_DSP_internal_wires := '1';
687             elsif MVSIZE_WRITE(h) = (0 to Addr_Width => '0') then
688               nextstate_DSP(h) <= dsp_init;
689             else
690               nextstate_DSP(h) <= dsp_exec;
691               busy_DSP_internal_wires := '1';
692             end if;
693             if wb_ready(h) = '1' then
694               dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h))))    <= '1';
695               dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
696             end if;
697           end if;
698
699           if decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1' or
700             decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1' or
701             decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1' or
702             decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position) = '1' or
703             decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position) = '1' then
704             -- KMUL signals are handeled here
705             if mul_stage_3_en(h) = '1' or  adder_stage_3_en(h) = '1' then
706               wb_ready(h) <= '1';
707             elsif recover_state(h) = '1' then
708               wb_ready(h) <= '1';
709             end if;
710             if MVSIZE_READ(h) > (0 to Addr_Width => '0') then
711               dsp_sci_req(h)(to_integer(unsigned(dsp_rs1_to_sc(h)))) <= '1';
712               if rf_rs2(h) = '0' then -- if the scalar does not come from the regfile
713                 dsp_sci_req(h)(to_integer(unsigned(dsp_rs2_to_sc(h)))) <= '1';
714                 dsp_to_sc(h)(to_integer(unsigned(dsp_rs2_to_sc(h))))(1) <= '1';
715                 dsp_sc_read_addr(h)(1)  <= RS2_Data_IE_lat(h)(Addr_Width - 1 downto 0);
716               end if;
717               dsp_to_sc(h)(to_integer(unsigned(dsp_rs1_to_sc(h))))(0) <= '1';
718               dsp_sc_read_addr(h)(0)  <= RS1_Data_IE_lat(h)(Addr_Width - 1 downto 0);
719               nextstate_DSP(h) <= dsp_exec;
720               busy_DSP_internal_wires := '1';
721             elsif MVSIZE_WRITE(h) = (0 to Addr_Width => '0') then
722               nextstate_DSP(h) <= dsp_init;
723             else
724               nextstate_DSP(h) <= dsp_exec;
725               busy_DSP_internal_wires := '1';
726             end if;
727             if wb_ready(h) = '1' then
728               dsp_sci_we(h)(to_integer(unsigned(dsp_rd_to_sc(h)))) <= '1';
729               dsp_sc_write_addr(h) <= RD_Data_IE_lat(h);
730             end if;
731           end if;
732
733         when others =>
734           null;
735       end case;
736     end if;
737
738   busy_DSP_internal(h)    <= busy_DSP_internal_wires;
739   dsp_except_condition(h) <= dsp_except_condition_wires(h);
740   dsp_taken_branch(h)     <= dsp_taken_branch_wires(h);
741
742 end process;
743
744 fsm_DSP_pipeline_controller : process(clk_i, rst_ni)
745 begin
```

```
746         if rst_ni = '0' then
747           dsp_data_gnt_i_lat(h)    <= '0';
748           adder_stage_1_en(h)      <= '0';
749           adder_stage_2_en(h)      <= '0';
750           adder_stage_3_en(h)      <= '0';
751           shifter_stage_1_en(h)    <= '0';
752           shifter_stage_2_en(h)    <= '0';
753           mul_stage_1_en(h)        <= '0';
754           mul_stage_2_en(h)        <= '0';
755           mul_stage_3_en(h)        <= '0';
756           accum_stage_1_en(h)      <= '0';
757           accum_stage_2_en(h)      <= '0';
758           accum_stage_3_en(h)      <= '0';
759           relu_stage_1_en(h)       <= '0';
760           relu_stage_2_en(h)       <= '0';
761                   busy_DSP_internal_lat(h) <= '0';
762           state_DSP(h)             <= dsp_init;
763         elsif rising_edge(clk_i) then
764           dsp_data_gnt_i_lat(h) <= dsp_data_gnt_i(h);
765           adder_stage_1_en(h)   <= dsp_data_gnt_i_lat(h) and add_en(h);
766           adder_stage_2_en(h)   <= adder_stage_1_en(h);
767           adder_stage_3_en(h)   <= adder_stage_2_en(h);
768           mul_stage_1_en(h)     <= dsp_data_gnt_i_lat(h) and mul_en(h);
769           mul_stage_2_en(h)     <= mul_stage_1_en(h);
770           mul_stage_3_en(h)     <= mul_stage_2_en(h);
771           relu_stage_1_en(h)    <= dsp_data_gnt_i_lat(h) and relu_en(h);
772           relu_stage_2_en(h)    <= relu_stage_1_en(h);
773           accum_stage_2_en(h)   <= accum_stage_1_en(h);
774           accum_stage_3_en(h)   <= accum_stage_2_en(h);
775           if dotpps(h) = '1' then
776             shifter_stage_1_en(h) <= mul_stage_2_en(h);
777             shifter_stage_2_en(h) <= shifter_stage_1_en(h);
778             accum_stage_1_en(h)   <= shifter_stage_2_en(h);
779           elsif dotp(h) = '1' then
780             accum_stage_1_en(h)   <= mul_stage_2_en(h);
781           else
782             shifter_stage_1_en(h) <= dsp_data_gnt_i_lat(h) and shift_en(h);
783             shifter_stage_2_en(h) <= shifter_stage_1_en(h);
784             shifter_stage_3_en(h) <= shifter_stage_2_en(h);
785             accum_stage_1_en(h)   <= dsp_data_gnt_i_lat(h) and accum_en(h);
786           end if;
787           halt_dsp_lat(h)          <= halt_dsp(h);
788           state_DSP(h)             <= nextstate_DSP(h);
789           busy_DSP_internal_lat(h) <= busy_DSP_internal(h);
790           SIMD_RD_BYTES(h)         <= SIMD_RD_BYTES_wire(h);
791           dsp_except_data(h)       <= dsp_except_data_wire(h);
792         end if;
793       end process;
794
795       DSP_FU_ENABLER_SYNC : process(clk_i, rst_ni)
796       begin
797         if rst_ni = '0' then
798           shift_en(h)         <= '0';
799           add_en(h)           <= '0';
800           relu_en(h)          <= '0';
801           accum_en(h)         <= '0';
802           mul_en(h)           <= '0';
803           add_en_pending(h)   <= '0';
804           shift_en_pending(h) <= '0';
805           mul_en_pending(h)   <= '0';
806           accum_en_pending(h) <= '0';
807           relu_en_pending(h)  <= '0';
808         elsif rising_edge(clk_i) then
809           shift_en(h)         <= shift_en_wire(h);
810           add_en(h)           <= add_en_wire(h);
811           relu_en(h)          <= relu_en_wire(h);
812           accum_en(h)         <= accum_en_wire(h);
813           mul_en(h)           <= mul_en_wire(h);
814           add_en_pending(h)   <= add_en_pending_wire(h);
815           shift_en_pending(h) <= shift_en_pending_wire(h);
816           mul_en_pending(h)   <= mul_en_pending_wire(h);
817           accum_en_pending(h) <= accum_en_pending_wire(h);
818           relu_en_pending(h)  <= relu_en_pending_wire(h);
819         end if;
820
821       end process;
822
823     end generate DSP_replicated;
```

```
824   FU_HANDLER_MC : if multithreaded_accl_en = 0 generate
825     DSP_FU_ENABLER_comb : process(all)
826     begin
827       for h in accl_range loop
828         shift_en_wire(h) <= shift_en(h);
829         add_en_wire(h)   <= add_en(h);
830         relu_en_wire(h)  <= relu_en(h);
831         accum_en_wire(h) <= accum_en(h);
832         mul_en_wire(h)   <= mul_en(h);
833         halt_hart(h)     <= '0';
834
835         if add_en(h) = '1' and busy_DSP_internal(h) = '0' then
836           add_en_wire(h) <= '0';
837         end if;
838         if mul_en(h) = '1' and busy_DSP_internal(h) = '0' then
839           mul_en_wire(h) <= '0';
840         end if;
841         if shift_en(h) = '1' and busy_DSP_internal(h) = '0' then
842           shift_en_wire(h) <= '0';
843         end if;
844         if accum_en(h) = '1' and busy_DSP_internal(h) = '0' then
845           accum_en_wire(h) <= '0';
846         end if;
847         if relu_en(h) = '1' and busy_DSP_internal(h) = '0' then
848           relu_en_wire(h) <= '0';
849         end if;
850
851         if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
852
853           case state_DSP(h) is
854
855             when dsp_init =>
856
857               -- Set signals to enable correct virtual parallelism operation
858               if decoded_instruction_DSP(KADDV_bit_position)    = '1' or
859                  decoded_instruction_DSP(KSVADDSC_bit_position) = '1' or
860                  decoded_instruction_DSP(KSVADDRF_bit_position) = '1' or
861                  decoded_instruction_DSP(KSUBV_bit_position)    = '1' or
862                  decoded_instruction_DSP(KVCP_bit_position)     = '1' then
863                 add_en_wire(h) <= '1';
864               elsif decoded_instruction_DSP(KDOTP_bit_position) = '1' then
865                 mul_en_wire(h)   <= '1';
866                 accum_en_wire(h) <= '1';
867               elsif decoded_instruction_DSP(KDOTPPS_bit_position) = '1' then
868                 mul_en_wire(h)   <= '1';
869                 shift_en_wire(h) <= '1';
870                 accum_en_wire(h) <= '1';
871               elsif decoded_instruction_DSP(KVRED_bit_position) = '1' then
872                 accum_en_wire(h) <= '1';
873               elsif decoded_instruction_DSP(KSVMULRF_bit_position) = '1' or
874                     decoded_instruction_DSP(KSVMULSC_bit_position) = '1' or
875                     decoded_instruction_DSP(KVMUL_bit_position)    = '1' then
876                 mul_en_wire(h) <= '1';
877               elsif decoded_instruction_DSP(KSRAV_bit_position) = '1' or
878                     decoded_instruction_DSP(KSRLV_bit_position) = '1' then
879                 shift_en_wire(h) <= '1';
880               elsif decoded_instruction_DSP(KRELU_bit_position) = '1' then
881                 relu_en_wire(h) <= '1';
882               end if;
883             when others =>
884               null;
885           end case;
886         end if;
887       end loop;
888     end process;
889   end generate FU_HANDLER_MC;
890
891   FU_HANDLER_MT : if multithreaded_accl_en = 1 generate
892     DSP_FU_ENABLER_comb : process(all)
893     begin
894
895       for h in accl_range loop
896
897         shift_en_wire(h)          <= shift_en(h);
898         add_en_wire(h)            <= add_en(h);
899         relu_en_wire(h)           <= relu_en(h);
900         accum_en_wire(h)          <= accum_en(h);
901         mul_en_wire(h)            <= mul_en(h);
```

```
902         add_en_pending_wire(h)        <= add_en_pending(h);
903         shift_en_pending_wire(h)      <= shift_en_pending(h);
904         mul_en_pending_wire(h)        <= mul_en_pending(h);
905         accum_en_pending_wire(h)      <= accum_en_pending(h);
906         relu_en_pending_wire(h)       <= relu_en_pending(h);
907         fu_req(h)                     <= (others => '0');
908         halt_hart(h)                  <= '0';
909
910
911         if add_en(h) = '1' and busy_DSP_internal(h) = '0' then
912           add_en_wire(h) <= '0';
913         end if;
914         if mul_en(h) = '1' and busy_DSP_internal(h) = '0' then
915           mul_en_wire(h) <= '0';
916         end if;
917         if shift_en(h) = '1' and busy_DSP_internal(h) = '0' then
918           shift_en_wire(h) <= '0';
919         end if;
920         if accum_en(h) = '1' and busy_DSP_internal(h) = '0' then
921           accum_en_wire(h) <= '0';
922         end if;
923         if relu_en(h) = '1' and busy_DSP_internal(h) = '0' then
924           relu_en_wire(h) <= '0';
925         end if;
926
927         if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
928
929           case state_DSP(h) is
930
931             when dsp_init =>
932
933               -- Set signals to enable correct virtual parallelism operation
934               if decoded_instruction_DSP(KADDV_bit_position)    = '1' or
935                 decoded_instruction_DSP(KSVADDSC_bit_position) = '1' or
936                 decoded_instruction_DSP(KSVADDRF_bit_position) = '1' or
937                 decoded_instruction_DSP(KSUBV_bit_position)    = '1' or
938                 decoded_instruction_DSP(KVCP_bit_position)     = '1' then
939                 if busy_add = '0' and add_en_pending = (accl_range => '0') then
940                   add_en_wire(h) <= '1';
941                 else
942                   add_en_pending_wire(h) <= '1';
943                   halt_hart(h) <= '1';
944                   fu_req(h)(0) <= '1';
945                 end if;
946               elsif decoded_instruction_DSP(KDOTP_bit_position) = '1' then
947                 if busy_mul = '0' and busy_acc = '0' and mul_en_pending = (accl_range => '0') and accum_en_pending = (accl_range => '0') then
948                   mul_en_wire(h)   <= '1';
949                   accum_en_wire(h) <= '1';
950                 else
951                   mul_en_pending_wire(h)   <= '1';
952                   accum_en_pending_wire(h) <= '1';
953                   halt_hart(h) <= '1';
954                   fu_req(h)(2) <= '1';
955                   fu_req(h)(3) <= '1';
956                 end if;
957               elsif decoded_instruction_DSP(KDOTPPS_bit_position) = '1' then
958                 if busy_mul = '0' and busy_acc = '0' and busy_shf = '0'  and mul_en_pending = (accl_range => '0') and accum_en_pending = (accl_range =>
959     '0') and shift_en_pending = (accl_range => '0') then
960                   mul_en_wire(h)   <= '1';
961                   shift_en_wire(h) <= '1';
962                   accum_en_wire(h) <= '1';
963                 else
964                   mul_en_pending_wire(h)   <= '1';
965                   shift_en_pending_wire(h) <= '1';
966                   accum_en_pending_wire(h) <= '1';
967                   halt_hart(h) <= '1';
968                   fu_req(h)(2) <= '1';
969                   fu_req(h)(1) <= '1';
970                   fu_req(h)(3) <= '1';
971                 end if;
972               elsif decoded_instruction_DSP(KVRED_bit_position) = '1' then
973                 if busy_acc = '0' and accum_en_pending = (accl_range => '0') then
974                   accum_en_wire(h) <= '1';
975                 else
976                   accum_en_pending_wire(h) <= '1';
977                   halt_hart(h) <= '1';
978                   fu_req(h)(3) <= '1';
979                 end if;
```

```
            elsif decoded_instruction_DSP(KSVMULRF_bit_position) = '1' or
                  decoded_instruction_DSP(KSVMULSC_bit_position) = '1' or
                  decoded_instruction_DSP(KVMUL_bit_position)    = '1' then
              if busy_mul = '0' and mul_en_pending = (accl_range => '0') then
                mul_en_wire(h) <= '1';
              else
                mul_en_pending_wire(h) <= '1';
                halt_hart(h) <= '1';
                fu_req(h)(2) <= '1';
              end if;
            elsif decoded_instruction_DSP(KSRAV_bit_position) = '1' or
                  decoded_instruction_DSP(KSRLV_bit_position) = '1' then
              if busy_shf = '0' and shift_en_pending = (accl_range => '0') then
                shift_en_wire(h) <= '1';
              else
                shift_en_pending_wire(h) <= '1';
                halt_hart(h) <= '1';
                fu_req(h)(1) <= '1';
              end if;
            elsif decoded_instruction_DSP(KRELU_bit_position) = '1' then
              if busy_rel = '0' and relu_en_pending = (accl_range => '0') then
                relu_en_wire(h) <= '1';
              else
                relu_en_pending_wire(h) <= '1';
                halt_hart(h) <= '1';
                fu_req(h)(4) <= '1';
              end if;
            end if;

          when dsp_halt_hart =>

            if fu_gnt(h)(0) = '1' then
              add_en_wire(h) <= '1';
              add_en_pending_wire(h) <= '0';
            elsif add_en_pending(h) = '1' and fu_gnt(h)(0) = '0'  then
              halt_hart(h) <= '1';
            end if;

            if fu_gnt(h)(1) = '1' then
              shift_en_wire(h) <= '1';
              shift_en_pending_wire(h) <= '0';
            elsif shift_en_pending(h) = '1' and fu_gnt(h)(1) = '0' then
              halt_hart(h) <= '1';
            end if;

            if fu_gnt(h)(2) = '1' then
              mul_en_wire(h) <= '1';
              mul_en_pending_wire(h) <= '0';
            elsif mul_en_pending(h) = '1' and fu_gnt(h)(2) = '0'  then
              halt_hart(h) <= '1';
            end if;

            if fu_gnt(h)(3) = '1' then
              accum_en_wire(h) <= '1';
              accum_en_pending_wire(h) <= '0';
            elsif accum_en_pending(h) = '1' and fu_gnt(h)(3) = '0'  then
              halt_hart(h) <= '1';
            end if;

            if fu_gnt(h)(4) = '1' then
              relu_en_wire(h) <= '1';
              relu_en_pending_wire(h) <= '0';
            elsif relu_en_pending(h) = '1' and fu_gnt(h)(4) = '0'  then
              halt_hart(h) <= '1';
            end if;

          when others =>
            null;
        end case;
      end if;
    end loop;
  end process;

  FU_Issue_Buffer_sync : process(clk_i, rst_ni)
  begin
    if rst_ni = '0' then
      fu_rd_ptr  <= (others => (others => '0'));
      fu_wr_ptr  <= (others => (others => '0'));
```

```
      fu_gnt    <= (others => (others => '0'));
    elsif rising_edge(clk_i) then
      fu_gnt <= fu_gnt_wire;
      for h in accl_range loop
        for i in 0 to 4 loop  -- Loop index 'i' is for the total number of different functional units (regardless what SIMD config is set)
          if fu_req(h)(i) = '1' then  -- if a reservation was made, to use a functional unit
            --to_integer(unsigned(fu_issue_buffer(i)(to_integer(unsigned(fu_wr_ptr(i)))))) <= h;  -- store the thread_ID in its corresponding buffer at the fu_wr_ptr position
            --fu_issue_buffer(to_integer(unsigned(fu_wr_ptr(i))))(i) <= std_logic_vector(unsigned(h));  -- store the thread_ID in its corresponding buffer at the fu_wr_ptr position
            fu_issue_buffer(i)(to_integer(unsigned(fu_wr_ptr(i))))  <= std_logic_vector(to_unsigned(h,TPS_CEIL));
            if unsigned(fu_wr_ptr(i)) = THREAD_POOL_SIZE - 2 then -- increment the pointer wr logic
              fu_wr_ptr(i) <= (others => '0');
            else
              fu_wr_ptr(i) <= std_logic_vector(unsigned(fu_wr_ptr(i)) + 1);
            end if;
          end if;
          case state_DSP(h) is
            when dsp_halt_hart =>
              if fu_gnt_en(h)(i) = '1' then
                if unsigned(fu_rd_ptr(i)) = THREAD_POOL_SIZE - 2 then  -- increment the read pointer
                  fu_rd_ptr(i) <= (others => '0');
                else
                  fu_rd_ptr(i) <= std_logic_vector(unsigned(fu_rd_ptr(i)) + 1);
                end if;
              end if;
            when others =>
              null;
          end case;
        end loop;
      end loop;
    end if;
  end process;

  FU_Issue_Buffer_comb : process(all)
  begin
    for h in accl_range loop
      fu_gnt_wire(h) <= (others => '0');
      fu_gnt_en(h)   <= (others => '0');
      if add_en_pending_wire(h) = '1' and busy_add_wire = '0' then
        fu_gnt_en(h)(0) <= '1';
      end if;
      if shift_en_pending_wire(h) = '1' and busy_shf_wire = '0' then
        fu_gnt_en(h)(1) <= '1';
      end if;
      if mul_en_pending_wire(h) = '1' and busy_mul_wire = '0' then
        fu_gnt_en(h)(2) <= '1';
      end if;
      if accum_en_pending_wire(h) = '1' and busy_acc_wire = '0' then
        fu_gnt_en(h)(3) <= '1';
      end if;
      if relu_en_pending_wire(h) = '1' and busy_rel_wire = '0' then
        fu_gnt_en(h)(4) <= '1';
      end if;
      case state_DSP(h) is
        when dsp_halt_hart =>
          for i in 0 to 4 loop
            if fu_gnt_en(h)(i) = '1' then
              fu_gnt_wire(to_integer(unsigned(fu_issue_buffer(i)(to_integer(unsigned(fu_rd_ptr(i)))))))(i) <= '1'; -- give a grant to fu_gnt(h)(i), such that the 'h' index points to the thread in "fu_issue_buffer"
            end if;
          end loop;
        when others =>
          null;
      end case;
    end loop;
  end process;


  DSP_BUSY_FU_SYNC : process(clk_i, rst_ni)
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      busy_add    <= busy_add_wire;
      busy_mul    <= busy_mul_wire;
      busy_shf    <= busy_shf_wire;
      busy_acc    <= busy_acc_wire;
      busy_rel    <= busy_rel_wire;
```

```
    end if;
  end process;

end generate FU_HANDLER_MT;

busy_add_wire <= '1' when multithreaded_accl_en = 1 and add_en_wire   /= (accl_range => '0') else '0';
busy_mul_wire <= '1' when multithreaded_accl_en = 1 and mul_en_wire   /= (accl_range => '0') else '0';
busy_shf_wire <= '1' when multithreaded_accl_en = 1 and shift_en_wire /= (accl_range => '0') else '0';
busy_acc_wire <= '1' when multithreaded_accl_en = 1 and accum_en_wire /= (accl_range => '0') else '0';
busy_rel_wire <= '1' when multithreaded_accl_en = 1 and relu_en_wire  /= (accl_range => '0') else '0';

MULTICORE_OUT_MAPPER : if multithreaded_accl_en = 0 generate
MAPPER_replicated : for h in fu_range generate

  MAPPING_OUT_UNIT_comb : process(all)
  begin
     dsp_sc_data_write_wire_int(h)  <= (others => '0');
     dsp_sc_data_write_wire(h)      <= dsp_sc_data_write_wire_int(h);
     SIMD_RD_BYTES_wire(h)          <= SIMD*(Data_Width/8);

     if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
       case state_DSP(h) is
         when dsp_init =>

           -- Set signals to enable correct virtual parallelism operation
           if (decoded_instruction_DSP(KDOTP_bit_position)    = '1'  or
               decoded_instruction_DSP(KDOTPPS_bit_position)  = '1'  or
               decoded_instruction_DSP(KVRED_bit_position)    = '1'  or
               decoded_instruction_DSP(KSVMULRF_bit_position) = '1'  or
               decoded_instruction_DSP(KVMUL_bit_position)    = '1'  or
               decoded_instruction_DSP(KSVMULSC_bit_position) = '1') and
               MVTYPE(h)(3 downto 2) = "00" then
             SIMD_RD_BYTES_wire(h) <= SIMD*(Data_Width/8)/2;
           end if;

         when dsp_exec =>

          -- Set signals to enable correct virtual parallelism operation
           if (decoded_instruction_DSP_lat(h)(KDOTP_bit_position)    = '1'  or
               decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position)  = '1'  or
               decoded_instruction_DSP_lat(h)(KVRED_bit_position)    = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
               decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
               (MVTYPE_DSP(h) = "00") then
             SIMD_RD_BYTES_wire(h) <= SIMD*(Data_Width/8)/2;
           end if;

           if decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1' or
              decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1' or
              decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1' or
              decoded_instruction_DSP_lat(h)(KVRED_bit_position)   = '1' then
             dsp_sc_data_write_wire_int(h)(31 downto 0) <= dsp_out_accum_results(h);  -- AAA add a mask in order to store the lower half word when
16-bit or entire word when 32-bit
           end if;

           if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)     = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
              MVTYPE_DSP(h) = "00" then
             for i in 0 to 2*SIMD-1 loop
               dsp_sc_data_write_wire_int(h)(7+8*(i) downto 8*(i)) <= dsp_out_mul_results(h)(7+8*(2*i) downto 8*(2*i));
             end loop;
           end if;

           if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
               decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
              (MVTYPE_DSP(h) = "01" or  MVTYPE_DSP(h) = "10") then
             dsp_sc_data_write_wire_int(h) <= dsp_out_mul_results(h);
           end if;

           if decoded_instruction_DSP_lat(h)(KSRAV_bit_position)   = '1' or
              decoded_instruction_DSP_lat(h)(KSRLV_bit_position)   = '1' then
             dsp_sc_data_write_wire_int(h)  <= dsp_out_shifter_results(h);
           end if;

           if decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position)   = '1' or
```

```
              decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position)   = '1' or
              decoded_instruction_DSP_lat(h)(KADDV_bit_position)      = '1' or
              decoded_instruction_DSP_lat(h)(KSUBV_bit_position)      = '1' or
              decoded_instruction_DSP_lat(h)(KVCP_bit_position)       = '1' then
             dsp_sc_data_write_wire_int(h) <= dsp_out_adder_results(h);
            end if;


            if decoded_instruction_DSP_lat(h)(KRELU_bit_position)  = '1' then
              dsp_sc_data_write_wire_int(h)   <= dsp_out_relu_results(h);
            end if;

            if    decoded_instruction_DSP_lat(h)(KBCAST_bit_position) = '1'  and MVTYPE_DSP(h) = "10" then
              for i in 0 to SIMD-1 loop
                dsp_sc_data_write_wire_int(h)(31+32*(i) downto 32*(i)) <= RS1_Data_IE_lat(h);
              end loop;
            elsif decoded_instruction_DSP_lat(h)(KBCAST_bit_position) = '1'  and MVTYPE_DSP(h) = "01" then
              for i in 0 to 2*SIMD-1 loop
                dsp_sc_data_write_wire_int(h)(15+16*(i) downto 16*(i)) <= RS1_Data_IE_lat(h)(15 downto 0);
              end loop;
            elsif decoded_instruction_DSP_lat(h)(KBCAST_bit_position)  = '1'  and MVTYPE_DSP(h) = "00" then
              for i in 0 to 4*SIMD-1 loop
                dsp_sc_data_write_wire_int(h)(7+8*(i)   downto 8*(i))  <= RS1_Data_IE_lat(h)(7 downto 0);
              end loop;
            end if;

            if halt_dsp(h) = '0' and halt_dsp_lat(h) = '1' then
              dsp_sc_data_write_wire(h) <= dsp_sc_data_write_int(h);
            end if;
           when others =>
            null;
         end case;
        end if;
    end process;

end generate;
end generate;

MULTITHREAD_OUT_MAPPER : if multithreaded_accl_en = 1 generate
  MAPPING_OUT_UNIT_comb : process(all)
  begin
    for h in 0 to (ACCL_NUM - FU_NUM) loop
      dsp_sc_data_write_wire_int(h)  <= (others => '0');
      dsp_sc_data_write_wire(h)      <= dsp_sc_data_write_wire_int(h);
      SIMD_RD_BYTES_wire(h) <= SIMD*(Data_Width/8);

      if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
        case state_DSP(h) is
          when dsp_init =>

            -- Set signals to enable correct virtual parallelism operation
            if (decoded_instruction_DSP(KDOTP_bit_position)    = '1'  or
                decoded_instruction_DSP(KDOTPPS_bit_position)  = '1'  or
                decoded_instruction_DSP(KVRED_bit_position)    = '1'  or
                decoded_instruction_DSP(KSVMULRF_bit_position) = '1'  or
                decoded_instruction_DSP(KVMUL_bit_position)    = '1'  or
                decoded_instruction_DSP(KSVMULSC_bit_position) = '1') and
                MVTYPE(h)(3 downto 2) = "00" then
              SIMD_RD_BYTES_wire(h) <= SIMD*(Data_Width/8)/2;
            end if;

          when dsp_exec =>

           -- Set signals to enable correct virtual parallelism operation
            if (decoded_instruction_DSP_lat(h)(KDOTP_bit_position)    = '1'  or
                decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position)  = '1'  or
                decoded_instruction_DSP_lat(h)(KVRED_bit_position)    = '1'  or
                decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
                decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
                decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
                MVTYPE_DSP(h) = "00" then
              SIMD_RD_BYTES_wire(h) <= SIMD*(Data_Width/8)/2;
            end if;

            if decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1' or
              decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1' or
              decoded_instruction_DSP_lat(h)(KVRED_bit_position)   = '1' then
```

```
            dsp_sc_data_write_wire_int(h)(31 downto 0) <= dsp_out_accum_results(0);  -- AAA add a mask in order to store the lower half word when 16-bit or entire word when 32-bit
          end if;

          if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
             decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
             decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
             MVTYPE_DSP(h) = "00" then
            for i in 0 to 2*SIMD-1 loop
              dsp_sc_data_write_wire_int(h)(7+8*(i) downto 8*(i)) <= dsp_out_mul_results(0)(7+8*(2*i) downto 8*(2*i));
            end loop;
          end if;

          if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
             decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
             decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
            (MVTYPE_DSP(h) = "01" or MVTYPE_DSP(h) = "10") then
            dsp_sc_data_write_wire_int(h) <= dsp_out_mul_results(0);
          end if;

          if decoded_instruction_DSP_lat(h)(KSRAV_bit_position)   = '1' or
             decoded_instruction_DSP_lat(h)(KSRLV_bit_position)   = '1' then
            dsp_sc_data_write_wire_int(h)  <= dsp_out_shifter_results(0);
          end if;

          if decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position)   = '1' or
             decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position)   = '1' or
             decoded_instruction_DSP_lat(h)(KADDV_bit_position)      = '1' or
             decoded_instruction_DSP_lat(h)(KSUBV_bit_position)      = '1' or
             decoded_instruction_DSP_lat(h)(KVCP_bit_position)       = '1' then
            dsp_sc_data_write_wire_int(h) <= dsp_out_adder_results(0);
          end if;


          if decoded_instruction_DSP_lat(h)(KRELU_bit_position)  = '1' then
            dsp_sc_data_write_wire_int(h) <= dsp_out_relu_results(0);
          end if;

          if decoded_instruction_DSP_lat(h)(KBCAST_bit_position) = '1' and MVTYPE_DSP(h) = "10" then
            for i in 0 to SIMD-1 loop
              dsp_sc_data_write_wire_int(h)(31+32*(i) downto 32*(i)) <= RS1_Data_IE_lat(h);
            end loop;
          elsif decoded_instruction_DSP_lat(h)(KBCAST_bit_position) = '1' and MVTYPE_DSP(h) = "01" then
            for i in 0 to 2*SIMD-1 loop
              dsp_sc_data_write_wire_int(h)(15+16*(i) downto 16*(i)) <= RS1_Data_IE_lat(h)(15 downto 0);
            end loop;
          elsif decoded_instruction_DSP_lat(h)(KBCAST_bit_position)  = '1' and MVTYPE_DSP(h) = "00" then
            for i in 0 to 4*SIMD-1 loop
              dsp_sc_data_write_wire_int(h)(7+8*(i)   downto 8*(i))  <= RS1_Data_IE_lat(h)(7 downto 0);
            end loop;
          end if;

          if halt_dsp(h) = '0' and halt_dsp_lat(h) = '1' then
            dsp_sc_data_write_wire(h) <= dsp_sc_data_write_int(h);
          end if;
        when others =>
          null;
      end case;
    end if;
  end loop;
 end process;
end generate;

--FU_IN_MAPPER_replicated : for f in accl_range generate
--FU_IN_MAPPER  : if (multithreaded_accl_en = 0 or (multithreaded_accl_en = 1 and f = 0)) generate
FU_replicated : for f in fu_range generate

 DSP_MAPPING_IN_UNIT_comb : process(all)
 variable h : integer;
 begin

   dsp_in_mul_operands(f)        <= (others => (others => '0'));
   dsp_in_adder_operands(f)      <= (others => (others => '0'));
   dsp_in_shift_amount(f)        <= (others => '0');
   dsp_in_shifter_operand(f)     <= (others => '0');
   dsp_in_relu_operands(f)       <= (others => '0');
   dsp_in_accum_operands(f)      <= (others => '0');
```

```
1369        for g in 0 to (ACCL_NUM - FU_NUM) loop
1370
1371          if multithreaded_accl_en = 1 then
1372            h := g;  -- set the spm rd/wr ports equal to the "for-loop"
1373          elsif multithreaded_accl_en = 0 then
1374            h := f;  -- set the spm rd/wr ports equal to the "for-generate"
1375          end if;
1376
1377          if dsp_instr_req(h) = '1' or busy_DSP_internal_lat(h) = '1' then
1378            case state_DSP(h) is
1379
1380              when dsp_exec =>
1381
1382                if (decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1' or
1383                   decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1') and
1384                   MVTYPE_DSP(h) = "00" then
1385                  for i in 0 to 2*SIMD-1 loop
1386                    dsp_in_mul_operands(f)(0)(15+16*(i) downto 16*(i)) <= (x"00" & (dsp_sc_data_read(h)(0)(7+8*(i) downto 8*(i)) and
1387        dsp_sc_data_read_mask(h)(7+8*(i) downto 8*(i))));
1388                    dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= (x"00" & (dsp_sc_data_read(h)(1)(7+8*(i) downto 8*(i)) and
1389        dsp_sc_data_read_mask(h)(7+8*(i) downto 8*(i))));
1390                    if dotp(h) = '1' then
1391                      dsp_in_accum_operands(f) <= dsp_out_mul_results(f);
1392                    elsif dotpps(h) = '1' then
1393                      dsp_in_shift_amount(f)    <= MPSCLFAC_DSP(h);
1394                      dsp_in_shifter_operand(f) <= dsp_out_mul_results(f);
1395                      dsp_in_accum_operands(f)  <= dsp_out_shifter_results(f);
1396                    end if;
1397                  end loop;
1398                end if;
1399
1400                if (decoded_instruction_DSP_lat(h)(KDOTP_bit_position)   = '1'  or
1401                   decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1') and
1402                  (MVTYPE_DSP(h) = "01" or MVTYPE_DSP(h) = "10") then
1403                  dsp_in_mul_operands(f)(0) <= dsp_sc_data_read(h)(0) and dsp_sc_data_read_mask(h);
1404                  dsp_in_mul_operands(f)(1) <= dsp_sc_data_read(h)(1) and dsp_sc_data_read_mask(h);
1405                  if dotp(h) = '1' then
1406                    dsp_in_accum_operands(f)  <= dsp_out_mul_results(f);
1407                  elsif dotpps(h) = '1' then
1408                    dsp_in_shift_amount(f)    <= MPSCLFAC_DSP(h);
1409                    dsp_in_shifter_operand(f) <= dsp_out_mul_results(f);
1410                    dsp_in_accum_operands(f)  <= dsp_out_shifter_results(f);
1411                  end if;
1412                end if;
1413
1414                if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
1415                   decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
1416                   decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
1417                   MVTYPE_DSP(h) = "00" then
1418                  for i in 0 to 2*SIMD-1 loop
1419                    if vec_read_rs2_DSP(h) = '0' then
1420                      if rf_rs2(h) = '1' then
1421                        dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= x"00" & RS2_Data_IE_lat(h)(7 downto 0); -- map the scalar value
1422                      elsif rf_rs2(h) = '0' then
1423                        dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= x"00" & dsp_sc_data_read(h)(1)(7 downto 0); -- map the scalar value
1424                      end if;
1425                    else
1426                      dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= x"00" & dsp_sc_data_read(h)(1)(7+8*(i) downto 8*(i));
1427                    end if;
1428                    dsp_in_mul_operands(f)(0)(15+16*(i) downto 16*(i))  <= x"00" & dsp_sc_data_read(h)(0)(7+8*(i) downto 8*(i));
1429                  end loop;
1430                end if;
1431
1432                if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
1433                   decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
1434                   decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
1435                   MVTYPE_DSP(h) = "01" then
1436                  if vec_read_rs2_DSP(h) = '0' then
1437                    if rf_rs2(h) = '1' then
1438                      for i in 0 to 2*SIMD-1 loop
1439                        dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= RS2_Data_IE_lat(h)(15 downto 0); -- map the scalar value
1440                      end loop;
1441                    elsif rf_rs2(h) = '0' then
1442                      for i in 0 to 2*SIMD-1 loop
1443                        dsp_in_mul_operands(f)(1)(15+16*(i) downto 16*(i)) <= dsp_sc_data_read(h)(1)(15 downto 0); -- map the scalar value
1444                      end loop;
1445                    end if;
1446                  else
```

```
      dsp_in_mul_operands(f)(1) <= dsp_sc_data_read(h)(1);
    end if;
    dsp_in_mul_operands(f)(0)    <= dsp_sc_data_read(h)(0);
  end if;

  if (decoded_instruction_DSP_lat(h)(KVMUL_bit_position)    = '1'  or
      decoded_instruction_DSP_lat(h)(KSVMULRF_bit_position) = '1'  or
      decoded_instruction_DSP_lat(h)(KSVMULSC_bit_position) = '1') and
     MVTYPE_DSP(h) = "10" then
    if vec_read_rs2_DSP(h) = '0' then
      if rf_rs2(h) = '1' then
        for i in 0 to SIMD-1 loop
          dsp_in_mul_operands(f)(1)(31+32*(i) downto 32*(i)) <= RS2_Data_IE_lat(h)(31 downto 0); -- map the scalar value
        end loop;
      elsif rf_rs2(h) = '0' then
        for i in 0 to SIMD-1 loop
          dsp_in_mul_operands(f)(1)(31+32*(i) downto 32*(i)) <= dsp_sc_data_read(h)(1)(31 downto 0); -- map the scalar value
        end loop;
      end if;
    else
      dsp_in_mul_operands(f)(1) <= dsp_sc_data_read(h)(1);
    end if;
    dsp_in_mul_operands(f)(0) <= dsp_sc_data_read(h)(0);
  end if;

  if decoded_instruction_DSP_lat(h)(KADDV_bit_position) = '1' then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    dsp_in_adder_operands(f)(1)   <= dsp_sc_data_read(h)(1);
  end if;

  if decoded_instruction_DSP_lat(h)(KSRAV_bit_position) = '1' or
     decoded_instruction_DSP_lat(h)(KSRLV_bit_position) = '1' then
    dsp_in_shifter_operand(f)      <= dsp_sc_data_read(h)(0);
    dsp_in_shift_amount(f)         <= RS2_Data_IE_lat(h)(4 downto 0); -- map the scalar value (shift amount)
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position)  = '1' and MVTYPE_DSP(h) = "10" then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    for i in 0 to SIMD-1 loop
      dsp_in_adder_operands(f)(1)(31+32*(i) downto 32*(i))   <= dsp_sc_data_read(h)(1)(31 downto 0);
    end loop;
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position)  = '1' and MVTYPE_DSP(h) = "01" then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    for i in 0 to 2*SIMD-1 loop
      dsp_in_adder_operands(f)(1)(15+16*(i) downto 16*(i))   <= dsp_sc_data_read(h)(1)(15 downto 0);
    end loop;
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDSC_bit_position)  = '1' and MVTYPE_DSP(h) = "00" then
    dsp_in_adder_operands(f)(0) <= dsp_sc_data_read(h)(0);
    for i in 0 to 4*SIMD-1 loop
      dsp_in_adder_operands(f)(1)(7+8*(i) downto 8*(i)) <= dsp_sc_data_read(h)(1)(7 downto 0);
    end loop;
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position) = '1' and MVTYPE_DSP(h) = "10" then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    for i in 0 to SIMD-1 loop
      dsp_in_adder_operands(f)(1)(31+32*(i) downto 32*(i))   <= RS2_Data_IE_lat(h)(31 downto 0);
    end loop;
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position) = '1' and MVTYPE_DSP(h) = "01" then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    for i in 0 to 2*SIMD-1 loop
      dsp_in_adder_operands(f)(1)(15+16*(i) downto 16*(i))   <= RS2_Data_IE_lat(h)(15 downto 0);
    end loop;
  end if;

  if decoded_instruction_DSP_lat(h)(KSVADDRF_bit_position) = '1' and MVTYPE_DSP(h) = "00" then
    dsp_in_adder_operands(f)(0)   <= dsp_sc_data_read(h)(0);
    for i in 0 to 4*SIMD-1 loop
      dsp_in_adder_operands(f)(1)(7+8*(i) downto 8*(i))   <= RS2_Data_IE_lat(h)(7 downto 0);
    end loop;
  end if;

```

```
1525            if decoded_instruction_DSP_lat(h)(KSUBV_bit_position)  = '1' then
1526              dsp_in_adder_operands(f)(0)     <= dsp_sc_data_read(h)(0);
1527              dsp_in_adder_operands(f)(1)     <= (not dsp_sc_data_read(h)(1));
1528            end if;
1529
1530            if decoded_instruction_DSP_lat(h)(KVRED_bit_position)  = '1' and MVTYPE_DSP(h) = "00" then
1531              for i in 0 to 2*SIMD-1 loop
1532                dsp_in_accum_operands(f)(15+16*(i) downto 16*(i)) <= x"00" & (dsp_sc_data_read(h)(0)(7+8*(i) downto 8*(i)) and
1533    dsp_sc_data_read_mask(h)(7+8*(i) downto 8*(i)));
1534              end loop;
1535            end if;
1536            if decoded_instruction_DSP_lat(h)(KVRED_bit_position) = '1' and (MVTYPE_DSP(h) = "01" or MVTYPE_DSP(h) = "10") then
1537              dsp_in_accum_operands(f) <= dsp_sc_data_read(h)(0) and dsp_sc_data_read_mask(h);
1538            end if;
1539
1540            if decoded_instruction_DSP_lat(h)(KRELU_bit_position)  = '1' then
1541              dsp_in_relu_operands(f) <= dsp_sc_data_read(h)(0);
1542            end if;
1543
1544            if decoded_instruction_DSP_lat(h)(KVCP_bit_position) = '1' then
1545              dsp_in_adder_operands(f)(0) <= dsp_sc_data_read(h)(0);
1546            end if;
1547
1548
1549          when others =>
1550            null;
1551        end case;
1552      end if;
1553    end loop;
1554  end process;
1555
1556  --end generate;
1557  --end generate;
1558
1559  --FU_IN_MAPPER  : if (multithreaded_accl_en = 0 or (multithreaded_accl_en = 1 and f = 0) generate
1560
1561    fsm_DSP_adder_stage_1 : process(all)
1562    variable h : integer;
1563    begin
1564      dsp_add_8_0_wire(f)  <= dsp_add_8_0(f);
1565      dsp_add_16_8_wire(f) <= dsp_add_16_8(f);
1566      for g in 0 to (ACCL_NUM - FU_NUM) loop
1567        if multithreaded_accl_en = 1 then
1568          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
1569        elsif multithreaded_accl_en = 0 then
1570          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
1571        end if;
1572        --  Addition in SIMD Virtual Parallelism is executed here, if the carries are blocked, we will have a chain of 8-bit or 16-bit adders, else we have
1573    32-bit adders
1574        for i in 0 to SIMD-1 loop
1575          if (adder_stage_1_en(h) = '1' or recover_state_wires(h) = '1') then
1576            -- Unwinding the loop:
1577            -- (1) the term "8*(4*i)" is used to jump between the 32-bit words, inside the 128-bit values read by the DSP
1578            -- (2) Each addition results in an 8-bit value, and the 9th bit being the carry, depending on the instruction (KADDV32, KADDV16, KADDV8)
1579    we either pass the or block the carries.
1580           -- (3) CARRIES:
1581            -- (a) If we pass all the carries in the 32-bit word, we will have executed KADDV32 (4*32-bit parallel additions)
1582            -- (b) If we pass the 9th and 25th carries we would have executed KADDV16 (8*16-bit parallel additions)
1583            -- (c) If we pass none of the carries then we would have executed KADDV8 (16*8-bit parallel additions)
1584           dsp_add_8_0_wire(f)(i)   <= std_logic_vector('0' & unsigned(dsp_in_adder_operands(f)(0)(7+8*(4*i)  downto 8*(4*i)))    +
1585    unsigned(dsp_in_adder_operands(f)(1)(7+8*(4*i)  downto 8*(4*i))) + twos_complement(h)(0+(4*i)));
1586           dsp_add_16_8_wire(f)(i)  <= std_logic_vector('0' & unsigned(dsp_in_adder_operands(f)(0)(15+8*(4*i) downto 8+8*(4*i)))  +
1587    unsigned(dsp_in_adder_operands(f)(1)(15+8*(4*i) downto 8+8*(4*i))) + carry_8_wire(f)(i) + twos_complement(h)(1+(4*i)));
1588           -- All the 8-bit adders are lumped into one output write signal that will write to the scratchpads
1589           -- Carries are either passed or blocked for the 9-th, 17-th, and 25-th bits
1590           carry_8_wire(f)(i)  <= dsp_add_8_0_wire(f)(i)(8)   and carry_pass(h)(0);
1591           carry_16_wire(f)(i) <= dsp_add_16_8_wire(f)(i)(8)  and carry_pass(h)(1);
1592          end if;
1593        end loop;
1594      end loop;
1595    end process;
1596
1597    fsm_DSP_adder_stage_2 : process(all)
1598    variable h : integer;
1599    begin
1600      carry_24_wire(f)             <= (others => '0');
1601      dsp_add_24_16_wire(f)        <= (others => (others => '0'));
1602      dsp_add_32_24_wire(f)        <= (others => (others => '0'));
```

```
    for g in 0 to (ACCL_NUM - FU_NUM) loop
      if multithreaded_accl_en = 1 then
        h := g;  -- set the spm rd/wr ports equal to the "for-loop"
      elsif multithreaded_accl_en = 0 then
        h := f;  -- set the spm rd/wr ports equal to the "for-generate"
      end if;
      -- Addition is here
      if halt_dsp_lat(h) = '0' then
      --  Addition in SIMD Virtual Parallelism is executed here, if the carries are blocked, we will have a chain of 8-bit or 16-bit adders, else we have
32-bit adders
        for i in 0 to SIMD-1 loop
          if (adder_stage_2_en(h) = '1' or recover_state_wires(h) = '1') then
            dsp_add_24_16_wire(f)(i) <= std_logic_vector('0' & unsigned(dsp_in_adder_operands_lat(f)(0)(7+8*(2*i) downto 8*(2*i))) +
                                                unsigned(dsp_in_adder_operands_lat(f)(1)(7+8*(2*i) downto 8*(2*i))) +
                                                    carry_16(f)(i) + twos_complement(h)(2+(4*i)));
            dsp_add_32_24_wire(f)(i) <= std_logic_vector('0' & unsigned(dsp_in_adder_operands_lat(f)(0)(15+8*(2*i) downto 8+8*(2*i))) +
                                                unsigned(dsp_in_adder_operands_lat(f)(1)(15+8*(2*i) downto 8+8*(2*i))) +
                                                    carry_24_wire(f)(i) + twos_complement(h)(3+(4*i)));
            -- All the 8-bit adders are lumped into one output write signal that will write to the scratchpads
            -- Carries are either passed or blocked for the 9-th, 17-th, and 25-th bits
            carry_24_wire(f)(i) <= dsp_add_24_16_wire(f)(i)(8) and carry_pass(h)(2);
          end if;
        end loop;
      end if;
    end loop;
  end process;

  fsm_DSP_adder : process(clk_i, rst_ni)
  variable h : integer;
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        -- Addition is here
        if add_en(h) = '1' and halt_dsp_lat(h) = '0' then
          carry_16(f) <= carry_16_wire(f);
          dsp_add_8_0(f)  <= dsp_add_8_0_wire(f);
          dsp_add_16_8(f) <= dsp_add_16_8_wire(f);
        --  Addition in SIMD Virtual Parallelism is executed here, if the carries are blocked, we will have a chain of 8-bit or 16-bit adders, else we have
normal 32-bit adders
          for i in 0 to SIMD-1 loop
            if (adder_stage_2_en(h) = '1' or recover_state_wires(h) = '1') then
              -- All the 8-bit adders are lumped into one output signal that will write to the scratchpads
              dsp_out_adder_results(f)(31+32*(i) downto 32*(i)) <= dsp_add_32_24_wire(f)(i)(7 downto 0) & dsp_add_24_16_wire(f)(i)(7 downto 0) &
dsp_add_16_8(f)(i)(7 downto 0) & dsp_add_8_0(f)(i)(7 downto 0);
            end if;
          end loop;
        end if;
        for i in 0 to SIMD-1 loop
          for j in 0 to 1 loop
            dsp_in_adder_operands_lat(f)(j)(15 +16*(i) downto 16*(i)) <= dsp_in_adder_operands(f)(j)(31+32*(i) downto 16+32*(i));
          end loop;
        end loop;
      end loop;
    end if;
  end process;

  fsm_DSP_shifter_stg_1 : process(clk_i, rst_ni)
  variable h : integer;
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        if shift_en(h) = '1' and (shifter_stage_1_en(h) = '1' or recover_state_wires(h) = '1') and halt_dsp_lat(h) = '0' then
          for i in 0 to SIMD-1 loop
            dsp_int_shifter_operand(f)(31+32*(i) downto 32*(i)) <= to_stdlogicvector(to_bitvector(dsp_in_shifter_operand(f)(31+32*(i) downto 32*(i)))
srl to_integer(unsigned(dsp_in_shift_amount(f))));
          end loop;
```

```
          --for i in 0 to 4*SIMD-1 loop -- latch the sign bits
            --dsp_in_sign_bits(f)(i) <= dsp_in_shifter_operand(f)(7+8*(i));
          --end loop;
          if MVTYPE_DSP(h) = "00" then
            for i in 0 to 4*SIMD-1 loop -- latch the sign bits
              dsp_in_shifter_operand_lat(f)(7+8*i downto 8*i) <= (others => dsp_in_shifter_operand(f)(7+8*i));
            end loop;
          elsif MVTYPE_DSP(h) = "01" then
            for i in 0 to 2*SIMD-1 loop -- latch the sign bits
              dsp_in_shifter_operand_lat(f)(15+16*i downto 16*i) <= (others => dsp_in_shifter_operand(f)(15+16*i));
            end loop;
          elsif MVTYPE_DSP(h) = "10" then
            for i in 0 to SIMD-1 loop -- latch the sign bits
              dsp_in_shifter_operand_lat(f)(31+32*i downto 32*i) <= (others => dsp_in_shifter_operand(f)(31+32*i));
            end loop;
          end if;
        end if;
      end loop;
    end if;
  end process;

  fsm_DSP_shifter_stg_2 : process(clk_i, rst_ni)
  variable h : integer;
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        if shift_en(h) = '1' and (shifter_stage_2_en(h) = '1' or recover_state_wires(h) = '1') and halt_dsp_lat(h) = '0' then
          if    MVTYPE_DSP(h) = "10" then
            for i in 0 to SIMD-1 loop
              dsp_out_shifter_results(f)(31+32*(i) downto 32*(i)) <= dsp_in_shifter_operand_lat_wire(f)(31 +32*(i) downto 32*(i)) or
dsp_int_shifter_operand(f)(31+32*(i) downto 32*(i));
            end loop;
          elsif MVTYPE_DSP(h) = "01" or (decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1' and MVTYPE_DSP(h) = "00") then --
KDOTPPS8 has been added here because the number of elements loaded for mul operations is equal for 8-bit and 16-bits instr
            for i in 0 to 2*SIMD-1 loop
              dsp_out_shifter_results(f)(15+16*(i) downto 16*(i)) <=  dsp_in_shifter_operand_lat_wire(f)(15 +16*(i) downto 16*(i)) or
(dsp_int_shifter_operand(f)(15+16*(i) downto 16*(i)) and dsp_shift_enabler(h)(15 downto 0));
            end loop;
          elsif MVTYPE_DSP(h) = "00" then
            for i in 0 to 4*SIMD-1 loop
              dsp_out_shifter_results(f)(7+8*(i) downto 8*(i)) <=  dsp_in_shifter_operand_lat_wire(f)(7 +8*(i) downto 8*(i)) or
(dsp_int_shifter_operand(f)(7+8*(i) downto 8*(i)) and dsp_shift_enabler(h)(7 downto 0));
            end loop;
          end if;
        end if;
      end loop;
    end if;
  end process;

  fsm_DSP_shifter_comb : process(all)
  variable h : integer;
  begin
    dsp_in_shifter_operand_lat_wire(f) <= (others => '0');
    for g in 0 to (ACCL_NUM - FU_NUM) loop
      if multithreaded_accl_en = 1 then
        h := g;  -- set the spm rd/wr ports equal to the "for-loop"
      elsif multithreaded_accl_en = 0 then
        h := f;  -- set the spm rd/wr ports equal to the "for-generate"
      end if;
      dsp_shift_enabler(h) <= (others => '0');
      if shift_en(h) = '1' and halt_dsp_lat(h) = '0' then
        if MVTYPE_DSP(h) = "01" then
          dsp_shift_enabler(h)(15 - to_integer(unsigned(dsp_in_shift_amount(h)(3 downto 0))) downto 0) <= (others => '1');
        elsif MVTYPE_DSP(h) = "00" then
          dsp_shift_enabler(h)(7 -  to_integer(unsigned(dsp_in_shift_amount(h)(2 downto 0))) downto 0) <= (others => '1');
        end if;
        if (decoded_instruction_DSP_lat(h)(KSRAV_bit_position) = '1' or decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1') and
           MVTYPE_DSP(h) = "10" then    -- 32-bit sign extension for for srl in stage 1
          for i in 0 to SIMD-1 loop
            --dsp_in_shifter_operand_lat(f)(31+32*(i) downto 31 - to_integer(unsigned(dsp_in_shift_amount(h)(4 downto 0)))+32*(i))   <= (others =>
dsp_in_sign_bits(h)(3+4*(i)));
            dsp_in_shifter_operand_lat_wire(f)(31+32*(i) downto 31 - to_integer(unsigned(dsp_in_shift_amount(f)(4 downto 0)))+32*(i)) <=
```

```
            dsp_in_shifter_operand_lat(f)(     31+32*(i) downto 31 - to_integer(unsigned(dsp_in_shift_amount(f)(4 downto 0)))+32*(i));
          end loop;
        elsif (decoded_instruction_DSP_lat(h)(KSRAV_bit_position) = '1' or decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1') and
              MVTYPE_DSP(h) = "01" then -- 16-bit sign extension for for srl in stage 1
          for i in 0 to 2*SIMD-1 loop
            --dsp_in_shifter_operand_lat(f)(15+16*(i) downto 15 - to_integer(unsigned(dsp_in_shift_amount(h)(3 downto 0)))+16*(i))   <= (others =>
dsp_in_sign_bits(h)(1+2*(i)));
            dsp_in_shifter_operand_lat_wire(f)(15+16*(i) downto 15 - to_integer(unsigned(dsp_in_shift_amount(f)(3 downto 0)))+16*(i)) <=
            dsp_in_shifter_operand_lat(f)(     15+16*(i) downto 15 - to_integer(unsigned(dsp_in_shift_amount(f)(3 downto 0)))+16*(i));
          end loop;
        elsif (decoded_instruction_DSP_lat(h)(KSRAV_bit_position) = '1'  or decoded_instruction_DSP_lat(h)(KDOTPPS_bit_position) = '1') and
              MVTYPE_DSP(h) = "00" then  -- 8-bit  sign extension for for srl in stage 1
          for i in 0 to 4*SIMD-1 loop
            --dsp_in_shifter_operand_lat(f)(7+8*(i) downto 7 - to_integer(unsigned(dsp_in_shift_amount(h)(2 downto 0)))+8*(i))    <= (others =>
dsp_in_sign_bits(h)(i));
            dsp_in_shifter_operand_lat_wire(f)(7+8*(i) downto 7 - to_integer(unsigned(dsp_in_shift_amount(f)(2 downto 0)))+8*(i)) <=
            dsp_in_shifter_operand_lat(f)(     7+8*(i) downto 7 - to_integer(unsigned(dsp_in_shift_amount(f)(2 downto 0)))+8*(i));
          end loop;
        end if;
      end if;
    end loop;
  end process;
  -- STAGE 1 --
  fsm_MUL_STAGE_1 : process(clk_i,rst_ni)
  variable h : integer;
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        if halt_dsp_lat(h) = '0' then
          if mul_en(h) = '1' and (mul_stage_1_en(h) = '1' or recover_state_wires(h) = '1') then
                    for i in 0 to SIMD-1 loop
            -- Unwinding the loop:
            -- (1) The impelemtation in the loop does multiplication for KDOTP32, and KDOTP16 using only 16-bit multipliers. "A*B" =
[Ahigh*(2^16) + Alow]*[Bhigh*(2^16) + Blow]
            -- (2) Expanding this equation "[Ahigh*(2^16) + Alow]*[Bhigh*(2^16) + Blow]"  gives: "Ahigh*Bhigh*(2^32) + Ahigh*Blow*(2^16) +
Alow*Bhigh*(2^16) + Alow*Blow" which are the terms being stored in dsp_out_mul_results
            -- (3) Partial Multiplication
              -- (a) "dsp_mul_a" <= Ahigh*Bhigh
              -- (b) "dsp_mul_b" <= Ahigh*Blow
              -- (c) "dsp_mul_c" <= Alow*Bhigh
              -- (d) "dsp_mul_d" <= Alow*Blow
            -- (4) "dsp_mul_a" is shifted by 32 bits to the left, "dsp_mul_b" and "dsp_mul_c" are shifted by 16-bits to the left, "dsp_mul_d" is not shifted
            -- (5) For 16-bit and 8-bit muls, the FUNCT_SELECT_MASK is set to x"00000000" blocking the terms in "dsp_mul_b" and "dsp_mul_c".
For executing 32-bit muls , we set the mask to x"FFFFFFFF"
            dsp_mul_a(f)(31+32*(i)  downto 32*(i)) <= std_logic_vector(unsigned(dsp_in_mul_operands(f)(0)(15+16*(2*i+1)    downto 16*(2*i+1))) *
unsigned(dsp_in_mul_operands(f)(1)(15+16*(2*i+1)  downto 16*(2*i+1))));
            dsp_mul_b(f)(31+32*(i)  downto 32*(i)) <= std_logic_vector((unsigned(dsp_in_mul_operands(f)(0)(16*(2*i+1) - 1  downto 16*(2*i)))   *
unsigned(dsp_in_mul_operands(f)(1)(15+16*(2*i+1)  downto 16*(2*i+1)))) and unsigned(FUNCT_SELECT_MASK(h)));
            dsp_mul_c(f)(31+32*(i)  downto 32*(i)) <= std_logic_vector((unsigned(dsp_in_mul_operands(f)(0)(15+16*(2*i+1)   downto 16*(2*i+1))) *
unsigned(dsp_in_mul_operands(f)(1)(16*(2*i+1) - 1 downto 16*(2*i))))   and unsigned(FUNCT_SELECT_MASK(h)));
            dsp_mul_d(f)(31+32*(i)  downto 32*(i)) <= std_logic_vector(unsigned(dsp_in_mul_operands(f)(0)(16*(2*i+1)  - 1  downto 16*(2*i)))   *
unsigned(dsp_in_mul_operands(f)(1)(16*(2*i+1) - 1 downto 16*(2*i))));
          end loop;
        end if;
      end if;
    end loop;
  end if;
  end process;

  fsm_MUL_STAGE_1_COMB : process(all)
  variable h : integer;
  begin
    mul_tmp_a(f) <= (others => (others => '0'));
    mul_tmp_b(f) <= (others => (others => '0'));
    mul_tmp_c(f) <= (others => (others => '0'));
    mul_tmp_d(f) <= (others => (others => '0'));
    for g in 0 to (ACCL_NUM - FU_NUM) loop
      if multithreaded_accl_en = 1 then
        h := g;  -- set the spm rd/wr ports equal to the "for-loop"
      elsif multithreaded_accl_en = 0 then
        h := f;  -- set the spm rd/wr ports equal to the "for-generate"
      end if;
```

```
      -- KDOTP and KSVMUL instructions are handeled here
      -- this part right here shifts the intermidiate resutls appropriately, and then accumulates them in order to get the final mul result
      if mul_en(h) = '1' and (mul_stage_2_en(h) = '1' or recover_state_wires(h) = '1') then
        for i in 0 to SIMD-1 loop
          if MVTYPE_DSP(h) /= "10" then
            ---------------------------------------------------------------------------------
            mul_tmp_a(f)(i) <= (dsp_mul_a(f)(15+16*(2*i)  downto 16*(2*i)) & x"0000");
            mul_tmp_d(f)(i) <= (x"0000" & dsp_mul_d(f)(15+16*(2*i)  downto 16*(2*i)));
            ---------------------------------------------------------------------------------
          elsif MVTYPE_DSP(h) = "10" then
            -- mul_tmp_a(f)(i) <= (dsp_mul_a(f)(31+32*(2*i)  downto 31*(2*i)) & x"0000");     -- The upper 32-bit results of the multiplication are
discarded   (Ah*Bh)
            mul_tmp_b(f)(i) <= (dsp_mul_b(f)(15+16*(2*i) downto 16*(2*i)) & x"0000");         -- Modified to only add the partail result to the lower 32-
bits   (Ah*Bl)
            mul_tmp_c(f)(i) <= (dsp_mul_c(f)(15+16*(2*i) downto 16*(2*i)) & x"0000");         -- Modified to only add the partail result to the lower 32-
bits   (Al*Bh)
            mul_tmp_d(f)(i) <= (dsp_mul_d(f)(31+32*(i)   downto 32*(i)));                    -- This is the lower 32-bit result of the partial mmultiplication
(Al*Bl)
          end if;
        end loop;
      end if;
    end loop;
  end process;

  -- STAGE 2 --
  fsm_MUL_STAGE_2 : process(clk_i, rst_ni)
  variable h : integer;
  begin
              if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        -- Accumulate the partial multiplications to make up bigger multiplications
        if mul_en(h) = '1' and (mul_stage_2_en(h) = '1' or recover_state_wires(h) = '1') and halt_dsp_lat(h) = '0' then
          for i in 0 to SIMD-1 loop
            dsp_out_mul_results(f)((Data_Width-1)+Data_Width*(i) downto Data_Width*(i))  <= (std_logic_vector(unsigned(mul_tmp_a(f)(i)) +
unsigned(mul_tmp_b(f)(i)) + unsigned(mul_tmp_c(f)(i)) + unsigned(mul_tmp_d(f)(i))));
          end loop;
        end if;
      end loop;
    end if;
  end process;

  fsm_RELU : process(clk_i, rst_ni)
  variable h : integer;
  begin
    if rst_ni = '0' then
    elsif rising_edge(clk_i) then
      for g in 0 to (ACCL_NUM - FU_NUM) loop
        if multithreaded_accl_en = 1 then
          h := g;  -- set the spm rd/wr ports equal to the "for-loop"
        elsif multithreaded_accl_en = 0 then
          h := f;  -- set the spm rd/wr ports equal to the "for-generate"
        end if;
        if relu_en(h) = '1' then
          if (relu_stage_1_en(h) = '1' or recover_state_wires(h) = '1') and halt_dsp_lat(h) = '0' then
            if    MVTYPE_DSP(h) = "10" then
              for i in 0 to SIMD-1 loop
                if dsp_in_relu_operands(f)(31+32*(i)) = '1' then
                  dsp_out_relu_results(f)(31+32*(i) downto 32*(i)) <= (others => '0');
                else
                  dsp_out_relu_results(f)(31+32*(i) downto 32*(i)) <= dsp_in_relu_operands(f)(31+32*(i) downto 32*(i));
                end if;
              end loop;
            elsif MVTYPE_DSP(h) = "01" then
              for i in 0 to 2*SIMD-1 loop
                if dsp_in_relu_operands(f)(15+16*(i)) = '1' then
                  dsp_out_relu_results(f)(15+16*(i) downto 16*(i)) <= (others => '0');
                else
                  dsp_out_relu_results(f)(15+16*(i) downto 16*(i)) <= dsp_in_relu_operands(f)(15+16*(i) downto 16*(i));
                end if;
              end loop;
            elsif MVTYPE_DSP(h) = "00" then
              for i in 0 to 4*SIMD-1 loop
```

```
            if dsp_in_relu_operands(f)(7+8*(i)) = '1' then
              dsp_out_relu_results(f)(7+8*(i) downto 8*(i)) <= (others => '0');
            else
              dsp_out_relu_results(f)(7+8*(i) downto 8*(i)) <= dsp_in_relu_operands(f)(7+8*(i) downto 8*(i));
            end if;
          end loop;
        end if;
      end if;
    end if;
  end loop;
 end if;
end process;

end generate FU_replicated;

 ACCUM_STG : ACCUMULATOR
            port map(
    clk_i                       => clk_i,
    rst_ni                      => rst_ni,
    MVTYPE_DSP                    => MVTYPE_DSP,
    accum_stage_1_en              => accum_stage_1_en,
    accum_stage_2_en              => accum_stage_2_en,
    recover_state_wires           => recover_state_wires,
    halt_dsp_lat                  => halt_dsp_lat,
    state_DSP                     => state_DSP,
    decoded_instruction_DSP_lat   => decoded_instruction_DSP_lat,
    dsp_in_accum_operands         => dsp_in_accum_operands,
    dsp_out_accum_results         => dsp_out_accum_results
            );

end DSP;
-------------------------------------------------------------------------------------------
-- END of DSP architecture ------------------------------------------------------------
-------------------------------------------------------------------------------------------
```

3. Scratchpad Memory Interface (SPI)

```
-- SCI  pinout --------------------
entity Scratchpad_memory_interface is
  port (
    clk_i, rst_ni           : in  std_logic;
    data_rvalid_i           : in  std_logic;
    state_LS                : in  fsm_LS_states;
    sc_word_count_wire      : in  integer;
    spm_bcast               : in  std_logic;
    harc_LS_wire            : in  accl_range;
    dsp_we_word             : in  array_2d(accl_range)(SIMD-1 downto 0);
    ls_sc_data_write_wire   : in  std_logic_vector(Data_Width-1 downto 0);
    dsp_sc_data_write_wire  : in  array_2d(accl_range)(SIMD_Width-1 downto 0);
    ls_sc_read_addr         : in  std_logic_vector(Addr_Width-(SIMD_BITS+3) downto 0);
    ls_sc_write_addr        : in  std_logic_vector(Addr_Width-(SIMD_BITS+3) downto 0);
    dsp_sc_write_addr       : in  array_2d(accl_range)(Addr_Width-1 downto 0);
    ls_sci_req              : in  std_logic_vector(SPM_NUM-1 downto 0);
    ls_sci_we               : in  std_logic_vector(SPM_NUM-1 downto 0);
    dsp_sci_req             : in  array_2d(accl_range)(SPM_NUM-1 downto 0);
    dsp_sci_we              : in  array_2d(accl_range)(SPM_NUM-1 downto 0);
    kmemld_inflight         : in  std_logic_vector(SPM_NUM-1 downto 0);
    kmemstr_inflight        : in std_logic_vector(SPM_NUM-1 downto 0);
    dsp_to_sc               : in  array_3d(accl_range)(SPM_NUM-1 downto 0)(1 downto 0);
    dsp_sc_read_addr        : in  array_3d(accl_range)(1 downto 0)(Addr_Width-1 downto 0);
    dsp_sc_data_read        : out array_3d(accl_range)(1 downto 0)(SIMD_Width-1 downto 0);
    ls_sc_data_read_wire    : out std_logic_vector(Data_Width-1 downto 0);
    ls_sci_wr_gnt           : out std_logic;
    dsp_sci_wr_gnt          : out std_logic_vector(accl_range);
    ls_data_gnt_i           : out std_logic_vector(SPM_NUM-1 downto 0);
    dsp_data_gnt_i          : out std_logic_vector(accl_range)
            );
end entity;  -----------------------------------------


architecture SCI of Scratchpad_memory_interface is

signal dsp_sc_data_write_int_wire   : array_2d(accl_range)(SIMD_Width-1 downto 0);
signal ls_sc_data_read_int_wire     : array_2d(accl_range)(Data_Width-1 downto 0);
signal rd_offset                    : array_3d(accl_range)(1 downto 0)(SIMD-1 downto 0);
```

```
39  signal wr_offset                    : array_2d(accl_range)(SIMD-1 downto 0);
40  signal dsp_sc_data_read_int_wire    : array_3d(accl_range)(1 downto 0)(SIMD_Width-1 downto 0);
41  signal dsp_sc_read_addr_lat         : array_3d(accl_range)(1 downto 0)(SIMD_BITS+1 downto 0);  --  Only need the lower part to check for the word
42  access
43  signal dsp_sci_req_lat              : array_2d(accl_range)(SPM_NUM-1 downto 0);
44  signal dsp_to_sc_lat                : array_3d(accl_range)(SPM_NUM-1 downto 0)(1 downto 0);
45  signal dsp_sc_data_read_wire        : array_3d(accl_range)(1 downto 0)(SIMD_Width-1 downto 0);
46  signal ls_sc_data_read_replicated   : array_2d(accl_range)(Data_Width-1 downto 0);
47  signal ls_sc_data_read_wire_replicated : array_2d(accl_range)(Data_Width-1 downto 0);
48  signal dsp_sci_wr_gnt_lat           : std_logic_vector(accl_range);
49  signal ls_sci_wr_gnt_replicated     : std_logic_vector(accl_range);
50  signal ls_sci_wr_gnt_lat_replicated : std_logic_vector(accl_range);
51  signal halt_dsp                     : std_logic_vector(accl_range);
52  signal sc_word_count                : array_2d_int(accl_range);
53  signal sc_we                        : array_2d(accl_range)(SIMD*SPM_NUM-1  downto 0);
54  signal sc_addr_wr                   : array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
55  signal sc_addr_rd                   : array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
56  signal sc_data_wr                   : array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0);
57  signal sc_data_rd                   : array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0);
58
59  component Scratchpad_memory
60   port(
61       clk_i                  : in  std_logic;
62       sc_we                  : in  array_2d(accl_range)(SIMD*SPM_NUM-1 downto 0);
63       sc_addr_wr             : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
64       sc_addr_rd             : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
65       sc_data_wr             : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0);
66       sc_data_rd             : out array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0)
67     );
68  end component;
69  ----------------------------------------------------------------------------------------------
70  ---------------------------------- SCI BEGIN -------------------------------------------------
71  begin
72
73
74   SC : Scratchpad_memory
75    port map(
76       sc_we          => sc_we,
77       clk_i          => clk_i,
78       sc_addr_rd     => sc_addr_rd,
79       sc_addr_wr     => sc_addr_wr,
80       sc_data_wr     => sc_data_wr,
81       sc_data_rd     => sc_data_rd
82      );
83
84   SPM_replicated : for h in accl_range generate
85
86   SCI_Exec_Unit : process(clk_i, rst_ni)  -- single cycle unit, fully synchronous
87   begin
88    if rst_ni = '0' then
89      dsp_sc_read_addr_lat(h) <= (others => (others => '0'));
90      dsp_to_sc_lat(h)       <= (others => (others => '0'));
91      ls_data_gnt_i          <= (others => '0');
92      dsp_sci_req_lat(h)     <= (others => '0');
93      sc_word_count(h)       <= 0;
94      elsif rising_edge(clk_i) then
95      halt_dsp(h)                 <= '0';
96      dsp_sci_wr_gnt_lat(h)       <= dsp_sci_wr_gnt(h);
97      ls_sci_wr_gnt_lat_replicated(h) <= ls_sci_wr_gnt_replicated(h);
98      dsp_sci_req_lat(h)          <= dsp_sci_req(h);
99      dsp_to_sc_lat(h)            <= dsp_to_sc(h);
100     if harc_LS_wire = h or spm_bcast = '1' then
101       sc_word_count(h)          <= sc_word_count_wire;
102     end if;
103     if unsigned(ls_data_gnt_i) /= 0 then
104       ls_sc_data_read_replicated(h)  <= ls_sc_data_read_wire_replicated(h);
105     end if;
106     if (dsp_sci_wr_gnt(h) = '0' and dsp_sci_we(h) /= (0 to SPM_NUM-1 => '0')) then
107       halt_dsp(h) <= '1';
108     end if;
109     if halt_dsp(h) = '0' then
110       dsp_sc_data_read(h) <= dsp_sc_data_read_wire(h);
111     end if;
112
113     for i in 0 to SPM_NUM-1 loop
114       if ls_sci_req(i) = '1' then  -- AAA most probably useless
115         ls_data_gnt_i(i) <= '1';
116       elsif ls_sci_req(i) = '0' then
```

```
117             ls_data_gnt_i(i) <= '0';
118           end if;
119                   if dsp_sci_req(h)(i) = '1' then
120            for k in 0 to 1 loop
121             dsp_sc_read_addr_lat(h)(k) <= dsp_sc_read_addr(h)(k)(SIMD_BITS+1 downto 0);
122            end loop;
123           end if;
124          end loop;
125        end if;
126      end process;
127
128      ls_sc_data_read_wire <= ls_sc_data_read_wire_replicated(harc_LS_wire);
129      ls_sci_wr_gnt        <= ls_sci_wr_gnt_replicated(harc_LS_wire);
130
131      SCI_Exec_Unit_comb : process(all)
132
133      begin
134        dsp_data_gnt_i(h)            <= '0';
135        for l in 0 to (SIMD*SPM_NUM)-1 loop
136          sc_we(h)(l)      <= '0';
137          sc_addr_rd(h)(l) <= (others => '0');
138          sc_addr_wr(h)(l) <= (others => '0');
139          sc_data_wr(h)(l) <= (others => '0');
140        end loop;
141        rd_offset(h)                    <= (others => (others => '0'));
142        dsp_sc_data_read_int_wire(h)       <= (others => (others => '0'));
143        wr_offset(h)                    <= (others => '0');
144                  ls_sci_wr_gnt_replicated(h)        <= ls_sci_wr_gnt_lat_replicated(h);
145                  dsp_sci_wr_gnt(h)                  <= dsp_sci_wr_gnt_lat(h);
146        ls_sc_data_read_wire_replicated(h) <= ls_sc_data_read_replicated(h);
147                  dsp_sc_data_write_int_wire(h)      <= (others => '0');
148        dsp_sc_data_read_wire(h)          <= dsp_sc_data_read(h);
149        for i in 0 to SPM_NUM-1 loop                -- Loop through scratchpads A,B,C,D
150
151          if data_rvalid_i = '1' then        -- LS write port
152            if ls_sci_req(i) = '1' and ls_sci_we(i) = '1' and ls_sci_wr_gnt = '1' then
153                                  if harc_LS_wire = h or spm_bcast = '1' then
154              sc_we(h)((SIMD)*i + sc_word_count(h)) <= '1';
155              sc_data_wr(h)(sc_word_count(h) + (SIMD)*i) <= ls_sc_data_write_wire(31 downto 0);
156              sc_addr_wr(h)(sc_word_count(h) + (SIMD)*i) <= ls_sc_write_addr;
157            end if;
158          end if;
159        end if;
160
161        if ls_data_gnt_i(i) = '1' then
162                                  if harc_LS_wire = h then
163          ls_sc_data_read_wire_replicated(h) <= sc_data_rd(h)((SIMD)*i + sc_word_count(h));  -- sc_word_count because data being read is delayed
164    one cycle after the request
165          end if;
166        end if;
167
168        if ls_sci_req(i) = '1' then        -- LS read port
169                                  if harc_LS_wire = h then
170          sc_addr_rd(h)(sc_word_count_wire + (SIMD)*i) <= ls_sc_read_addr;
171          end if;
172        end if;
173
174        if dsp_sci_we(h)(i) = '1' and dsp_sci_wr_gnt(h) = '1' then        -- DSP write port;
175          for j in 0 to SIMD-1 loop        -- Loop through the sub-scratchpads
176            sc_we(h)((SIMD)*i+j)    <= dsp_we_word(h)(j);
177            sc_addr_wr(h)((SIMD)*i+j) <= std_logic_vector(unsigned(dsp_sc_write_addr(h)(Addr_Width - 1 downto SIMD_BITS+2)) + wr_offset(h)(j));
178            sc_data_wr(h)((SIMD)*i+j) <= dsp_sc_data_write_int_wire(h)(31+32*j downto 32*j);
179          end loop;
180        end if;
181
182        if dsp_sci_req(h)(i) = '1' and dsp_to_sc(h)(i)(0) = '1' and dsp_data_gnt_i(h) = '1' then        -- DSP read port 1
183          for j in 0 to SIMD-1 loop      -- Loop through the sub-scratchpads
184            sc_addr_rd(h)((SIMD)*i+j) <= std_logic_vector(unsigned(dsp_sc_read_addr(h)(0)(Addr_Width - 1 downto SIMD_BITS+2)) +
185    rd_offset(h)(0)(j));
186          end loop;
187        end if;
188        for j in 0 to SIMD-1 loop        -- Loop through the sub-scratchpads
189          if dsp_sci_req_lat(h)(i) = '1' and dsp_to_sc_lat(h)(i)(0) = '1' then        -- DSP read port 1
190            dsp_sc_data_read_int_wire(h)(0)(31+32*j downto 32*j) <= sc_data_rd(h)((SIMD)*i+j);
191          end if;
192        end loop;
193
194        if dsp_sci_req(h)(i) = '1' and dsp_to_sc(h)(i)(1) = '1' and dsp_data_gnt_i(h) = '1' then       -- DSP read port 2
```

```
195           for j in 0 to SIMD-1 loop        -- Loop through the sub-scratchpads
196             sc_addr_rd(h)((SIMD)*i+j) <= std_logic_vector(unsigned(dsp_sc_read_addr(h)(1)(Addr_Width - 1 downto SIMD_BITS+2)) +
197     rd_offset(h)(1)(j));
198           end loop;
199         end if;
200         for j in 0 to SIMD-1 loop        -- Loop through the sub-scratchpads
201           if dsp_sci_req_lat(h)(i) = '1' and dsp_to_sc_lat(h)(i)(1) = '1' then         -- DSP read port 2
202             dsp_sc_data_read_int_wire(h)(1)(31+32*j downto 32*j) <= sc_data_rd(h)((SIMD)*i+j);
203           end if;
204         end loop;
205
206         -- Allow a DSP read only if the SPM(i) being loaded belongs to another thread and the instruction is not a broadcast load (data hazard)
207         if kmemld_inflight(i) = '1' and dsp_sci_req(h)(i) = '1' and h /= harc_LS_wire and spm_bcast = '0' then
208           dsp_data_gnt_i(h) <= '1';
209         -- Allow a dsp read only when it is not currently being read by a kmemstr becuase we only have one read port (structural hazard)
210         elsif kmemstr_inflight(i) = '1' and dsp_sci_req(h)(i) = '1' and h /= harc_LS_wire then
211           dsp_data_gnt_i(h) <= '1';
212         -- Allow a DSP read if there are no current LSU accesses to SPM(i)
213         elsif kmemld_inflight(i) = '0' and kmemstr_inflight(i) = '0' and dsp_sci_req(h)(i) = '1' then
214           dsp_data_gnt_i(h) <= '1';
215         end if;
216
217         if dsp_sci_we(h) = (0 to SPM_NUM-1 => '0') then
218           dsp_sci_wr_gnt(h) <= '0';
219         -- Allow the DSP to write only if the kmemld is filling the SPM(i) of another thread
220         elsif kmemld_inflight(i) = '1' and dsp_sci_we(h)(i) = '1' and h /= harc_LS_wire and spm_bcast = '0' then
221           dsp_sci_wr_gnt(h) <= '1';
222         -- Allow the DSP to write only when the kmemstr is reading SPM(i) of another thread
223         elsif kmemstr_inflight(i) = '1' and dsp_sci_we(h)(i) = '1' and  h /= harc_LS_wire then
224           dsp_sci_wr_gnt(h) <= '1';
225         -- Allow the DSP to write if there are no current LSU accesses to SPM(i)
226         elsif kmemld_inflight(i) = '0' and kmemstr_inflight(i) = '0' and dsp_sci_we(h)(i) = '1' then
227           dsp_sci_wr_gnt(h) <= '1';
228         end if;
229
230         if kmemld_inflight(i) = '1' and dsp_sci_we(h)(i) = '0' then -- One LSU write enable request will put the ls_sci_wr_gnt to '1' if there are no ongoing
231     DSP writes to the same scratchpad
232           ls_sci_wr_gnt_replicated(h) <= '1';
233         elsif kmemld_inflight(i) = '1' and dsp_sci_we(h)(i) = '1' and (h /= harc_LS_wire) and spm_bcast = '0' then
234           ls_sci_wr_gnt_replicated(h) <= '1';
235         elsif unsigned(kmemld_inflight) = 0 then   -- All the ls_sci_we must be zero in-order to switch the ls_sci_wr_gnt back to '0'
236           ls_sci_wr_gnt_replicated(h) <= '0';
237         end if;
238       end loop;
239
240         ------------------------------------------------------------------------------------------------------------------------------------------------------
241         -- ######     ###    ########    ###      #######  #######      #######  ####### ######  #######  #######  --
242         -- ##     #    #  #       ##       #  #      ##      #  ##           ##      ## ##     # ##     #  ##        ##      # --
243         -- ##     #   #####      ##      #####     #######  ####### ###  ##      ## ######   ##     #  ####### #######  --
244         -- ##     #   ##   ##     ##     ##   ##    ## ##     ##            ##      ## ## ##   ##     #  ##        ## ##    --
245         -- ######   ##    ##    ##    ##    ##    ##  ##    #######      #######  ##  ##  ######  #######  ##  ##   --
246         ------------------------------------------------------------------------------------------------------------------------------------------------------
247
248       for i in 0 to SIMD-1 loop
249         if (to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))) = 4*i) and (i /= 0) then
250           wr_offset(h)(i-1 downto 0) <= (others => '1');
251         end if;
252       end loop;
253       for i in 0 to SIMD-1 loop
254         if (to_integer(unsigned(dsp_sc_write_addr(h)(SIMD_BITS+1 downto 0))) = 4*i) then
255           for j in 0 to SIMD-1 loop
256             if j <= (SIMD-1)-i then
257               dsp_sc_data_write_int_wire(h)(31+32*(j+i) downto 32*(j+i)) <= dsp_sc_data_write_wire(h)(31+32*j downto 32*j);
258             elsif j > (SIMD-1)-i then
259               dsp_sc_data_write_int_wire(h)(31+32*(j-(SIMD-1)+(i-1)) downto 32*(j-(SIMD-1)+(i-1))) <= dsp_sc_data_write_wire(h)(31+32*j downto
260     32*j);
261             end if;
262           end loop;
263         end if;
264       end loop;
265
266       for k in 0 to 1 loop
267         for i in 0 to SIMD-1 loop
268           if (to_integer(unsigned(dsp_sc_read_addr(h)(k)(SIMD_BITS+1 downto 0))) = 4*i) and (i /= 0) then
269             rd_offset(h)(k)(i-1 downto 0) <= (others => '1');
270           end if;
271         end loop;
272         for i in 0 to SIMD-1 loop
```

```
        if (to_integer(unsigned(dsp_sc_read_addr_lat(h)(k))) = 4*i) then
          for j in 0 to SIMD-1 loop
            if j >= i then
              dsp_sc_data_read_wire(h)(k)(31+32*(j-i) downto 32*(j-i)) <= dsp_sc_data_read_int_wire(h)(k)(31+32*j downto 32*j);
                                    elsif j < i then
              dsp_sc_data_read_wire(h)(k)(31+32*((SIMD-1)-i+(j+1)) downto 32*((SIMD-1)-i+(j+1))) <= dsp_sc_data_read_int_wire(h)(k)(31+32*j
downto 32*j);
            end if;
          end loop;
        end if;
      end loop;
    end loop;

  end process;

  end generate SPM_replicated;

end SCI;
-------------------------------------------------------------------------------------------
-- END of SCI architecture ------------------------------------------------------------
-------------------------------------------------------------------------------------------
```

4. Scratchpad Memories

```
-------------------------------------------------------------------------------------------
entity Scratchpad_memory is
  port(
      clk_i                 : in  std_logic;
      sc_we                 : in  array_2d(accl_range)(SIMD*SPM_NUM-1 downto 0);
      sc_addr_wr            : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
      sc_addr_rd            : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Addr_Width-(SIMD_BITS+3) downto 0);
      sc_data_wr            : in  array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0);
      sc_data_rd            : out array_3d(accl_range)(SIMD*SPM_NUM-1 downto 0)(Data_Width-1 downto 0)
      );
end Scratchpad_memory;

-------------------------------------------------------------------------------------------
architecture SC of Scratchpad_memory is

signal mem : array_3d(ACCL_NUM*SIMD*SPM_NUM-1 downto 0)(2**(Addr_Width-(SIMD_BITS+2))-1 downto 0)(Data_Width-1 downto 0);
signal h   : std_logic_vector(ACCL_NUM*SIMD*SPM_NUM downto 0);
attribute ram_style : string;
attribute ram_style of mem : signal is "block";

begin

 --------- replicate logic three times --------------------------------
  spm_replicas : for g in accl_range generate
  spm_banks    : for h in 0 to SIMD*SPM_NUM -1 generate

    write_logic: process(clk_i) --
    begin
      if(clk_i'event and clk_i='1') then
        sc_data_rd(g)(h) <= mem(g*SIMD*SPM_NUM + h)(to_integer(unsigned(sc_addr_rd(g)(h))));
       if sc_we(g)(h) = '1' then        --write mode
         mem(g*SIMD*SPM_NUM + h)(to_integer(unsigned(sc_addr_wr(g)(h)))) <= sc_data_wr(g)(h);
       end if; -- we
      end if; -- clk
    end process;

  end generate spm_banks;
  end generate spm_replicas;
  -- end of replicated logic ----------------------------------------------

end SC;
```

Glossary

ANN: Artificial Neural Networks

CNN: Convolutional Neural Networks

CSR: Control and Status Registers

DCNN: Deep Convolutional Neural Networks

DLP: Data Level Parallelism

FPGA: Field Programmable Gate Array

FU: Functional unit (general name for any arithmetic or logic unit)

F0x: Fault tolerant version of the T0 cores designed to make the Klessydra cores reliable in space environments prone to faults

Harc: (hardware context) a positive integer number identifying a hardware thread in the processing core.

Hart: hardware thread

IMT: Interleaved Multithreading.

IoT: Internet of Things.

ILP: Instruction Level Parallelism.

IPC: Instructions per Cycle.

IRQ: interrupt request.

ISA: Instruction Set Architecture.

Klessydra: the name of the family of processing cores reported in this manual.

MIPS: Millions of Instructions Per Second.

NT: Number of Active harts in the core

Modelsim: RTL Simulator.

OOO: Out-of-order architecture.

PULP: an open-source multi-core processor architecture.

PULPino: an open-source System-on-Chip single-core microcontroller architecture.

ReLu: Rectified Linear Unit, it rectifies negative values to zero.

RI5CY: Generic four-stage pipeline Riscy core from Pulpino

RISC: Reduced Instruction Set Computing

RISC-V: Open RISC instruction set architecture.

S0: a core belonging to the Klessydra family featuring single-thread execution at minimum hardware cost

SIMD: Single Instruction Multiple Data

SPE: Special Purpose Engine, the engine the executes the SPMU instruction

SPI: Scratchpad Memory Interface, that is the interface that manages the communications between the SPE, LSU, and SPMs.

SPM: Scratchpad memory, which is a local memory accesses by the LSU and SPE

SPMU: Special Purpose Mathematical Unit, this is the hardware accelerator of the T13, that has two integrated entities. The SPE and SPI.

T0: an IMT implementation in the Klessydra family, supporting interleaved multiple thread execution

T1: upgraded version of the T0 core designed to widen the target applications of Klessydra through hardware acceleration

TLP: Thread Level Parallelism

TPS: Thread Pool Size, is the number of hardware threads in the core

TPB: Thread Pool Baseline, is the minimum baseline required to not have any pipeline stalls

Vivado: Software Suite for Synthesizing RTL on XILINX FPGAs

VGG16: A deep fully connected convolutional neural networking algorithm, used for image recognition

Zero-Riscy: Generic two-stage pipeline Riscy core from Pulpino

Bibliography

[1] Shilov, Anton. *"Samsung Completes Development of 5nm EUV Process Technology"*. www.anandtech.com.

[2] Shilov, Anton. *"TSMC: First 7nm EUV Chips Taped Out, 5nm Risk Production in Q2 2019"*

[3] Moore, Gordon E. *(1965-04-19). "Cramming more components onto integrated circuits". Electronics.*

[4] Omura, Yasuhisa, Abhijit Mallik, and Naoto Matsuo. *MOS Devices for Low-voltage and Low-energy Applications*. John Wiley & Sons, 2017.

[5] Ge, Fen, Ning Wu, Hao Xiao, Yuanyuan Zhang, and Fang Zhou. "Compact Convolutional Neural Network Accelerator for IoT Endpoint SoC." *Electronics* 8, no. 5 (2019): 497.

[6] Samie, F.; Bauer, L.; Henkel, J. "From Cloud Down to Things: An Overview of Machine Learning in Internet of Things". IEEE Internet Things J. **2019**, 4662, 1.

[7]. A. Waterman, K. Asanovic, Ed., The RISC-V Instruction Set Manual - Volume I: User-Level ISA - Document Ver-sion 2.2, May 2017. [Online] https://riscv.org/specifications/

[8]. A. Waterman, K. Asanovic, Ed., The RISC-V Instruction Set Manual - Volume II: Privileged ISA - Document Ver-sion 1.10, May 2017. [Online] https://riscv.org/specifications/

[9] *RISC-V Cores and SoC Overview". RISC-V. 25 September 2019. Retrieved 5 October 2019.*

[10] Rossi, Davide, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. "PULP: A parallel ultra low power platform for next generation IoT applications." In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1-39. IEEE, 2015.

[11] Cheikh, A., Cerutti, G., Mastrandrea, A., Menichelli, F., Olivieri, M., "The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes", Proc. of APPLEPIES 2017, *Lecture Notes in Electrical Engineering*, 2018, Springer.

[12] Abbas, Z.; Mastrandrea, A.; Olivieri, M., A Voltage-Based Leakage Current Calculation Scheme and its Application to Nanoscale MOSFET and FinFET Standard-Cell Designs, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 22(12), pp. 2549-2560, Dec. 2014.

[13] M. Makni, M. Baklouti, S. Niar, M. W. Jmal and M. Abid, "A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems," *11th Int. Design & Test Symposium (IDT)*, Hammamet, 2016, pp. 154-159.

[14] Trevor Martin, Ed., *Designer's Guide to the Cortex-M Processor Family*;2nd Edition; 2016. Elsevier.

[15] Olivieri, M., Cheikh, A., Cerutti, G., Mastrandrea, A., & Menichelli, F.,Investigation on the optimal pipeline organi-zation in RISC-V multi-threaded soft processor cores. In Proc. of 2017 New Generation of CAS (NGCAS),(pp. 45-48). IEEE.

[16] Bechara, C., Berhault, A., Ventroux, N., Chevobbe, S., Lhuillier, Y., David, R. and Etiemble, D., 2011, December. A small footprint interleaved multithreaded processor for embedded systems. In *2011 18th IEEE International Confer-ence on Electronics, Circuits, and Systems*(pp. 685-690). IEEE.

[17] Traber, A., Zaruba, F., Stucki, S., Pullini, A., Haugou, G., Flamand, E., Gurkaynak, F.K. and Benini, L., 2016, Janu-ary. PULPino: A small single-core RISC-V SoC. In *3rd RISCV Workshop*.

[18] Conti, F. "An open-source microcontroller system based on RISC-V Pulpino free open source GitHub repository"

[19] Pulpino custom RI5CY toolchain "ri5cy_gnu_toolchain on GitHub featuring patches for zero_riscy and riscy cores"

[20] Blasi, L., Vigli, F., Cheikh, A., Mastrandrea, A., Menichelli, F., Olivieri, M., A RISC-V Fault-Tolerant Microcon-troller Core Architecture Based on a Hardware Thread Full-Weak protection and a Thread-Controlled Watch-Dog Timer, In: *Applications in Electronics Pervading Industry, Environment and Society. ApplePies*. 2019.

[21] S. Gupta, N. Gala, G.S.Madhusudan e V.Kamakoti, «SHAKTI-F: A Fault Tolerant Microprocessor Architecture,» in *2015 IEEE 24th Asian Test Symposium*, 2015.

[22] F. Menichelli and M. Olivieri, "Static Minimization of Total Energy Consumption in Memory Subsystem for Scratchpad-Based Systems-on-Chips," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17, no. 2, pp. 161-171, Feb. 2009

[23] Olivieri, M., Menichelli, F., Mastrandrea, A., Optimal pipeline stage balancing in the presence of large isolated interconnect delay (2017) Electronics Letters, 53 (4), pp. 229-231.

[24]. Malavenda, C.S., Menichelli, F., Olivieri, M., "Delay-tolerant, low-power protocols for large security-critical wireless sensor networks", (2012) Journal of Computer Networks and Communications.

[25] Malavenda, C.S., Menichelli, F., Olivieri, M., "A regulation-based security evaluation method for data link in wireless sensor network", (2014) Journal of Computer Networks and Communications.

[26]. Malavenda, C.S., Menichelli, F., Olivieri, M., "Wireless and Ad Hoc sensor networks: An industrial example using delay tolerant, low power protocols for security-critical applications", (2014) Lecture Notes in Electrical Engineering, 289, pp. 153-162.

[27] Jim Duffy, "8 Internet things that are not IoT" https://www.networkworld.com/ . June, 26, 2014

[28] Sun, Yi, Ding Liang, Xiaogang Wang, and Xiaoou Tang. "Deepid3: Face recognition with very deep neural networks." *arXiv preprint arXiv:1502.00873* (2015).

[29] Genesys 2 Reference Manual by Digilent, [Online] https://reference.digilentinc.com/reference/programmable-logic/genesys-2/reference-manual

[30] XILINX 7-Series User Guide and reference manual https://www.xilinx.com/video/fpga/7-series-fpga-overview.html

[31]Cheikh.A, Klessydra-T02, "A multi-threaded microprocessor interleaving as minimum two harts, which is pin-to-pin compatible with pulpino riscy cores"

[32] Cheikh.A, Klessydra-T03, "A multi-threaded microprocessor interleaving as minimum three harts, which is pin-to-pin compatible with pulpino riscy cores"

[33] Cheikh.A Klessydra-T13, "An Extended Version of the T0x multithreaded cores, with custom vector instructions, and superscalar execution. The core is pin-to-pin compatible with the pulpinor-iscy cores"

[34] Blasi.L,Vigli,F Klessydra-F03, "A fault tolerant version of the T03x core, using triple redundancy approach to ensure fault tolerance"

[35] RISC-V GNU Toolchain " https://github.com/riscv/riscv-gnu-toolchain"

[36] *Steinke, Stefan; Lars Wehmeyer; Bo-Sik Lee; Peter Marwedel (2002). "Assigning Program and Data Objects to Scratchpad for Energy Reduction" (PDF). University of Dortmund. Retrieved 3 October 2013.*: "3.2 Scratchpad model .. The scratchpad memory uses software to control the location assignment of data."

[37] Rajeshwari Banakar, Scratchpad Memory : A Design Alternative for Cache. On-chip memory in Embedded Systems // CODES'02. May 6–8, 2002

[38] Huthmann, Jens, Julian Oppermann, and Andreas Koch. "Automatic high-level synthesis of multi-threaded hardware accelerators." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-4. IEEE, 2014.

[39] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A unified graphics and computing architecture." *IEEE micro* 28, no. 2 (2008): 39-55.

[40] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

[41] Tindall, Lucas, Cuong Luong, and Andrew Saad. "Plankton classification using vgg16 network." (2015).

[42] Liu, Bin, Xiaoyun Zhang, Zhiyong Gao, and Li Chen. "Weld Defect Images Classification with VGG16-Based Neural Network." In *International Forum on Digital TV and Wireless Multimedia Communications*, pp. 215-223. Springer, Singapore, 2017.

[43] Rezende, Edmar, Guilherme Ruppert, Tiago Carvalho, Antonio Theophilo, Fabio Ramos, and Paulo de Geus. "Malicious software classification using VGG16 deep neural network's bottleneck features." In *Information Technology-New Generations*, pp. 51-59. Springer, Cham, 2018.

[44] 7 Series FPGAs Memory Resources User Guide Xilinx https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[45] UltraScale Architecture DSP Slice User Guide - Xilinx

"https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf"

[46] SIMD Instructions Considered Harmful "https://www.sigarch.org/simd-instructions-considered-harmful"

[47] Vector vs SIMD: Dynamic Power Efficiency "https://massivebottleneck.com/2019/02/17/vector-vs-simd-dynamic-power-efficiency/"

[48] Vivado Design Suite User Guide: Using Constraints "https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug903-vivado-using-constraints.pdf"

[49] R. M. Tomasulo "An Efficient Algorithm for Exploiting Multiple Arithmetic Units" IBM Journal of Research and Development

[50] J.A. Farrell ; T.C. Fischer "Issue logic for a 600-MHz out-of-order execution microprocessor" EEE Journal of Solid-State Circuits (Volume: 33 , Issue: 5 , May 1998)

[51] B.A. Gieseke ; R.L. Allmon ; D.W. Bailey ; B.J. Benschneider ; S.M. Britton ; J.D. Clouser ; H.R. Fair "A 600 MHz superscalar RISC microprocessor with out-of-order execution" 1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers

[52] Gautschi, Michael, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25, no. 10 (2017): 2700-2713.

[53] Garofalo, Angelo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors." *Philosophical Transactions of the Royal Society A* 378, no. 2164 (2020): 20190155.