CHALMERS
UNIVERSITY OF TECHNOLOGY

Implementation of a RISC-V Processor with Hardware Accelerator

Bachelor thesis in Computer Science

LUDVIG BLOMKVIST
JONAS IBRAHIMOGLU OSCARSSON
LUCAS NILSSON
ADAM STENSEKE
JOAKIM WENNERBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Bachelor thesis 2019

Implementation of a RISC-V Processor with Hardware Accelerator

LUDVIG BLOMKVIST

JONAS IBRAHIMOGLU OSCARSSON

LUCAS NILSSON

ADAM STENSEKE

JOAKIM WENNERBERG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden 2019

Implementation of a RISC-V Processor with Hardware Accelerator
LUDVIG BLOMKVIST, JONAS IBRAHIMOGLU OSCARSSON, LUCAS NILSSON, ADAM STENSEKE, JOAKIM WENNERBERG

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Examiner: Miquel Pericàs, Department of Computer Science and Engineering

Bachelor's Thesis 2019
Department of Computer Science and Engineering
Division of Computer Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: RISC-V, Author: RISC-V foundation, Public Domain

Typeset in LaTeX. Template made by David Frisk

Gothenburg, Sweden 2019

Implementation of a RISC-V Processor with Hardware Accelerator
LUDVIG BLOMKVIST, JONAS IBRAHIMOGLU OSCARSSON,
LUCAS NILSSON, ADAM STENSEKE, JOAKIM WENNERBERG
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Due to the approaching end of Moore's law, there are limits to how fast software can run on processors even in the future. A possible solution is to start doing calculations in hardware instead of software, as this is significantly faster. This project aimed to develop a hardware accelerator for a RISC-V core. This was accomplished by designing a hardware matrix multiplier to run alongside a RISC-V processor. Through testing, it was shown that hardware matrix multiplication is significantly faster than the equivalent computation in software. With this result, the project group was able to implement a method to compute matrices which is much faster than software calculation and can be used for commercial purposes.

Sammandrag

I och med det stundande slutet på Moores lag kommer begränsningar på hur snabbt program kan exekveras på processorer även i framtiden. En möjlig lösning är att börja utföra vissa beräkningar i hårdvara istället för i mjukvara, eftersom detta är betydligt snabbare. Det här projektet eftersträvade att utveckla en hårdvaruaccelerator för en RISC-V-kärna. Detta gjordes genom att konstruera en matrismultiplikator som körs tillsammans med en PULPino-processor. Genom tester visades det att matrismultiplikation i hårdvara är betydligt snabbare än likvärdig beräkning i mjukvara. Med detta resultat lyckades projektgruppen implementera en metod för matrisberäkningar som är mycket snabbare än mjukvaruberäkning och som kan användas i kommersiella syften.

Acknowledgements

We would like to thank Pedro Petersen Moura Trancoso at Chalmers University of Technology for his supervision, guidance and encouragement during the course of the project and for providing us with the required equipment.

We would like to thank the people at University of California, Berkeley, who were involved in the founding and development of the RISC-V instruction set architecture and the many volunteers who have contributed to its further development.

We would also like to thank the people involved in the PULP project at ETH Zürich and The University of Bologna for creating RISC-V implementations, such as the PULPino used for this bachelor's thesis, and making them freely available to use.

Gothenburg, May 2019

Contents

Glossary

- **APB (Advanced Peripheral Bus):** standard for a communication bus, similar to AXI but with a slightly simpler implementation, suitable for low bandwidth control accesses, e.g. register interfaces on system peripherals.
- **AXI (Advanced Extensible Interface):** standard which describes a high performance, high clock frequency, on chip communication bus, suitable for embedded micro controllers.
- **FPGA (Field Programmable Gate Array):** programmable integrated circuit that can realise functionality specified by code written in hardware description languages.
- **HDL (Hardware Description Language):** language used to describe the behaviour of logic circuits.
- **ISA (Instruction Set Architecture):** interface specification between user and a processor; set of instructions used by a processor.
- **Parallel Ultra Low Power (PULP) platform:** a set of low-power implementations of the RISC-V ISA of which one is used in this project.
- **PULPino:** microcontroller implementation belonging to the PULP platform. Rather than just being a processor core, it also has memory, buses, and various peripherals that are utilised in this project such as UART and SPI.
- **OEM (Original Equipment Manufacturer):** company that builds/assembles a finished product and delivers it to the market.
- **RISC-V:** standardised open ISA specifications.
- **SPI (Serial Peripheral Interface):** standard for synchronous, bi-directional serial communication. De facto standard in a lot of embedded systems.
- **SystemVerilog:** A common HDL. This is the language that is used in this project.
- **UART (Universal Asynchronous Receiver/Transmitter):** hardware device able to send and receive data serially. It is asynchronous in the sense that both sender and receiver have to know the transmission speed being used.
- **VHDL (Very High Speed Integrated Circuit Hardware Description language):** A common HDL.

1
Introduction

Computation algorithms have for a long time usually been implemented using software, as this is convenient and relatively easy to do. It is however far more efficient to implement them in hardware, as this increases the computational speed and efficiency, which is becoming even more relevant as of recently. Increases in processor speed can no longer be relied upon to the same extent as in the past in order to speed up the execution of algorithms. This is a consequence of the impending end of Moore's law.

1.1 End of Moore's Law

Moore's law is the name given to an observation made by Gordon Moore, co-founder of Intel Corporation, in an article from 1965 [1]. It states that the number of transistors that fit on an integrated circuit doubles every other year. As a result of this, processors generally got faster with time.

The chip manufacturing industry has managed to uphold Moore's Law for some 40 years, but in recent years this growth has stagnated, and some chip manufacturers have given up on Moore's law [2]. It is no longer economically viable to pursue further increases in transistor density, as the physical constraints have become too difficult to overcome [1].

The end of Moore's law has implications for software programmers. Previously, programs would get faster as the processors running the software achieved faster clock speeds. With the end of Moore's law, it is no longer possible to rely on processors getting faster with time to the same extent, in order to achieve better performance. One suggestion to overcome this problem is to design specialised hardware accelerators to perform certain calculations, as this is much faster than doing the same calculations in software [3]. Examples of these are graphics cards, which are hardware accelerators for graphics calculations. While a processor is able to do these calculations, they will be performed much slower than they would on a graphics card.

A current problem is that modifying hardware to fit your own use may have serious ramifications if proprietary hardware is used and/or modified. Asanović and Patterson have made arguments against proprietary Instruction Set Architectures (ISAs) for a number of reasons [4], some of which will be presented in this paragraph. Firstly, a license to use proprietary hardware from ARM, IBM and Intel costs up to ten million dollars, and it comes with restrictions on how you are allowed to use or modify the hardware for your needs. Secondly, the companies do not have exclusive

knowledge and experience of how to design an ISA. Lastly, proprietary standards are not guaranteed to last; when a company goes bankrupt, all the hardware that was designed by the company will be lost, or at least not OEM supported.

Hardware accelerators need controllers in order to work, but as discussed in the previous paragraph, using a controller with a proprietary ISA carries a big monetary cost. It is therefore more feasible for developers to use a controller with an open ISA to develop hardware accelerators. Such a controller can be modified to fit the developers' needs, without incurring licensing fees. An example of an ISA that is open is the RISC-V, which will be the focus in this project.

A domain in which the use of hardware acceleration is appealing is linear algebra. Operations on large matrices require relatively much computation time. As such, it would be useful to speed up these calculations so that domains that are heavy users of linear algebra can more efficiently perform them. The RISC-V ISA does not offer these matrix operations by default, but because of its open-source nature, they can be added by users as needed [5].

1.2 Objective

The purpose of this project is to design a hardware accelerator for matrix operations and implement it together with a RISC-V processor. The accelerator will support a set of operations. After the hardware accelerator is implemented, its performance will be evaluated and compared against the RISC-V core doing the same calculations in software. It will then be compared with other alternative ways of doing equivalent matrix calculations.

1.2.1 Demarcations

The hardware accelerator will only support matrices consisting of integer values, it will not support floating point matrix operations or matrices with complex numbers. Adding support for these features would lead to increased complexity and would exceed the scope of the project without contributing further towards the objective.

The scope of what operations the accelerator will support is limited due to time constraints of the project. The main focus is not to implement as many operations as possible, but rather to implement an accelerator that supports one or more operations together with a RISC-V processor.

1.3 Report Structure

This report is hereinafter divided into four sections; technical background, methods, results, and discussion.

In the technical background section, theoretical background knowledge that is useful for understanding the project is presented to the reader. Discussion on different algorithms for common hardware accelerated operations is presented. Furthermore, there is a discussion on how algorithms can be implemented in hardware.

The method section contains information on how the hardware accelerator was implemented in practice. Specifications of the different parts of the processor-accelerator system are presented. This section also discusses how the efficacy of the hardware accelerator is evaluated.

The result section presents which of the specifications the implemented system fulfilled, and how well the system performed in evaluation tests.

Lastly, in the discussion section the perspective is broadened to discuss the implications of this project. Test results are analysed and a conclusion is presented.

2
Technical Background

There is some background information needed in order to be able to implement a hardware accelerator for matrix operations. Firstly, knowledge about RISC-V will be necessary since it is the open ISA that will be used in this project. Secondly, theoretical knowledge about algorithms for matrix calculations will be necessary to decide how the matrix operations should be implemented. Finally, knowledge about digital design and hardware description languages is also necessary so that said algorithms can be implemented in hardware.

2.1 RISC-V

Proprietary ISAs are the intellectual property of the companies who own them, which puts restrictions on what changes, if any, that customers are allowed to do on the hardware level [4]. Firstly, licenses to use proprietary ISAs can cost up to several million dollars [4]. Secondly, these licenses often will not let licensees modify the hardware components [4]. These aspects of proprietary ISAs can make them difficult to use for hardware developers. A possible solution is ISAs like RISC-V.

RISC-V is a project aiming to create an open and free ISA, which can even be commercialised without having to pay licence fees [6]. This ISA was originally conceived at University of California, Berkeley [6].

RISC-V is, as the name suggests, a RISC based ISA [5]. There are different versions of the ISA supporting 32-bit, 64-bit and 128-bit architectures [5]. RISC-V supports a number of core instructions, and any additional features or instructions can be implemented as needed, thanks to its open source nature.

Due to the existence of C compilers for RISC-V [7], it is not necessary to learn RISC-V assembly language. Additionally, a hardware accelerator that is separate from the processor core will be created in this project, which does not require detailed knowledge about the ISA. As such, discussion regarding the finer details of the architecture will be left out in this report.

2.2 Algorithms for Common Matrix Operations

In this section, the algorithms that could be used to implement the hardware will be described. Various implementations are possible with various complexity.

The most simple operations to implement are the matrix addition and subtraction operations. They are binary operations defined for matrices of equal dimensions,

where each element of the first matrix is added or subtracted with the element on the same position in the second matrix.

Doing matrix additions and subtractions in hardware should be faster for big matrices than in software. In software, operations with matrices must be calculated sequentially, but in hardware, many operations can be parallelised, resulting in less overall instructions, and consequently higher performance.

When looking at which algorithms to implement, certain things need to be taken into consideration. Firstly, the asymptotic complexity is an important factor as algorithms with lower orders of complexity should take less time to calculate, especially for big matrices. Secondly, how difficult they are to implement in hardware and if the shortened calculation time is worth the added complexity. Lastly, to how useful they are and how frequent the operations would be used in practice.

That being said, looking at the asymptotic complexity can be misleading, as it only shows how fast an algorithm is as the number of elements goes to infinity. In this project, only fairly small matrices are dealt with, so considerations must also be made regarding how fast an algorithm is for a small number of elements.

2.2.1 Matrix Multiplication Algorithms

A central operation in linear algebra is the matrix multiplication, which can be calculated with a number of different algorithms. The most simple algorithm to implement is the naïve matrix multiplication algorithm, which works by following the basic definition of matrix multiplication. This algorithm has a complexity of $\mathcal{O}(n^3)$ [8].

While there are algorithms that are faster than the naïve matrix multiplication algorithm, they are not straightforward to implement, be it in software or hardware. One such example is Strassen's algorithm, which has a complexity of $\mathcal{O}(n^{2.81})$ [8]. This complexity is lower than the complexity of the naïve matrix multiplication algorithm, but is also harder to implement and probably does not lead to better performance for small matrices, based on the amount of instructions required to calculate each element. In other words, Strassen's algorithm is expensive for small matrices but the amount of computation needed scales slower than naïve matrix multiplication, making it a cheaper algorithm if the matrix dimensions are large enough.

Another algorithm for matrix multiplication is the Coppersmith-Winograd algorithm, which has an asymptotic complexity of $\mathcal{O}(n^{2.376})$ [9] [10]. While this algorithm clearly has a better asymptotic complexity, it is also a complicated algorithm to implement and will, much like Strassen's algorithm, likely lead to slower performance for any matrices that will be calculated in the scope of this project.

There is a recently discovered algorithm for matrix multiplication with slightly better complexity of $\mathcal{O}(n^{2.373})$ [10]. This algorithm, much like Coppersmith-Winograd and Strassen's algorithms, will most likely also lead to slower performance for small matrices.

Hardware accelerators for small matrices would benefit more from implementing the naïve matrix multiplication algorithm. One matrices get large enough, more performance can be gained from using one of the other three algorithms discussed.

2.3 Implementation in Hardware

The project group has considered two ways to implement a processor with hardware matrix calculation support. The first way is through an instruction set extension, and the second way is through implementation of a hardware accelerator that runs in parallel with the processor. Although a hardware accelerator will be implemented in this project, the instruction extension method will be discussed briefly.

An instruction extension requires changing the structure of the processor in order to support new instructions. This is a rather complex task but should theoretically allow for faster matrix calculations than having a hardware accelerator external to the processor core, as there is less communication overhead.

A hardware accelerator on the other hand offers advantages such as being easier to implement and having the processor core still conform with the basic RISC-V ISA. Another advantage of designing a hardware accelerator is that it can be used with any processor, as long as there is an interface between the processor and the hardware accelerator.

It is possible to synthesise code, describing hardware and written in a hardware description language (HDL), either in the form of application specific integrated circuits (ASICs) or in programmable logic devices such as field-programmable gate arrays (FPGAs) [11]. ASICs are electronic circuits which can be fully customised for the task at hand without wasting any hardware resources [11]. ASICs offer the highest performance possible and use less power and space once they are implemented. Consequently they can not be modified since the design is embedded into a piece of silicon [11]. An ASIC design is expensive and time consuming to complete but when completed the production cost per chip is relatively low, provided that the chip is produced in large enough quantities [11]. Thus, ASICs are generally chosen for high-volume manufacturing or when factors such as execution speed, low power consumption and/or reliability are of paramount importance.

On the other hand, programmable logic devices such as FPGAs can easily be reconfigured, both during product development and after product deployment [11]. This makes FPGAs suitable when development needs to be fast, simple and cheap and when the finished product needs to possess some dynamic properties, e.g. when the end product specification might change or needs frequent hardware updates [11].

In this project an FPGA will be utilised due to its reconfigurable nature, despite its inferior performance compared to ASIC in terms of use of space, speed and power consumption. Kuon and Rose showed in 2007 that an FPGA uses approximately 35 times more space and has around a fourth of the speed of an ASIC [12]. This however is not a problem since the FPGA to be used provides enough speed and power to fulfill the goals of the project. FPGAs are also used for prototyping in the development of ASIC products, and thanks to the open source nature of this project, the resulting system could potentially be realised in an ASIC without any licensing costs.

There are some physical constraints that need to be taken into account when working with an FPGA. Most notably that FPGAs have a limited amount of space, usually measured by the number of logic cells they contain, to implement the functionality specified in code. This needs to be taken into consideration to make sure

that the FPGA is able to fit both a RISC-V processor and an additional hardware accelerator. For this reason a rather small RISC-V core could be chosen so as to leave as many logic cells as possible to the accelerator portion. Even so, the accelerator will have a limited upper size and this needs to be taken into account when choosing the matrix sizes to support.

2.4 Choice of RISC-V Implementation

One important decision in this project is which RISC-V implementation to choose, as RISC-V is not a processor in itself but a set of specifications declaring what instructions processors should support. Different hardware implementations of the RISC-V architecture have different purposes, sizes, and are run at different frequencies but the basic instruction set is the same. Thus, different RISC-V implementations support the same programs as long as the programs do not use any extended instructions.

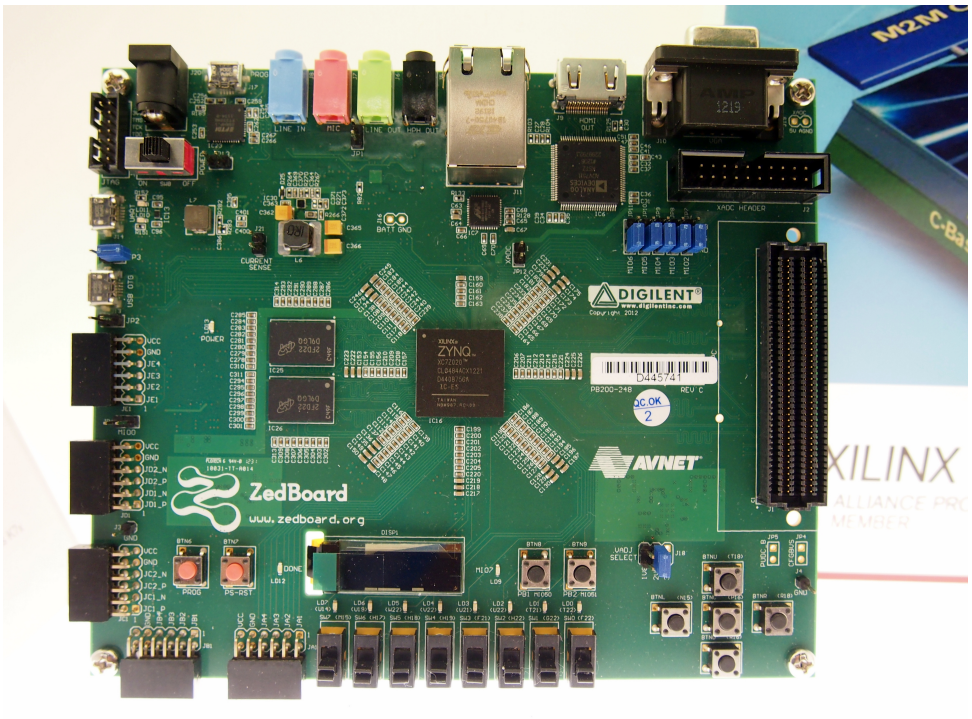

Figure 2.1: ZedBoard development board. The ZedBoard contains a Zynq-7000 SoC with a Xilinx Artix-7 FPGA, and an ARM Cortex-A9 MPCore processor, making it possible to implement combined hardware and software designs. In this project, the FPGA will be used to run a RISC-V processor with a hardware accelerator. The complete function of the system will be specified in Section 3.2. Photo licensed under CC0.

A Digilent ZedBoard development board, as seen in Figure 2.1, was provided to us. The ZedBoard contains a Zynq-7000 system on chip (SoC), which in turn contains a Xilinx Artix-7 FPGA and an ARM Cortex-A9 MPCore processor [13].

The FPGA has 85000 logic cells and the chosen RISC-V implementation must fit in these while leaving enough logic cells for an additional hardware accelerator. The data memory of the RISC-V implementation must be big enough to fit the two input matrices that are to be transferred to the accelerator for calculation as well as the output matrix that the accelerator returns. It is also an advantage if the RISC-V implementation is written in VHDL, as this is the hardware description language that most group members of the project group have experience with. Ideally, the chosen RISC-V implementation should have existing support for the ZedBoard development board, so that support for it does not need to be created for this project.

3
Methods

The introduction chapter explained that the objective of this project is to implement a RISC-V processor with a hardware accelerator. This chapter describes how the system with its microcontroller and hardware accelerator was implemented and present the specifications for it. This chapter will also describe how the performance of the hardware accelerator was evaluated and how comparisons between different methods of doing equivalent operations through other means was made.

3.1 The PULPino RISC-V Implementation

In Section 2.4, a number of requirements on the RISC-V implementation were established. No solution that met all of the established requirements was found, but the PULPino implementation of RISC-V was chosen as it came the closest to fulfilling them out of out of the different alternatives that were considered.

PULPino is an open-source microcontroller belonging to a family of RISC-V cores and systems that are part of the Parallel Ultra Low Power (PULP) Platform, a project founded by ETH Zürich and the University of Bologna in 2013 [14]. The PULPino has a 32-bit RISC-V processor and peripheral units providing support for communication with external systems, such as SPI and UART. The PULPino has enough data memory for the matrices, existing support for the ZedBoard, and occupied a relatively small part of the Xilinx Artix-7 FPGA resources, leaving room for a hardware accelerator. The default clock speed of the PULPino is 5 MHz.

However, PULPino is not written in VHDL, but in SystemVerilog. In order to simplify integration with a hardware accelerator, the project group had to learn enough SystemVerilog to be able to write the accelerator using it.

3.2 System Specifications

The main components of the system are the ARM processor running Linux, the PULPino microcontroller containing a RISC-V core, the accelerator written in SystemVerilog and an external computer to access the Linux system through the ZedBoard's Micro-USB UART port. An overview of the system is shown in Figure 3.1. There were size constraints on both the hardware and software level as the PULPino and the accelerator needed to fit together in the logic cells available in the FPGA, and the program running on the PULPino core was limited by its 32 kilobytes of instruction memory. As both hardware and software needed to be written for this project, the development of each was done in parallel with each other.

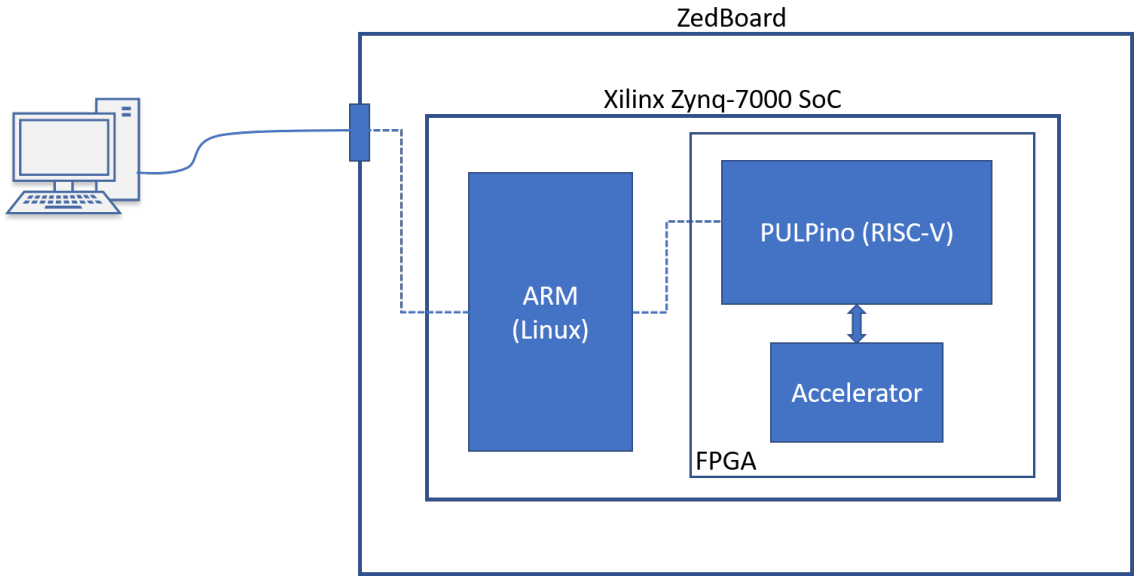

Figure 3.1: An overview of the system setup. The system is accessed externally with a computer connected to the ZedBoard's Micro-USB UART port.

The specifications for the system were divided into two categories: hardware and software specifications. The hardware specifications describe what requirements the accelerator and the whole system had to fulfill, while the software specifications describe what requirements the programs running on the ARM and the PULPino processors on the ZedBoard had to fulfill. The project group was split into two subgroups, one of which worked on the software specifications and the other on the hardware specifications.

3.2.1 Hardware Specifications

The hardware was comprised of a Digilent ZedBoard development board in which the PULPino and a hardware accelerator for matrix multiplication were implemented. The hardware had to meet the specifications in Table 3.1.

The system, including both the PULPino and the hardware accelerator, needed to be kept small enough to fit on the ZedBoard FPGA. This constraint limited the upper size of the accelerator; it was not possible to implement a hardware accelerator that was too big, which limited the amount of operations the accelerator could perform, and more importantly the size of the input matrices.

The hardware accelerator itself needed to be able to perform at least matrix multiplication on square matrices, as matrix multiplication is a common operation in linear algebra. With an accelerator supporting square matrices it would only be a matter of writing sophisticated enough software to be able to support all matrices also supported by the naïve matrix multiplication algorithm. This is due to the fact that a matrix with small dimensions can always be extended to a larger, square matrix by adding zeros to fill the corresponding rows and columns and still have the same properties when it comes to matrix multiplication. The hardware accelerator needed to be able to support operations done on at least unsigned integers, and did

not need to support any error handling in case an integer overflow happens. Matrix multiplication was an important operation to support because it is more complex than regular (element wise) multiplication and has a cubic complexity, so it would benefit from hardware acceleration.

Table 3.1: Table showing hardware specifications for the system. The finished product needed to fulfill these basic requirements in order to ensure basic functionality. The hardware specifications in this project have label prefix H. The software specifications can be found in Table 3.2.

Label	Short description
H.1	Accelerator and PULPino fit on ZedBoard FPGA.
H.2	Accelerator supports square matrices.
H.3	Accelerator can perform matrix multiplication.

3.2.2 Software Specifications

The PULPino implementation that was used in this project had been tailored for the ZedBoard. It made use of both its existing ARM Cortex A-9 processor, and the PULPino core itself which it implemented in the FPGA. Having the ARM processor handle user input and output and leaving the PULPino free to do its calculations was a potential advantage of running two separate, but communicating, programs. This approach was chosen and the specifications were therefore split into the software for the program that runs on the ARM, and the program that runs on the PULPino. Running these processors in parallel to each other enabled the PULPino to focus on doing crucial software operations and interfacing with the hardware accelerator, while the ARM processor could focus on other tasks.

The main responsibility of the program that runs on the PULPino is to do matrix calculations both with and without the hardware accelerator. The ARM processor, however, runs programs enabling miscellaneous functionality that was required for this project, such as loading data directly into the PULPino memory, presenting results provided by the PULPino to the end user and, by some means, controlling the FPGA, i.e. resetting, programming and starting the device. The specifications for the software portion of this project is described in Table 3.2.

The main purpose of the program running on the ARM processor was to facilitate communication between the user and the PULPino. The user needed to be able to send commands to the PULPino through a terminal window, and the PULPino needed to be able to send back information to the user through the terminal window. Another function that the ARM program would need was the ability to do timekeeping. In order to be able to measure how well the hardware accelerator performs compared to a non-accelerated equivalent solution, some form of timekeeping had to be implemented. Lastly, the ARM program needed to be able to transfer matrix data directly into the PULPino data memory, so the accelerator had data to do operations on. The transfer of matrix data was necessary because the PULPino was not connected to any external sources, such as sensors, which could provide the system with input data. To provide data for the accelerator to perform calculations

on, 4x4 matrices were randomly generated and loaded directly into the data memory to simulate an external source of data.

The program running on the PULPino needed to be able to communicate with the program running on the ARM processor. As the PULPino has direct access to the accelerator, it communicates with it and returns the results to the ARM program. The PULPino program needed to be able to perform matrix operations both with and without hardware acceleration and receive the result from the accelerator. Furthermore, the user had to be able to change program settings in regards to whether to multiply with or without the accelerator, as well as to be able to enter matrices manually. Lastly, the PULPino program had to be able to call the program running on the ARM processor to perform certain actions, such as starting timekeeping, requesting new input matrices and displaying the result in a terminal window.

Table 3.2: Table showing software specifications for the system. The finished product needed to fulfill these requirements. Notice that the program running on the ARM Cortex A-9 has the label prefix SA and the program running on the PULPino has the label prefix SB.

Label	**Short description**
SA.1	Support for communication with user.
SA.2	Support for timekeeping to measure speed.
SA.3	Ability to transfer matrix data to and from accelerator.
SB.1	Ability to perform matrix multiplication in software.
SB.2	Ability to perform matrix multiplication with accelerator.
SB.3	Support for user to change settings.
SB.4	Support for communication between ARM and PULPino.

3.3 Implementation of Hardware Specifications

In the selection process between extending the instruction set and implementing a separate accelerator in Section 2.3, developing a separate accelerator was chosen as this reduces the complexity and scope of the project and therefore increased the chances of successful completion within the allotted time period. There were also other benefits to this approach such as the fact that the processor is available to run instructions while the accelerator is busy doing calculations.

3.3.1 Communication between Hardware and Software

The accelerator was connected to the APB bus of the PULPino. This is a rather slow interface, but it was relatively simple to implement. There also exists an AXI bus, which is faster and more complex than the APB, through which the two main memory modules (data- and instruction RAM) and also the APB bus are connected to the core. Connecting the accelerator directly to the AXI bus would allow for faster communication but as both time and knowledge of these protocols were limited the

simpler of the two was chosen. Using the APB bus also conforms to what seems to be the de facto standard in the PULPino project as most other peripherals in the PULPino are also connected to the AXI interconnect through the APB bus. Devices connected to the APB bus are represented in the virtual memory space of the core region based on where on the bus they are connected. The accelerator is located last in the list of already connected peripherals, which gave the accelerator the virtual memory space 0x1A108004 up to 0x1A108FFF.

Any user software or other programs responsible for input to the PULPino device that wishes to do matrix multiplication need to place the desired input matrices at a specific memory address. The accelerator works with 8-bit unsigned integers placed in registers corresponding to the addresses. Its important to note that although the PULPino is byte-addressable, the accelerator only accepts whole words that contain four of these elements, as seen in table 3.3. Writing four bytes simultaneously saves execution time and helps speed up the calculation process. To perform a calculation and retrieve the 16-bit answer, the matrix data has to be written to or read from addresses as shown in Tables 3.3 and 3.4, respectively. The accelerator operations are done through combinational logic. In combinational logic, the current output only depends on the current input. This means that as the input data are written to the accelerator it will continuously fill the output memory positions with the results of the calculations so far. This also implies that the result of the calculations is ready to be retrieved as soon as the last word has been written to the accelerator, making the apparent time the actual calculations take negligible.

Table 3.3: Addresses for writing to the 4x4 matrix multiplication accelerator, and the bits interpreted as a matrix element

Address	Bit 31:24	Bit 23:16	Bit 15:8	Bit 7:0
0x1A108004:	Mat_in_1[3]	Mat_in_1[2]	Mat_in_1[1]	Mat_in_1[0]
0x1A108008:	Mat_in_1[7]	Mat_in_1[6]	Mat_in_1[5]	Mat_in_1[4]
0x1A10800C:	Mat_in_1[11]	Mat_in_1[10]	Mat_in_1[9]	Mat_in_1[8]
0x1A108010:	Mat_in_1[15]	Mat_in_1[14]	Mat_in_1[13]	Mat_in_1[12]
0x1A108014:	Mat_in_2[3]	Mat_in_2[2]	Mat_in_2[1]	Mat_in_2[0]
0x1A108018:	Mat_in_2[7]	Mat_in_2[6]	Mat_in_2[5]	Mat_in_2[4]
0x1A10801C:	Mat_in_2[11]	Mat_in_2[10]	Mat_in_2[9]	Mat_in_2[8]
0x1A108020	Mat_in_2[15]	Mat_in_2[14]	Mat_in_2[13]	Mat_in_2[12]

Data is written to the memory of the accelerator using a C script. The software writes entire 32-bit integers containing four 8-bit numbers to the accelerator, and reads integers containing two 16-bit numbers when receiving the solution matrix.

3.3.2 Internal Structure of Hardware Accelerator

The general structure of the hardware accelerator can be found in Figure 3.2. Here, the naïve matrix multiplication algorithm was chosen, as it is the simplest and fastest for small matrices, as discussed in Section 2.2.1.

The matrix data is loaded from the APB into a specific register corresponding to each element in the matrix. From here the output of the registers fan out into

Table 3.4: Addresses for reading from the 4x4 matrix multiplication accelerator, and the bits representing each matrix element in the address

Address	Bit 31:16	Bit 15:0
0x1A108004:	Mat_ut[1]	Mat_ut[0]
0x1A108008:	Mat_ut[3]	Mat_ut[2]
0x1A10800C:	Mat_ut[5]	Mat_ut[4]
0x1A108010:	Mat_ut[7]	Mat_ut[6]
0x1A108014:	Mat_ut[9]	Mat_ut[8]
0x1A108018:	Mat_ut[11]	Mat_ut[10]
0x1A10801C:	Mat_ut[13]	Mat_ut[12]
0x1A108020:	Mat_ut[15]	Mat_ut[14]

several blocks of adders and multipliers, each block calculating the value of one cell in the output matrix. Since this is all combinational logic, the result is ready almost instantly. In order to reduce the risk of overflow, the signals into these blocks were extended to a larger size. For 4x4 matrix multiplication, the minimum number of bits required to completely negate the risk of overflow is 18, see Equation 3.1. However the 4x4 accelerator was made with 16-bit outputs to save space, so overflow can happen in this implementation. The accelerator currently has no way of discovering or dealing with it, but this is listed in section 5.3.1 as an extended product goal.

$$B_{min} = log_2(2^8 * 2^8 * 4) \tag{3.1}$$

The output is stored in registers and is then put back out on the APB when reading is performed.

3.4 Implementation of Software Specifications

The software development went through multiple stages and different ideas were tested, discarded and re-engineered. It was not obvious how to utilise the ARM processor running Linux, if at all, for anything other than uploading program code to the PULPino with a program called SPI Loader, as described in the instructions for the PULPino. SPI Loader utilises an SPI peripheral, connected to the AXI bus of the PULPino, through which it can directly read from and write to the memory without involving the processor core. This is how compiled C code can be written to the instruction memory, as well as any other data written to the data memory. After data has been written, SPI Loader restarts the PULPino which then starts executing the program.

The software component of the system consists of two parts; a C program running on the PULPino, and a Linux shell script running on the ARM processor. The C program can perform matrix multiplication in software and with the hardware accelerator, as well as receive commands and transmit data through the PULPino UART. The Linux shell script provides keyboard input and allows users to send commands, and see received data in a terminal window. While the shell script has an intuitive menu system and automatically formats any data received in a fashion

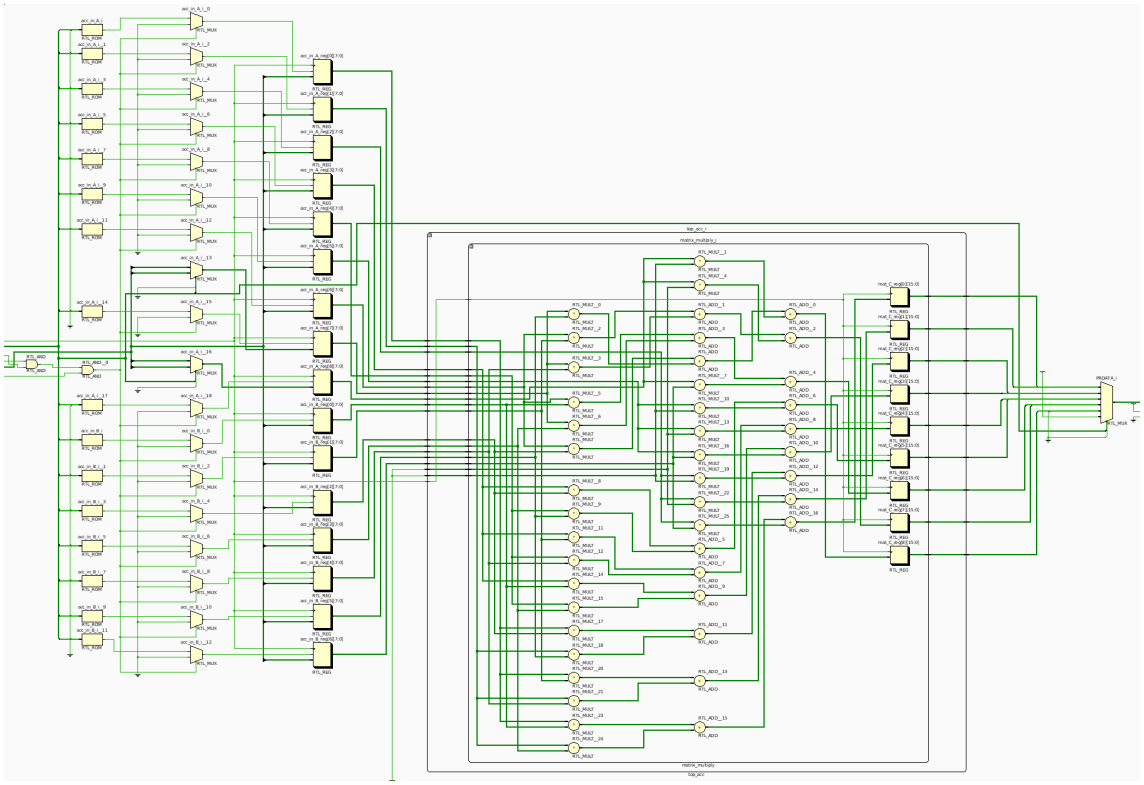

Figure 3.2: The internal structure of the 4x4 matrix multiplication accelerator. Data wires are simplified into a single bus line.

appropriate for screen output, the PULPino program was implemented so that any device with UART capability can control and receive data from the PULPino.

Because of how the PULPino implementation for the ZedBoard relies on the ARM processor to transfer program code to the PULPino, Linux and therefore the ARM processor would be used at least for that purpose. To start with it was unknown how big instruction memory the RISC-V core would have, so it was unclear how big of a program it could fit. When the PULPino was chosen and implemented, its default instruction memory size of 32 kilobytes was set.

The next thing to consider was how to communicate with the RISC-V core. Initially it was decided to use an external keypad and display. This was with the intent to make the PULPino and the accelerator a standalone system, not depending on the ARM processor for input and output. After a while this was changed to instead using UART communication between the ARM (Linux) and the PULPino as this would allow for a better user experience through a terminal window.

It was decided that 8-bit integers would be used for each of the cells in the input matrices. This was initially to make sure that the data memory would be able to fit two 32x32 input matrices as well as a 32x32 output matrix. When the basic design was agreed upon and groups had been created for the hardware and software part, some time was spent getting familiar with how the software interacted with the processor, how to compile code and to find potential bugs during compilation.

3.4.1 The PULPino Program

The program running on the PULPino core, available in Appendix B.2, continuously monitors the UART port to receive any incoming commands. Whenever it receives the ASCII character '>', it reads all incoming characters and saves them in a string until it encounters the ASCII '\0' (null) character. A full UART message is therefore of the ASCII format: ">string\0".

Through a series of if-else statements, the string is compared with the set of supported commands and if it is a valid command, the appropriate actions are taken. This usually means performing a task, as doing matrix multiplication, and sending back data/messages in ASCII format through the UART. If the string does not correspond to any of the supported commands, an error message is sent back. Valid commands, together with their functions, are listed in Table 3.5. This setup, in which input and output to the PULPino is done through UART messages, makes it possible for the PULPino and the hardware accelerator to be controlled and used by any device capable of serial/UART communication. For debugging purposes an oscilloscope with a UART decoder can be used to read data, sent by the PULPino program in ASCII format.

When the program receives a request to begin a calculation, it checks the value in the variable "accelerator" to decide whether to perform the calculation in software or to utilise the hardware accelerator. If it is to be done in software, it reads the data memory locations reserved for matrix A and matrix B, employs the naïve matrix multiplication algorithm and writes the result back into the memory location reserved for the result matrix. If it is to be done in the hardware accelerator, the program writes matrices A and B to the accelerator address space, effectively transferring the data through the APB-bus, as described in Section 3.3.1.

In both the software and hardware accelerator cases, in order to count the clock cycles, a PULPino timer register is reset and started right before calculation begins. It is then stopped right after the last matrix cell of the result has been written to the data memory position reserved for the result matrix. The cycle count is sent through UART upon receiving the command "get time".

3.4.2 The Linux Scripts

To act as an interface between the user and the matrix calculation capabilities of the PULPino system and its hardware accelerator, the Linux system running on the ARM processor is used. Two shell scripts were written for this purpose. The main script (start.sh) is what is run to begin using the project system. It is used to present a menu through which commands can be sent to and data received from the PULPino program. It in turn makes use of the second script (load_matrix.sh) which reads text files comprised of ASCII numbers in a 4x4 format, converts the data to the format used by SPI Loader, and then uses SPI Loader to upload that data to the PULPino data memory positions reserved for the two input matrices. An illustration of this is shown in Figure 3.3 in which a text file containing 16 cells of the number two is transferred to the memory location of input matrix A (0x100400+).

Table 3.5: A list of commands that can be sent to the PULPino program through UART. Any device capable of transmitting and receiving UART data can access the system's matrix calculation capabilities. The results are sent back through UART in ASCII format.

Command	Function
"off"	Disable the accelerator. Used by Linux shell script to restore accelerator setting after reset caused by SPI Loader.
"on"	Enable the accelerator. Used by Linux shell script to restore accelerator setting after reset caused by SPI Loader.
"show settings"	Send current accelerator setting through UART.
"print result"	Send the result matrix stored in PULPino data memory through UART.
"accelerator off"	Disable accelerator.
"accelerator on"	Enable accelerator.
"calculate"	Begin matrix multiplication in software or using the accelerator depending on current accelerator setting. Send result through UART.
"get time"	Send the cycle count from PULPino timer register.

When run, start.sh presents a menu in which commands can be entered. Supported commands are shown in Table 3.6

Most of the commands are identical to the commands listed for the PULPino program in Table 3.5, which can also be seen in the scripts themselves in Appendix B.3. When a user enters one of those commands, the script forwards that command to the PULPino program via UART, receives the PULPinos reply, formats it and presents it to the user. The command "demo" is particularly useful as it demonstrates all the capabilites of the system in one go. First it generates random matrices and uploads them to the PULPino data memory. Then it tells the PULPino to perform matrix calculation in software. It displays the result as well as the number of cycles taken to perform the calculation. It then enables the hardware accelerator and performs the same multiplication again. The result and cycle count is once again displayed. Finally, based on the two clock cycle values, a message about the relative performance between the two calculations is shown.

3.5 Evaluation of performance

To evaluate the performance of the hardware accelerated system, matrix multiplication with and without hardware acceleration was done. In order to keep as many variables as possible constant between these two test cases, they were both run on the PULPino, and as such they had the same access speed to memory addresses. Furthermore, the timing was done in clock cycles, as measuring the execution time is dependent on the clock speed of the PULPino, which can vary depending on how it is implemented in hardware.

Counting clock cycles was done with one of the two available 32-bit PULPino

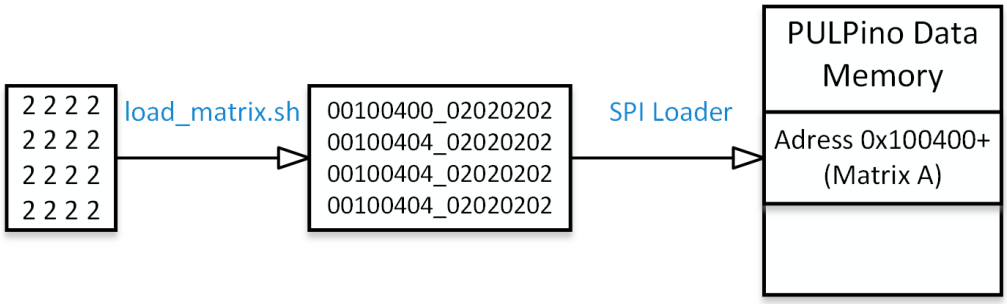

Figure 3.3: A text file comprised of matrix values is converted by load_matrix.sh into the format needed by SPI Loader and then transferred to data memory. The text files are either generated by start.sh, from manual user input, or by randomization using the Linux $RANDOM variable.

Table 3.6: A list of commands that can be entered into the Linux shell script "start.sh", which functions as the interface between users and the program running on the PULPino system.

Command	Function
"help"	List available commands.
"show settings"	Show current accelerator setting.
"enter matrices"	Allow user to manually enter matrices, then upload them.
"load matrices"	Generate two random matrices and upload to PULPino.
"print matrices"	Print the input matrices currently in PULPino data memory.
"print result"	Print the result matrix currently in PULPino data memory.
"accelerator off"	Disable accelerator.
"accelerator on"	Enable accelerator.
"calculate"	Begin matrix multiplication in software or using the accelerator depending on current accelerator setting.
"demo"	Perform a demonstration of the system.
"quit"	Exit the Linux shell script.

timer registers. These registers are incremented by one with each clock cycle. The timing starts just before the first cell of input matrix A is transferred into the accelerator, or calculated by the PULPino, and ends when the last result matrix cell has been transferred back into the data memory. Note that the transfer from the accelerator back to the data memory is in software, as hardware for doing this automatically has not been implemented.

The hardware implementation of the system was also analysed in terms of area usage on the Xilinx Artix-7 FPGA. As the FPGA had a limited amount of programmable logic cells, referred to as look up tables (LUTs), and flip-flops, this put a constraint on the dimensions of the hardware accelerator. It is useful to know how much of the FPGA resources are used by the hardware accelerator, in order to evaluate which future extensions to implement.

4
Results

In this chapter, the specifications and goals that have been met will be shown, as will the performance evaluation and comparison with other alternatives. In addition to this chapter, a step-by-step guide on how to set up the environment for compiling C code and synthesising SystemVeilog code has been added in Appendix A.

4.1 Results of Specifications

The system fulfills all of the basic specifications that were set up in Section 3.2, and thus has basic functionality. In Table 4.1, each of these specifications are followed up on. None of the extended product goals were achieved, but would be interesting additions if further development is done on this project in the future.

Table 4.1: Table following up on the specifications that were set in Section 3.2, in Tables 3.1 and 3.2.

Label	**Fulfilled?**	**Comment**
H.1	Yes	See Table 4.2.
H.2	Yes	Supports 4x4 matrices.
H.3	Yes	8-bit input, 16-bit output.
SA.1	Yes	
SA.2	Yes	Using the PULPino timer registers.
SA.3	Yes	Via SPI.
SB.1	Yes	
SB.2	Yes	
SB.3	Yes	
SB.4	Yes	Via UART.

4.2 Test Results

In this section, the results from the evaluation methods discussed in Section 3.5 are followed up on. An analysis of both the FPGA resource usage and the performance of the hardware accelerator in terms of clock cycles is done. The broader implications of these results will be further discussed in Chapter 5.

4.2.1 Utilisation of FPGA Resources

The hardware accelerator supports matrix multiplication on 4x4 matrices. Each of the input elements are 8-bit integers and the output elements are 16-bit elements. However, in order to prevent integer overflow on the output matrix, it is recommended to limit the input to 6-bits. The space required for the accelerator FPGA implementation is shown in Table 4.2. The utilisation for just the accelerator was taken as a difference between the resources used by an FPGA with just the PULPino on it, and an FPGA with both a PULPino and an accelerator. Consequently, this can therefore differ slightly from the actual number of units used due to the optimisations made by the synthesis tool, Xilinx Vivado.

Table 4.2: Hardware resources used on the Xilinx Artix-7 FPGA by the PULPino and hardware accelerator. A 4x4 matrix multiplication hardware accelerator uses around a tenth of the available LUTs on its own, putting a hardware constraint on the upper size limit of a hardware accelerator.

Type	**Available**	**Used by PULPino**	**Used by accelerator**	**Total**
Flip-flop	106400	9871 (9.3%)	511 (0.48%)	10382 (9.8%)
LUT	53200	15592 (29%)	5231 (9.8%)	20823 (39%)

With the results in the Table 4.2, it can be concluded that there is a rather low upper limit on the size of a matrix multiplication accelerator due to the amount of LUTs used with the matrix multiplication hardware. As a consequence of the low upper limit, the potential performance increase is also hindered, if not some form of tiling is implemented, as discussed in Section 5.3.1.

4.2.2 Performance Evaluation Results

As stated in Section 3.5, the tests have been run on the same PULPino configuration, and yielded the results in Table 4.3. With hardware acceleration, a result that is 4.5 times faster than the equivalent operation in software was achieved. This factor can be improved by accelerating matrices with bigger dimensions, as more and more calculations are parallelised compared to doing them in software. However, it is important to point out that in essence the actual calculation done by the hardware accelerator takes no more than one clock cycle. The rest are clock cycles spent transferring data to and from the accelerator.

Table 4.3: Test results for matrix multiplication. This table shows the number of clock cycles needed to perform matrix multiplication on a 4x4 matrix with and without hardware acceleration. With hardware acceleration, a performance improvement factor of 4.5 was achieved.

Processor	**Mode**	**Clock cycles**
PULPino	Hardware accelerated	134
PULPino	Software	603

5

Discussion

In this chapter, the test results will be analysed and the project will be discussed from a broader perspective. There will be discussion on how this project can be implemented, how it can be improved or modified to fit certain uses, and also some ethical aspects concerning the project.

5.1 Analysis of Test Results

In the introduction chapter, it was described how hardware specific solutions can be designed to increase computational performance of systems. The implications of the test results presented in Section 4.2.2 show that there is a potential for big performance improvements by using hardware solutions, thus acting as a proof of concept of this original claim. With hardware acceleration, the number of clock cycles it takes to calculate matrix multiplication was reduced to almost a fifth compared to non-accelerated computation.

One thing that the test results did not take into account is the absolute time it takes to perform matrix multiplication. While hardware acceleration did reduce the amount of clock cycles, regular processors on the market have at least 500 times faster clock speed than the default 5 MHz clock speed of the PULPino, meaning that this particular hardware accelerator still struggles to compete with non-hardware accelerated matrix multiplication on regular processors. However, there is optimisation potential by attempting to increase the clock speed or using implementing this hardware accelerator using ASICs, as discussed in Section 2.3.

New problems arise from the use of hardware solutions. Firstly, there is a problem with availability. Applications that use hardware acceleration are only compatible with an end system when a hardware accelerator is present, which makes hardware specific solution much harder to reach to the market. With an FPGA, some of the availability problems are solved since they are re-configurable, but on the other hand they suffer from comparably lower performance in terms of area usage, power consumption and performance, compared to equivalent ASIC implementations.

Another problem is, as demonstrated by the test results, that there is a limit with how much data hardware accelerators can operate on when it is implemented with an FPGA. FPGAs have a limited amount of resources in terms of flip-flops and programmable logic cells, which the hardware designer must be aware of.

5.2 Areas of Application

Matrix multiplication was found to be significantly faster in hardware, compared to software implementations. Consequently, this hardware accelerator is best used with programs that uses matrix multiplication operations frequently. Examples of such applications are different machine learning algorithms. The YOLOv3 (You Only Look Once, version 3) algorithm is a machine learning algorithm for computer vision, which uses matrix multiplication in the algorithm [15].

Through this project, a proof of concept has been demonstrated; it is possible to develop hardware using open-source tools, without having to pay license fees. This project can even be commercialised in the future. This method can not only be used to develop matrix multiplication accelerators, but all kinds of hardware solutions.

Due to hardware space limitation on the Xilinx Artix-7 FPGA, it is recommended to use a bigger FPGA or an ASIC for designing matrix multiplication accelerators. For example, in machine learning, matrices would normally be bigger than this FPGA would have space for.

5.3 Further Development

The implemented accelerator has limited functionality, which creates room for improvements. Firstly, the interface is designed in such a way that adding new operations and changing the dimensions of the accelerator is possible. The accelerator serves as a base which anyone can extend to fit their own application. Secondly, the accelerator does not necessarily need the PULPino implementation of RISC-V, or even a RISC-V processor, to work. It is possible to use any processor that uses the standardised APB interconnect.

In fact, since PULPino is a small implementation of RISC-V, it comes with constraints when extending it with a hardware accelerator. As the PULPino only has 32 kilobytes of data memory, this limits the maximum size of the matrices that can be calculated. However, this memory could be extended, although this would take up more resources on the FPGA.

This hardware accelerator design has a few drawbacks that can be handled in a future project. Firstly, moving around matrices in the memory will take time. For example, if a user calculates a matrix, and wants to use the resulting matrix in another calculation, they would have to individually move each block of elements from the output matrix space to one of the input matrix spaces. Secondly, the accelerator has no way of detecting or handling integer overflow.

Other ways to improve this product is to implement the extended product goals that were declared in Section 5.3.1.

5.3.1 Hardware Accelerator

In Tables 3.1 and 3.2, the specifications for a minimum viable prototype were listed. There are however features that are good to include to the hardware accelerator, that will be discussed in this section.

One issue that came up during the project is how much computing power is needed to synthesise hardware accelerators for matrices that have bigger dimensions. If more computing power was available, the matrix dimensions could have been bigger. As the size of the matrices being calculated increases the more time is saved by using a hardware accelerator compared to computing the equivalent in software. This is because the accelerator is constructed in such a way that, up until its limiting size, the time to compute will be constant regardless of the size of the input matrices. This differ from a calculation in software where the limit in size is much less obstructive (mostly depending on RAM) and the time to compute will grow exponentially as the input matrices becomes larger.

Recall from Section 2.2.1 that naïve matrix multiplication is an $\mathcal{O}(n^3)$ operation. This means that the bigger the input matrices are, the time it takes to sequentially calculate the product has a cubic growth. However, naïve matrix multiplication can also be made completely parallel, as each cell of the result can be computed independently of every other cell. A hardware accelerator would be able to perform matrix multiplication on every cell in parallel, so the total execution time would be unaffected as the size of the matrices grow, provided that the matrices fit in the hardware accelerator. This is not true in software, as when the problem grows, so does the execution time. With this example in mind, it would be beneficial for performance if the hardware accelerator could support big matrices, and this can be implemented if there is time and computing resources for it.

The above goal would mean that there is a need for an additional error handling feature, as operations are not defined for matrices of all dimensions. For example, it is not possible to do matrix multiplication with two matrices where the inner dimensions are not equal, because this is not in the domain of the matrix multiplication operations. However, in the scope of this project it will be assumed that input is of acceptable sizes.

There are a number of different operations that would be useful for an end user to be hardware accelerated, such as inverting a matrix, dot product, determinant, etc. An additional goal would be to implement additional operations into the hardware accelerator, and possibly additional support for these operations done on signed integers and/or floating point numbers. It is important to note that supporting additional operations does not mean they have to be performance tested later on.

Another useful feature would be to implement some form of divide and conquer technique, that could extend the accelerator to support larger matrices. By adding hardware to the design that partitions the input matrices according to what is called block matrix multiplication [16], it is possible to use an accelerator which by itself has some size limitations, i.e. dimensions $= X$, to calculate larger matrices with dimensions $> X$. This would not be as fast as natively supporting the bigger matrix size due to lesser parallelisation, but it would be a good trade-off between area usage and performance, as it still would be faster than calculating the bigger matrix in software.

5.4 Ethical Aspects

As mentioned in Section 5.2, this project can be used with machine learning, which comes with a number of ethical considerations. Machine learning can be used for purposes that can be deemed malicious or ethically questionable, such as censorship and autonomous weapons [17]. Application of machine learning can also cause mass unemployment with automation of labour [17].

These are possible, but indirect consequences of this project. However, as there are other alternatives of doing fast matrix multiplication calculations, it is outside of our control. These possible consequences can not be mitigated by not publishing the results of this project. This project most likely will have little to no effect on these ethical issues.

5.5 Concluding Remarks

Throughout the course of this project, the final product was scaled down. Initially, the hardware accelerator was going to support 32x32 matrices with 8-bit integers, and multiple operations. This was thought to be a straightforward goal to achieve. However, as many problems were encountered during synthesis attempts of the accelerator, a decision was made to shrink it until synthesis was successful. This ended up being a 4x4 accelerator. Judging by the resource use in the FPGA, as shown in the test section, it is possible that a 32x32 accelerator wouldn't have fit anyway, even if synhesis had been successful.

One of the original goals was to test accelerators of different sizes and compare these to each other in terms of speed and power consumption. It was eventually not done due to lack of time and not having a good method to compare them. It was also unclear what the method would be for evaluating power consumption for just the accelerator.

However, even with the downscaled product, the project goal was still accomplished.

In conclusion, hardware acceleration has the potential to vastly increase calculation performance compared to calculations done entirely in software. As processors are unlikely to achieve much higher clock speeds in the near future due to the end of Moore's law, hardware accelerators can be used to complement processors in order to speed up calculations.

Bibliography

[1] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[2] S. Blank, "What the globalfoundries' retreat really means," *IEEE Spectrum*, 2018. `https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means`, accessed 2019-03-15.

[3] J. Hennessy and D. A. Patterson, "John hennessy and david patterson 2017 acm a.m. turing award lecture." `https://www.youtube.com/watch?v=3LVeEjsn8Ts`, 2018. Accessed: 2019-05-15.

[4] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[6] RISC-V Foundation, "About the risc-v foundation." `https://riscv.org/risc-v-foundation/`. Accessed: 2019-06-04.

[7] GNU MCU Eclipse, "The risc-v embedded gcc." `https://gnu-mcu-eclipse.github.io/toolchain/riscv/`. Accessed: 2019-06-07.

[8] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, "Implementation of strassen's algorithm for matrix multiplication," in *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pp. 32–32, IEEE, 1996.

[9] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of symbolic computation*, vol. 9, no. 3, pp. 251–280, 1990.

[10] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd.," in *STOC*, vol. 12, pp. 887–898, Citeseer, 2012.

[11] H. R. C. Dally, Willian J. and T. R. Aamodt, *Digital Design Using VHDL*. Cambridge University Press, 2016.

[12] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 26, no. 2, pp. 203–215, 2007.

[13] *ZedBoard Hardware User's Guide, Version 2.2*. 2014.

[14] PULP Team, "Pulp platform." `https://pulp-platform.org`. Accessed: 2019-03-12.

[15] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[16] Gilbert Strang, "Lecture 3: Multiplication and inverse matrices." https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/video-lectures/lecture-3-multiplication-and-inverse-matrices/. Accessed: 2019-05-15.

[17] O. Häggström, "Blunda inte för riskerna med artificiell intelligens." http://www.dagensarena.se/essa/blunda-inte-for-riskerna-med-artificiell-intelligens/. Accessed: 2019-03-25.

A

Appendix 1

The following documentation is based off a Linux environment and the goal is to show how to get a PULPino running on the ZedBoard, and to show how C programs can be compiled for the PULPino. Note that this process might require a lot of trial and error, as things might behave differently in different environments, and information might not be up to date by the time this document is read.

Setting up the PULPino environment

To set up the development environment for the RISC-V core we followed the instructions on the PULPino GitHub repository. However, many steps are outdated and some documentation is missing, resulting in it difficulties to figure out how to compile the core and the GCC implementation in the PULPino. This section will focus on making a comprehensive documentation based on the original documentation from the PULPino GitHub page with additional information added.

Compile PULPino RISC-V core

Step 0

Start by installing the Vivado 2015.1 and Xilinx SDK toolchains. The PULPino is only tested with the Vivado 2015.1 version, so if you have a newer version, it may not work. Both of these can be found on the Xilinx web page. Remember to take note on where they are installed because we need to link them into the path in the next step.

Step 1

We are now going to create a Bash script that enable the correct environment variables. It will also link the Vivado and Xilinx SDK toolchains to the PATH.

```
#!/bin/bash

export BOARD=zedboard
export XILINX_PART="xc7z020clg484-1"
export XILINX_BOARD="em.avnet.com:zynq:zed:c"
```

```
PATH=$PATH:/usr/local/Xilinx/Vivado/2015.1/bin:
          /usr/local/Xilinx/SDK/2015.1/bin

RISCY_RV32F=0
```

In this case, a ZedBoard is used here with part number "xc7z020clg484-1" and the board used here is a "em.avnet.com:zynq:zed:c". These may have to be changed to fit the board that is being used. These are used by the Vivado and Xilinx SDK to generate the core, so it is important that they are correct. Furthermore, the PATH variable needs to point to where the Vivado and Xilinx SDK is installed.

The "RISCY_RV32F=0" environment variable is used to specify the core to not use floating point variables, if this is wanted it should be set to a "1". It also tells the compiler that we want to synthesise the RISCY core.

Step 2

Run the "./update-ips.py" script that is located in the top directory of the PULPino folder.

Step 3

Next step is to open a terminal and execute the script. This will setup all the variables and paths needed for the compilation. When done, navigate to the "fpga/" directory and execute "make all". This will build the bitstream for the processor. When done, continue following the original documentation on the PULPino GitHub repository from step 4.

Build C programs for the PULPino

Now it is expected that the FPGA contains the RISC-V core and that it is possible to communicate with the ARM processor, running Linux, using UART. The next step is to set up the GCC compiler with the PULPino so it is possible to create C programs that can run on the RISC-V core.

Step 0

The first step is to acquire the GCC for the PULPino core. The toolchain is called "ri5cy_gnu_toolchain" and can be found on GitHub.

Step 1

Compile the toolchain and place it wherever you like and grab the path to its bin/ directory. This is now added to the path in the previous Bash script, so its full PATH looks like this, "PATH=$PATH:/usr/local/Xilinx/Vivado/2015.1/bin: /usr/local/Xilinx/SDK/2015.1/bin:/usr/local/riscv/bin".

Step 2

Now it is time to link the compiler to be used with the PULPino. This part is the complicated step and may take some time before working properly. First off enter the "sw/" directory in the top folder, create a map called "build/" and then copy the file named "cmake_configure.riscv.gcc.sh" into that directory. This file have to be somewhat modified. The value of "TARGET_C_FLAGS" has to be changed to "-O3 -march=rv32g -g". "GCC_MARCH" value then has to be changed to "rv32g". The last thing to change is to do some instructions before the script is executed, so at the top of the file add the following code:

```
source [PATH TO THE BASH SCRIPT CREATED EARLIER]
mkdir -p $HOME/pulpino/sw/build/CMakeFiles/CMakeTmp/
$HOME/pulpino/sw/build/./fix_linker
```

The first line should replace the part with the square brackets with the path to the Bash script created earlier. This is just to make sure everything the compiler needs exists in the path before it starts. The second line simply creates a directory used by the script on the next line. The last one runs a script that corrects a bug that otherwise will terminate the compilation. The script looks like following:

```
#!/bin/bash

# ./fix_linker
# Fixes linker problem by creating default link script.

riscv32-unknown-elf-ld --verbose |
        head -n -1 |
        tail -n +7 |
        sed '168 a \ \ _fbss = .;' |
        sed '169 a \ \ . = .;' >
        $HOME/pulpino/sw/build/CMakeFiles/CMakeTmp/riscv.ld
```

When done and all the paths have been changed to the correct paths for each individual user, it is time to try and compile. Most likely it will not work. One common problem is that it complains about a "-m32" flag. To fix this, run the following script from the "build/" folder to find every occurrence of the "-m32" flag and remove it.

```
#!/bin/bash

for file in $(find); do
    if [[ -f $file ]]; then
        [[ $(cat $file | grep m32) ]]
        if [[ $? == 0 ]]; then
            echo writing...
            echo $file
           sed 's/\-m32//g' $file > tmp && mv tmp $file
       fi
    fi
```

done

Remember not to have this script in the "build/", only that you run it from there. Otherwise it will also scan itself and remove the string, thus corrupting the script. Sometimes the script has to be executed multiple times in order to work for unknown reasons.

Step 3

Now it is up for trial and error. Along the way we have stumbled across many other errors and have included two strange ones below. If your error is different, then read the error carefully and make sure that the steps above are done correctly.

1. usr/local/riscv/lib/gcc/riscv32-unknown-elf/7.1.1/../../../../riscv32-unknown-elf/bin/ld:riscv.ld:1: syntax error
    - Remove first 5 lines and last line in CMakeFiles/CMakeTmp/riscv.ld
2. Dereferencing pointer to incomplete type td_thr_getfpregs.c
    - Build the entire NewLib, multilib.

B

Appendix 2

B.1 HDL Code

Here the HDL code used to implement the hardware accelerator will be presented. The code is written in SystemVerilog and synthesised using Xilinx Vivado 2015.1. The code is mainly contained in three files, apb_acc.sv, matrix_multiply.sv and top_acc.sv. The rest of the code is modifications and additions of the PULPino code, which enables communication with the accelerator.

The first notable addition to the code of PULPino is in the file apb_bus.sv in the rtl/includes directory. Here, the addresses to the accelerator should be defined.

apb_bus.sv

```
// addresses for the accelerator
`define ACC_START_ADDR          32'h1A10_8000
`define ACC_END_ADDR            32'h1A10_8FFF
```

The addresses defined above can then be used in the files periph_bus_wrap.sv and peripherals.sv in order to add the accelerator to the APB.

periph_bus_wrap.sv

```
...
//port for accelerator on periph_bus_wrap module
APB_BUS.Master      accelerator_master
...
//add accelerator module as an APB master
`APB_ASSIGN_MASTER(s_masters[9], accelerator_master);
assign s_start_addr[9] = `ACC_START_ADDR;
assign s_end_addr[9]   = `ACC_END_ADDR;
```

peripherals.sv

```
...
APB_BUS s_acc_bus();
...
.accelerator_master( s_acc_bus          )
...
apb_acc apb_acc_i
```

```
    (
      .HCLK         ( clk_i          ),
      .HRESETn      ( rst_n          ),

      .PADDR        ( s_acc_bus.paddr      ),
      .PWDATA       ( s_acc_bus.pwdata     ),
      .PWRITE       ( s_acc_bus.pwrite     ),
      .PSEL         ( s_acc_bus.psel       ),
      .PENABLE      ( s_acc_bus.penable    ),
      .PRDATA       ( s_acc_bus.prdata     ),
      .PREADY       ( s_acc_bus.pready     ),
      .PSLVERR      ( s_acc_bus.pslverr    )
    );
```

From the side of the accelerator, communication with the APB is handled through the module apb_acc in apb_acc.sv.

apb_acc.sv

```
module apb_acc
  #(
  parameter APB_ADDR_WIDTH = 12
  //APB slaves are 4KB by default
  )
  (
  input  logic                          HCLK,
  input  logic                          HRESETn,
  input  logic [APB_ADDR_WIDTH-1:0]     PADDR,
  input  logic                 [31:0]   PWDATA,
  input  logic                          PWRITE,
  input  logic                          PSEL,
  input  logic                          PENABLE,
  output logic                 [31:0]   PRDATA,
  output logic                          PREADY,
  output logic                          PSLVERR
  );

  logic                   clk;
  logic [15:0][15:0]      acc_out;
  logic [15:0][7:0]       acc_in_A;
  logic [15:0][7:0]       acc_in_B;
  logic [11:0]            addr;
  logic [31:0]            data;


  top_acc
  top_acc_i
```

```
(
.clk            (       clk         ),
.acc_out        (       acc_out     ),
.acc_in_A       (       acc_in_A    ),
.acc_in_B       (       acc_in_B    )
);

assign clk = HCLK;
assign addr = PADDR[11:2]; //Accelerator is word-adressed

//Assign inputs from the bus
always_ff @(posedge HCLK)
begin
  if (PSEL && PENABLE && PWRITE)
  begin
    case (addr)
      1:
      begin
        acc_in_A[0] <= PWDATA[7:0];
        acc_in_A[1] <= PWDATA[15:8];
        acc_in_A[2] <= PWDATA[23:16];
        acc_in_A[3] <= PWDATA[31:24];
      end
      2:
      begin
        acc_in_A[4] <= PWDATA[7:0];
        acc_in_A[5] <= PWDATA[15:8];
        acc_in_A[6] <= PWDATA[23:16];
        acc_in_A[7] <= PWDATA[31:24];
      end
      3:
      begin
        acc_in_A[8] <= PWDATA[7:0];
        acc_in_A[9] <= PWDATA[15:8];
        acc_in_A[10] <= PWDATA[23:16];
        acc_in_A[11] <= PWDATA[31:24];
      end
      4:
      begin
        acc_in_A[12] <= PWDATA[7:0];
        acc_in_A[13] <= PWDATA[15:8];
        acc_in_A[14] <= PWDATA[23:16];
        acc_in_A[15] <= PWDATA[31:24];
      end
      5:
      begin
```

```
            acc_in_B[0] <= PWDATA[7:0];
            acc_in_B[1] <= PWDATA[15:8];
            acc_in_B[2] <= PWDATA[23:16];
            acc_in_B[3] <= PWDATA[31:24];
          end
          6:
          begin
            acc_in_B[4] <= PWDATA[7:0];
            acc_in_B[5] <= PWDATA[15:8];
            acc_in_B[6] <= PWDATA[23:16];
            acc_in_B[7] <= PWDATA[31:24];
          end
          7:
          begin
            acc_in_B[8] <= PWDATA[7:0];
            acc_in_B[9] <= PWDATA[15:8];
            acc_in_B[10] <= PWDATA[23:16];
            acc_in_B[11] <= PWDATA[31:24];
          end
          8:
          begin
            acc_in_B[12] <= PWDATA[7:0];
            acc_in_B[13] <= PWDATA[15:8];
            acc_in_B[14] <= PWDATA[23:16];
            acc_in_B[15] <= PWDATA[31:24];
          end
        endcase
      end
    end

    //Assign outputs to the bus from the accelerator
    always_comb
    begin
      PRDATA = '0;
      case (addr)
        1:
        begin
          PRDATA[15:0] = acc_out[0];
          PRDATA[31:16] = acc_out[1];
        end
        2:
        begin
          PRDATA[15:0] = acc_out[2];
          PRDATA[31:16] = acc_out[3];
        end
        3:
```

```
        begin
          PRDATA[15:0] = acc_out[4];
          PRDATA[31:16] = acc_out[5];
        end
        4:
        begin
          PRDATA[15:0] = acc_out[6];
          PRDATA[31:16] = acc_out[7];
        end
        5:
        begin
          PRDATA[15:0] = acc_out[8];
          PRDATA[31:16] = acc_out[9];
        end
        6:
        begin
          PRDATA[15:0] = acc_out[10];
          PRDATA[31:16] = acc_out[11];
        end
        7:
        begin
          PRDATA[15:0] = acc_out[12];
          PRDATA[31:16] = acc_out[13];
        end
        8:
        begin
          PRDATA[15:0] = acc_out[14];
          PRDATA[31:16] = acc_out[15];
        end
        default:
        PRDATA = 32'b11111111111111111111111111111111;
      endcase
    end

    assign PREADY  = 1'b1;
    assign PSLVERR = 1'b0;

endmodule
```

Module apb_acc connects to the module top_acc, which acts as a container for the accelerator logic. In the case of the accelerator being extended with other functionality than matrix multiplication, this is where it could be added and selected operation logic could be implemented.

top_acc.sv

```
module top_acc (
  input  logic                clk,
```

```
  output logic [15:0][15:0] acc_out,
  input  logic [15:0][7:0]  acc_in_A,
  input  logic [15:0][7:0]  acc_in_B
  );
  matrix_multiply #(
  )
  matrix_multiply_i (
  .mat_C      (acc_out ),
  .mat_A      (acc_in_A),
  .mat_B      (acc_in_B),
  .clk        (clk     )
  );
endmodule
```

Lastly, the following file contains the matrix multiplication logic.

matrix_multiply.sv

```
module matrix_multiply(
  input  logic                clk ,
  input  logic [15:0][7:0]  mat_A,
  input  logic [15:0][7:0]  mat_B,
  output logic [15:0][15:0] mat_C
);

  always @(posedge clk)
  begin

    mat_C[0] <=
                mat_A[0] * mat_B[0] +
                mat_A[1] * mat_B[4] +
                mat_A[2] * mat_B[8] +
                mat_A[3] * mat_B[12];
    mat_C[1] <=
                mat_A[0] * mat_B[1] +
                mat_A[1] * mat_B[5] +
                mat_A[2] * mat_B[9] +
                mat_A[3] * mat_B[13];
    mat_C[2] <=
                mat_A[0] * mat_B[2] +
                mat_A[1] * mat_B[6] +
                mat_A[2] * mat_B[10] +
                mat_A[3] * mat_B[14];
    mat_C[3] <=
                mat_A[0] * mat_B[3] +
                mat_A[1] * mat_B[7] +
                mat_A[2] * mat_B[11] +
                mat_A[3] * mat_B[15];
```

```
mat_C[4] <=
                mat_A[4] * mat_B[0] +
                mat_A[5] * mat_B[4] +
                mat_A[6] * mat_B[8] +
                mat_A[7] * mat_B[12];
mat_C[5] <=
                mat_A[4] * mat_B[1] +
                mat_A[5] * mat_B[5] +
                mat_A[6] * mat_B[9] +
                mat_A[7] * mat_B[13];
mat_C[6] <=
                mat_A[4] * mat_B[2] +
                mat_A[5] * mat_B[6] +
                mat_A[6] * mat_B[10] +
                mat_A[7] * mat_B[14];
mat_C[7] <=
                mat_A[4] * mat_B[3] +
                mat_A[5] * mat_B[7] +
                mat_A[6] * mat_B[11] +
                mat_A[7] * mat_B[15];
mat_C[8] <=
                mat_A[8] * mat_B[0] +
                mat_A[9] * mat_B[4] +
                mat_A[10] * mat_B[8] +
                mat_A[11] * mat_B[12];
mat_C[9] <=
                mat_A[8] * mat_B[1] +
                mat_A[9] * mat_B[5] +
                mat_A[10] * mat_B[9] +
                mat_A[11] * mat_B[13];
mat_C[10] <=
                mat_A[8] * mat_B[2] +
                mat_A[9] * mat_B[6] +
                mat_A[10] * mat_B[10] +
                mat_A[11] * mat_B[14];
mat_C[11] <=
                mat_A[8] * mat_B[3] +
                mat_A[9] * mat_B[7] +
                mat_A[10] * mat_B[11] +
                mat_A[11] * mat_B[15];
mat_C[12] <=
                mat_A[12] * mat_B[0] +
                mat_A[13] * mat_B[4] +
                mat_A[14] * mat_B[8] +
                mat_A[15] * mat_B[12];
mat_C[13] <=
```

```
                    mat_A[12] * mat_B[1] +
                    mat_A[13] * mat_B[5] +
                    mat_A[14] * mat_B[9] +
                    mat_A[15] * mat_B[13];
    mat_C[14] <=
                    mat_A[12] * mat_B[2] +
                    mat_A[13] * mat_B[6] +
                    mat_A[14] * mat_B[10] +
                    mat_A[15] * mat_B[14];
    mat_C[15] <=
                    mat_A[12] * mat_B[3] +
                    mat_A[13] * mat_B[7] +
                    mat_A[14] * mat_B[11] +
                    mat_A[15] * mat_B[15];
    end
endmodule
```

B.2 C Code

main.c

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "pulpino.h"
#include "uart.h"
#include "timer.h"

// Address of hardware accelerator
#define ACCELERATOR_ADDRESS      0x1A108004
// Address of matrix A data memory space
#define MATRIX_A_ADDRESS         0x100400
// Address of matrix B data memory space
#define MATRIX_B_ADDRESS         0x101400
// Address of result matrix memory space
#define MATRIX_ANSWER_ADDRESS    0x102020

void sendToAccelerator()
{
 unsigned int data;
 unsigned int counter = 0;

 for (unsigned int posA = MATRIX_A_ADDRESS;
      posA < MATRIX_A_ADDRESS + 4*4; posA += 4)
 {
    data = *(volatile unsigned int*) (posA);
     *(volatile unsigned int*) (ACCELERATOR_ADDRESS + counter)
     = data;
     counter += 4;
 }

 for (unsigned int posB = MATRIX_B_ADDRESS;
      posB < MATRIX_B_ADDRESS + 4*4; posB += 4)
 {
     data = *(volatile unsigned int*) (posB);
     *(volatile unsigned int*) (ACCELERATOR_ADDRESS + counter)
     = data;
     counter += 4;
 }
}

void saveResult(int savePos, unsigned int resultValue)
{
  *(volatile unsigned int*) (MATRIX_ANSWER_ADDRESS + savePos)
  = resultValue;
}

void calcWithAccelerator()
{
   // unsigned int data, tmp_1, tmp_2;
   // unsigned int pos = 0;
   unsigned int data, tmp_1, tmp_2, tmp_3;
```

```
  int resultPos=0;
    // Move data to the accelerator
    sendToAccelerator();

    for (unsigned int currInt = ACCELERATOR_ADDRESS;
         currInt < ACCELERATOR_ADDRESS + 4*8; currInt += 4)
    {
    data = *(volatile unsigned int*) (currInt);
    tmp_1 = data & ((unsigned int)pow(2, 16) - 1);
    tmp_2 = (data & ((unsigned int)pow(2, 16) - 1) << 16)
             >> 16;

    // Shifting the bits of tmp_2 16 steps to the left
    // and doing bitwise OR with tmp_1
    // tmp_3 thus contains both the 16-bit numbers
    tmp_3 = tmp_1 | (tmp_2 << 16);
    saveResult(resultPos, tmp_3);
    resultPos += 4;


    }

}

void calcWithSoftware()
{
  int resultPos=0;
  unsigned int tmp;
  unsigned int matrixSol[4][4];

   for (unsigned char y = 0; y < 4; y++)
     {
       for (unsigned char x = 0; x < 4; x++)
      {
          matrixSol[y][x] = 0;
          for (unsigned char k = 0; k < 4; k++)
          {
              matrixSol[y][x] += (unsigned int)
            ((unsigned char) (*(volatile unsigned int*)
                          (MATRIX_A_ADDRESS + 4*y) >> 8*k)
              * (unsigned char) (*(volatile unsigned int*)
                         (MATRIX_B_ADDRESS + 4*k) >> 8*x));
          }
      }
     }

     for (int y = 0; y < 4; y++)
     {
       for (int x = 0; x < 3; x+=2)
        {
          tmp = matrixSol[y][x] | (matrixSol[y][x+1] << 16);
          saveResult(resultPos, tmp);
          resultPos += 4;
        }
     }
}
```

```
long addr;

void printInputMatrices()
{
  __uint8_t data;

    for (int x = 0; x < 16; x++)
    {
      data = *(volatile __uint8_t*)addr;
      printf("%d ", data);
      addr++;
    }
  // Transmit a newline character to flush UART
  printf("\n");
}

void printAnswerMatrix()
{
    __uint16_t data;
    addr = MATRIX_ANSWER_ADDRESS;

    for (int x = 0; x < 16; x++)
    {
      data = *(volatile __uint16_t*)addr;
      printf("%d ", data);
      // We are reading 16-bit numbers. Jump 2 bytes.
      addr += 2;
    }
  // Transmit a newline character to flush UART
  printf("\n");
}

void shortDelay()
{
    for (int i = 0; i < 100000; i++)
  {
      asm volatile("nop");
    }
}

int main()
{
    char command[20];
    char accelerator[4] = "off";
    char fromArm;

    printf("UART INTERFACE READY\n");
    shortDelay();

    while (1)
    {
        fromArm = uart_getchar();
        if (fromArm == '>')
        {
       for (int i=0; i<20; i++)
```

```
        {
        command[i] = uart_getchar();
        // If the last character received through UART
        // is NULL, stop receiving
        if (command[i] == '\0')
            {
      break;
            }
        }
        // when "off" or "on" is sent, the ARM Linux
        // script is trying to restore
        // the correct accelerator status after a restart
        // of this program caused by SPI Loader.
        // Just update the variable.
 if (strcmp(command, "off") == 0)
 {
   strcpy(accelerator, "off");
 }

 else if (strcmp(command, "on") == 0)
{
 strcpy(accelerator, "on");
}

 else if (strcmp(command, "show settings") == 0)
   {
     printf("Current settings:\nAccelerator: %s\n", accelerator);
   }

 else if (strcmp(command, "print matrices") == 0)
 {
   addr = MATRIX_A_ADDRESS;
 printInputMatrices();
 addr = MATRIX_B_ADDRESS;
 printInputMatrices();
 }

 else if (strcmp(command, "print result") == 0)
 {
 //addr = MATRIX_ANSWER_ADDRESS;
 printAnswerMatrix();
  }

 else if (strcmp(command, "accelerator off") == 0)
   {
     strcpy(accelerator, "off");
     uart_send("Accelerator: off\n", 17);
     uart_wait_tx_done();
   }

else if (strcmp(command, "accelerator on") == 0)
   {
   strcpy(accelerator, "on");
   uart_send("Accelerator: on\n", 16);
   uart_wait_tx_done();
   }
```

```
      else if (strcmp(command, "calculate") == 0)
        {
          if (strcmp(accelerator, "on") == 0)
      {
          reset_timer();
        start_timer();
          calcWithAccelerator();
          stop_timer();
          printAnswerMatrix();
        }

          else if (strcmp(accelerator, "off") == 0)
      {
        reset_timer();
        start_timer();
        calcWithSoftware();
        stop_timer();
        printAnswerMatrix();

      }
    }

    else if (strcmp(command, "get time") == 0)
    {
        // Request for timer count. Send via UART
      printf("%d\n", get_time());
    }

    else
        {
        uart_send("Invalid command! Type help for a
                    full list.\n", 44);
         uart_wait_tx_done();
        }
    }
}
```

B.3 Shell Scripts

start.sh

```
#/bin/ash

#  @Description: Runs on ARM Linux and transfers the PULPino
#  main program to the PULPino instruction memory, initiates UART
#  communication between ARM Linux and the PULPino and enables
#  control of the system through a series of commands.

# Transfers the PULPino program and properly intializes UART
# communication between the ARM Linux UART interface (/dev/ttyPS0)
# and that of the PULPino microcontroller.
./spiload -t1 main_program.txt

# Delete any pre-existing output.txt, which is where UART messages
# from the PULPino are stored for processing by this script.
rm output.txt

# Redirect incoming UART messages to output.txt.
cat /dev/ttyPS0 >> output.txt &

clear

# Whenever SPI Loader is used, in our case by the scripts
# load_matrix.sh & start.sh (this script), it finishes by
# resetting the PULPino. This means the variable used by
# the PULPino program to remember whether the accelerator
# is to be used or not is reset. By remembering it here,
# we can inform the PULPino of the correct state after a
# reset.
ACCEL_STATUS="off"

# Welcome message upon startup.
echo -e "\nWelcome to the DATX02-19-30 PULPino System!"
echo -e "\nType "help" for a full list of commands."
echo -e "\nEnter command:"

# Send command over UART.
send()
{
    # Erase output.txt in preparation for new UART transfer
    > output.txt

    # The '>' and null characters initiate command parsing and
    # indicate end of transfer respectively in the PULPino program.
    echo -e ">$1\0" > /dev/ttyPS0

    # Give UART-transfer enough time
    sleep 1
}

print_reply()
{
    echo
```

```
    # Print the received data
    cat output.txt
    echo
    # Erase output.txt in preparation for new UART transfer
    > output.txt
}

manual_input()
{
  # Make sure the text files containing user entry are empty
  > ./MATRICES/temp1.txt
  > ./MATRICES/temp2.txt
  for i in `seq 1 $1`
  do
    echo -e "Enter values between 0-63 of 4x4 matrix $i. \\
             'q' to cancel:\n"
    for n in `seq 1 16`
      do
        while :
        do
          read -p "$n: " value

          # The user can cancel input at any time
          if [ "$value" = "q" ]; then
            echo -e "\nManual input cancelled.\n"
            # If cancelled, break out and back into main loop
            break 4
          elif [ "$value" -ge 0 -a "$value" -le 63 ]; then
            # Check if the value is within 0-63 (6-bit)
            echo $value >> ./MATRICES/temp$i.txt
            break
          else
            echo -e "\nInvalid input. Enter a value between 0-63:\n"
          fi
        done
      done
      xargs -n4 < ./MATRICES/temp$i.txt > ./MATRICES/manual$i.txt
      rm ./MATRICES/temp$i.txt  # Cleanup
      done
}

calculate()
{
  send "calculate"
  # Sleep for 1 second to allow PULPino enough time to transfer answer.
  sleep 1

  # output.txt contains the input matrix cells, sent from the PULPino.
  # xargs reformats these into 4 column format and saves in output2.txt
  xargs -n4 < output.txt > output2.txt

  # Replace all the whitespaces with tabs for readability when printing
  # the matrices.
  sed -in "s/\s/\t/g" output2.txt

  echo -e "\n        [  Answer  ]\n-------------------------"
```

```
  # Print line 1-4 (answer matrix)
  sed -n 1,4p output2.txt
  echo

  # Get the cycle count from the PULPinos timer register (Timer A)
  send "get time"

  # Save it in the cycle_count variable
  cycle_count=$( cat output.txt )
  echo -e "Calculation cycle count: $cycle_count\n"

  # Delete the temporary files
  rm output2.txt output2.txtn
}

print_matrices()
{
  send "print matrices"
  # Sleep for 1 second to allow PULPino enough time to transfer
  # the contents of the input matrix data in its data memory.
  sleep 1

  # output.txt contains the input matrix cells, sent from the PULPino.
  # xargs reformats these into 4 column format and saves in output2.txt
  xargs -n4 < output.txt > output2.txt

  # Replace all the whitespaces with tabs for readability when printing
  # the matrices.
  sed -in "s/\s/\t/g" output2.txt
  echo -e "\n        [ Matrix 1 ]\n-------------------------------"

  # Print line 1-4 (matrix 1)
  sed -n 1,4p output2.txt
  echo -e "\n        [ Matrix 2 ]\n-------------------------------"

  # Print line 5-8 (matrix 2)
  sed -n 5,8p output2.txt
  echo

  # Delete the temporary files
  rm output2.txt output2.txtn
}

#-------------------------------[ MAIN LOOP ]-------------------------------
while :
do
  read -p '> ' command

  case $command in

    "help")
      echo -e "\nAvailable commands:"
      echo      "--------------------"
      echo "help              -- this menu"
      echo "show settings     -- print the current system settings"
```

```
  echo "enter matrices    -- manually enter one or two 4x4     \\
                            matrices"
  echo "load matrices     -- load pairs of random matrices into \\
                            PULPino data memory."
  echo "print matrices    -- print the input matrices currently \\
                            in the PULPinos data memory"
  echo "print result      -- print the latest calculated result \\
                            from the PULPinos data memory"
  echo "accelerator off   -- disable use of the hardware       \\
                            accelerator"
  echo "accelerator on    -- enable use of the hardware        \\
                            accelerator"
  echo "calculate         -- begin calculation with the current \\
                            settings"
  echo "demo              -- Perform matrix multiplication on  \\
                            two randomized 4x4 matrices with"
  echo "                  -- and without the accelerator and   \\
                            compare time taken."
  echo "                  -- with the current accelerator settings"
  echo "quit              -- quit the script"
  echo
  ;;

"show settings")
  send "$command"
  print_reply
  ;;

"accelerator on")
  ACCEL_STATUS="on"
  send "$command"
  print_reply
  ;;

"accelerator off")
  ACCEL_STATUS="off"
  send "$command"
  print_reply
  ;;

"enter matrices")
  echo -e "\nWould you like to input one or two matrices? \\
           (\"one\"/\"two\"):\n"
  while :
  do
    read -p '> ' command
    if [ "$command" = "one" ]; then
      echo
      manual_input "1"
      echo -e "\n        [ Matrix 1 ]\n-------------------------"
      sed "s/\s/\t/g" ./MATRICES/manual1.txt
      echo -e "\nWhich matrix memory position do you want to \\
               upload it to (1,2) ? \n"
      while :
      do
        read -p '> ' position
```

```
        if [ "$position" = "1" ] || [ "$position" = "2" ]; then
          echo -e "\nUploading the matrix to PULPino matrix \\
                   memory position $position.\n"
          ./MATRICES/load_matrix.sh ./MATRICES/manual1.txt \\
                                    $position 1> /dev/null
            # PULPino program restarted. Restore accelerator
            # setting.
            echo -e ">$ACCEL_STATUS\0" > /dev/ttyPS0
          break
        else
          echo -e "\nIncorrect position. Enter 1 or 2:\n"
        fi
      done
      rm ./MATRICES/manual1.txt              # Cleanup
      break
    elif [ "$command" = "two" ]; then
       echo
      manual_input "2"
      echo -e "\n        [ Matrix 1 ]\n-------------------------------"
      sed "s/\s/\t/g" ./MATRICES/manual1.txt
      echo -e "\n        [ Matrix 2 ]\n-------------------------------"
      sed "s/\s/\t/g" ./MATRICES/manual2.txt
      echo -e "\nUploading the matrices to PULPino data memory.\n"
      ./MATRICES/load_matrix.sh ./MATRICES/manual1.txt \\
                                ./MATRICES/manual2.txt 1> /dev/null
        # PULPino program restarted. Restore accelerator setting.
        echo -e ">$ACCEL_STATUS\0" > /dev/ttyPS0
      rm ./MATRICES/manual1.txt ./MATRICES/manual2.txt # Cleanup
      break
    else
      echo -e "\nInvalid entry. Enter \"one\" or \"two\"\n"
    fi
  done


  ;;

"load matrices")
  for i in `seq 1 16`; do echo $(( ( RANDOM %63 ) )); done \\
                       | xargs -n4 >> ./MATRICES/random1.txt
  for i in `seq 1 16`; do echo $(( ( RANDOM %63 ) )); done \\
                       | xargs -n4 >> ./MATRICES/random2.txt
  ./MATRICES/load_matrix.sh ./MATRICES/random1.txt \\
                            ./MATRICES/random2.txt 1> /dev/null
    # PULPino program restarted. Restore accelerator setting.
    echo -e ">$ACCEL_STATUS\0" > /dev/ttyPS0
  echo -e "\nRandom matrix pair was uploaded to data memory."
  echo -e "\n        [ Matrix 1 ]\n-------------------------------"
  sed "s/\s/\t/g" ./MATRICES/random1.txt
  echo -e "\n        [ Matrix 2 ]\n-------------------------------"
  sed "s/\s/\t/g" ./MATRICES/random2.txt
  echo
  rm ./MATRICES/random1.txt ./MATRICES/random2.txt      # Cleanup
  ;;

"print matrices")
```

```
    print_matrices
    ;;

"print result")

    send "$command"
    # Sleep for 1 second to allow PULPino enough time to transfer
    sleep 1

    # the contents of the input matrix data in its data memory.
    # output.txt contains the input matrix cells, sent from the
    # PULPino. xargs reformats these into 4 column format and
    # saves in output2.txt
    xargs -n4 < output.txt > output2.txt

    # Replace all the whitespaces with tabs for readability when
    # printing the matrices.
    sed -in "s/\s/\t/g" output2.txt
    echo -e "\n        [  Result  ]\n-------------------------"
    cat output2.txt         # Print the matrix multiplication result
    echo
    rm output2.txt output2.txtn        # Delete the temporary files
    ;;

"calculate")
    calculate
    ;;

"demo")
    echo -e "\n------------------------------------------------------------"
    echo -e "DEMO: Performing matrix multiplication of two random \\
          4x4 matrices with"
    echo -e "      and without the hardware accelerator, then     \\
          printing a summary."
    echo -e "------------------------------------------------------------\n"

    send "accelerator off"

    for i in `seq 1 16`; do echo $(( ( RANDOM %63 ) )); done \\
                          | xargs -n4 >> ./MATRICES/random1.txt
    for i in `seq 1 16`; do echo $(( ( RANDOM %63 ) )); done \\
                          | xargs -n4 >> ./MATRICES/random2.txt

    ./MATRICES/load_matrix.sh ./MATRICES/random1.txt \\
                              ./MATRICES/random2.txt 1> /dev/null
    # PULPino program restarted. Restore accelerator setting.
    echo -e ">$ACCEL_STATUS\0" > /dev/ttyPS0
    print_matrices
    echo -e "\n----------[ Calculation done in software ]----------"
    calculate
    cycle_count2=$cycle_count
    send "accelerator on"
    echo -e "\n---[ Calculation done with hardware accelerator ]---"
    calculate
    echo -e "The accelerator was $( awk -v x=$cycle_count  \\
            -v y=$cycle_count2 'BEGIN { print y / x }' )  \\
```

```
                times faster!\n"
        rm ./MATRICES/random*.txt           # Cleanup

        ;;

    "quit")
        echo -e "\nExiting. Good Bye!\n"
        # Kill cat processes now that we are no longer listening
        killall cat
        rm output.txt
        exit 0
        ;;

    *) echo -e "\nInvalid command! Enter \"help\" to list all \\
                available commands.\n";;
  esac
done
```

load_matrix.sh

```
#!/bin/ash

# @Description: Writes matrix data to PULPino data memory.
# @Version: 2 - Rewritten for compatibility with ash (Almquist shell)
# that is available on the ARM Linux system.

MATRIX1_ADDRESS=0x100400
MATRIX2_ADDRESS=0x101400

if [ "$1" = "-h" ] || [ "$1" = "--help" ]; then
  echo "\n[ load_matrix.sh - Transfers one or two matrices      \\
        contained in separate text files to the correct         \\
        adresses in the PULPino memory. ]\n"
  echo "\nOnly matrices with element counts that are multiples \\
        of 4 are supported!\n"
  echo "\nUsage: ./load_matrix.sh [FILE1] [FILE2]"
  echo "\n\texample: ./load_matrix.sh matrix1.txt \\
  matrix2.txt\t\t\t --Uploads matrix data contained \\
  in the files to the PULPino data memory."
  echo "\n\texample: ./load_matrix.sh matrix.txt 2\t\t\t\t \\
  --Uploads the matrix data in matrix.txt to the memory address
  \\reserved for matrix 2."
  echo
  exit 0
fi

#-----------------------[ FUNCTION: "prepare_data" ]-------------------
# Description:  Reads text file containing matrix data and generates
# a stim file comprised of memory addresses and the data to be
# written that "SPI Loader" can read.
#
# Parameters: Matrix_filename and {1,2} indicating which of the
# reserved matrix memory positions to write to.
#
# Example: prepare_data matrix1.txt 2    read the matrix data in
# matrix1.txt and generate a stim-file with memory addresses starting
# from the first position in memory reserved for matrix 2.
#-----------------------------------------------------------------------

prepare_data()
{
  #MATRIX_SIZE=$(wc $1 | head -n1 | cut -d " " -f3)
  TRIM_LEADING_WHITESPACE=$(wc $1 | sed 's/^ *//g')
  MATRIX_SIZE=$(echo $TRIM_LEADING_WHITESPACE | cut -d " " -f1)
  ELEMENT_COUNT=$(echo $TRIM_LEADING_WHITESPACE | cut -d " " -f2)
  STIM_FILENAME="$(echo $1 | cut -d "." -f1)_stim.txt"
  if [ "$2" = "1" ]; then
    CURRENT_ADDRESS=$MATRIX1_ADDRESS
  else
    CURRENT_ADDRESS=$MATRIX2_ADDRESS
  fi

  echo -e "Matrix file \"$1\" contains a $MATRIX_SIZE x \\
    $MATRIX_SIZE matrix.\n$ELEMENT_COUNT elements and their \\
```

```
    memory addresses are added to a stim file \\
    /home/ftp/MATRICES/$STIM_FILENAME\"."
  > /home/ftp/MATRICES/$STIM_FILENAME

  # tempdata.txt will contain the matrix elements in 4 columns.
  xargs -n4 < $1 > tempdata.txt
  # tempdata2.txt will contain the above values in hexadecimal and in
  # a correct format: a 0 is added to 1-digit numbers, i.e. "a" -> "0a"
  > tempdata2.txt

  # tempaddress.txt will contain the memory addresses to write to.
  # This file will be merged with the above
  # tempdata.txt to yield a working stim file.
  > tempaddress.txt

  # For each line in tempdata.txt, convert to hex and invert order.
  while read -r n1 n2 n3 n4
  do
    if [ $n4 -lt 16 ]; then
      printf '0%x' "$n4" >> tempdata2.txt
    else
      printf '%x' "$n4" >> tempdata2.txt
    fi
    if [ $n3 -lt 16 ]; then
      printf '0%x' "$n3" >> tempdata2.txt
    else
      printf '%x' "$n3" >> tempdata2.txt
    fi
    if [ $n2 -lt 16 ]; then
      printf '0%x' "$n2" >> tempdata2.txt
    else
      printf '%x' "$n2" >> tempdata2.txt
    fi
    if [ $n1 -lt 16 ]; then
      printf '0%x\n' "$n1" >> tempdata2.txt
    else
      printf '%x\n' "$n1" >> tempdata2.txt
    fi
  done < tempdata.txt

  # Each line in the stim file will contain 4 elements.
  # Divide element count by 4 to know how many address
  # entries are needed.
  ADDRESS_ROW_COUNT=$(($ELEMENT_COUNT / 4))

  for i in `seq 1 $ADDRESS_ROW_COUNT`
  do
    printf '00%x_\n' "$CURRENT_ADDRESS" >>  tempaddress.txt
    CURRENT_ADDRESS=$(($CURRENT_ADDRESS + 0x4))
  done
  merge
  rm tempaddress.txt tempdata.txt tempdata2.txt          # Cleanup
}

#--------------------[ END "prepare_data" ]----------------------
```

```
upload()
{
  echo "Using SPI loader to transfer the spim data \\
    into PULPino memory."
  /home/ftp/spiload $1
}

# Merges two text files line by line.
# Instead of "paste" which is unavailable
# in our lightweight ARM Linux distribution
merge()
{
  > /home/ftp/MATRICES/temp.txt  # Remove any existing temp.txt
  for i in `seq 1 $MATRIX_SIZE`
  do
    sed -n $((i))p tempaddress.txt >> /home/ftp/MATRICES/temp.txt
    sed -n $((i))p tempdata2.txt >> /home/ftp/MATRICES/temp.txt
  done

  xargs -n2 < /home/ftp/MATRICES/temp.txt \\
    > /home/ftp/MATRICES/$STIM_FILENAME
  # Remove all the whitespaces to make it a valid stim-file.
  sed -in "s/\s//g" /home/ftp/MATRICES/$STIM_FILENAME
}

#-------------------------[ MAIN PROGRAM ]---------------------------

# Second argument always first filename. Check if it exists.
if [ ! -e $1 ]; then
  echo -e "File $1 does not exist."
  exit 1
fi

if [ "$2" = "1" ] || [ "$2" = "2" ]; then
    if [ "$2" = "1" ]; then
    START_ADDRESS=$MATRIX1_ADDRESS
  else
    START_ADDRESS=$MATRIX2_ADDRESS
  fi
  echo "One matrix file and position provided. Converting for start \\
  address $START_ADDRESS and uploading it."

  # A matrix is to be loaded into matrix position 1 or 2. Send matrix
  # file and memory location to "prepare_data" as parameters.
  prepare_data $1 $2

  upload "/home/ftp/MATRICES/$STIM_FILENAME"

# If the second filename is invalid, output error message and exit.
elif [ ! -e $2 ]; then
    echo -e "File $2 does not exist."
    exit 1
else
    echo "Two matrix files provided. Converting and uploading them:"
    echo "-------------------------------------------------------------"
```

```
    prepare_data $1 "1"
    cp -f /home/ftp/MATRICES/$STIM_FILENAME \\
      /home/ftp/MATRICES/combined_stim.txt
    echo ""
    prepare_data $2 "2"
    cat /home/ftp/MATRICES/$STIM_FILENAME >> \\
      /home/ftp/MATRICES/combined_stim.txt
    echo ""
    echo "The matrices in \"$1\" and \"$2\"  \\
      were converted and added to /home/ftp/MATRICES/combined_stim.txt"
    echo ""
    upload "/home/ftp/MATRICES/combined_stim.txt"
fi

rm temp.txt 2> /dev/null                    # Cleanup
#-------------------------------[ END MAIN ]--------------------------------
```