Implementation of Hardware Accelerators on Zynq

Toft, Jakob Kenn; Nannarelli, Alberto

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Technical report - Implementation of Hardware Accelerators on Zynq

Jakob Toft

September 14, 2016

Supervisor: Alberto Nannarelli

Abstract

In the recent years it has become obvious that the performance of general purpose processors are having trouble meeting the requirements of high performance computing applications of today. This is partly due to the relatively high power consumption, compared to the performance, of general purpose processors, which has made hardware accelerators an essential part of several datacentres and the worlds fastest super-computers.

In this work, two different hardware accelerators were implemented on a Xilinx Zynq SoC platform mounted on the ZedBoard platform. The two accelerators are based on two different benchmarks, a Monte Carlo simulation of European stock options and a Telco telephone billing application. Each of the accelerators test different aspects of the Zynq platform in terms of floating-point and binary coded decimal processing speed. The two accelerators are compared with the performance of the ARM Cortex-9 processor featured on the Zynq SoC, with regard to execution time, power dissipation and energy consumption.

The implementation of the hardware accelerators were successful. Use of the Monte Carlo processor resulted in a significant increase in performance. The Telco hardware accelerator showed a very high increase in performance over the ARM-processor.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays High Performance Computing (HPC) is being used for a wide range of applications, ranging from weather forecasts, scientific simulations etc. to financial calculations. In financial applications there IS often a need for the calculations to be finished fast and/or precise, having the results of a simulation of a stock before competitors can give a firm an advantage on the stock market, and in the banking sector precision is important. For simulations to be processed fast they are often done in large data-centres, which in return have a large power consumption. The power consumption is a significant problem and an important part of the design considerations of data-centres, as even the local price on electricity may decide whether or not a data-centre can be a profitable operation. The goal is thus both a high performance and a low power-consumption. Achieving this can be hard, if not impossible, if only general-purpose processors are used [2].

Achieving a high performance and maintaining a low power consumption can in many cases be done by using a system utilizing hardware accelerators, as the primary computing load can be transferred to faster and more power efficient specialized processing cores. Several different hardware accelerator technologies are already in use and available on the commercial market, the three most common types are Graphical Processing Units (GPUs), Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs), each technology has its own pros and cons. The advantage of ASICs and FPGAs is that they can implement fine-tuned algorithms on a hardware level, and even implement the algorithms in parallel if the applications allows it, which in return makes it possible for them to obtain an even higher performance [1]. An advantage of FPGAs is that they have a relatively low cost compared to ASICs when operating on a small scale, and are very flexible compared to GPUs.

Most hardware accelerators are connected to Central Processing Units (CPUs) through general bus' such as the PCIe-bus. This connection can be restrictive on the performance of hardware accelerators that process large amounts of data, due to the bus between accelerator and CPU being unable to deliver data at high enough rates. Other methods to connect CPU and accelerator exists, such as special bus' and placing the CPU and accelerator on the same silicon die, one thing these share is an increased cost and a reduced customizability of the accelerator. The fastest connection between a hardware accelerator and CPU can be achieved when both are placed on the same silicon die, minimizing the distance between the two, making it easier to maximize the bandwidth between the two. This implementation method has until now made the hardware of the final system non-reconfigurable, the rather new Xilinx Zynq SoC does however overcome this problem, as it features a dual-core ARM CPU and programmable logic on the same silicon die [8], giving the pros of using an FPGA as hardware accelerator while achieving a better bus-connection between CPU and FPGA.

In this report the implementation of two different hardware accelerators on the Xilinx Zynq SoC platform is documented. The hardware accelerators are two different ASPs each designed for a different benchmark, each benchmark have different requirements with regard to data throughput and type of calculations. The two benchmarks used is a Monte Carlo simulation of european stock option prices and a Telco benchmark simulating a telephone billing application. The hardware accelerators is implemented on the programmable logic of the

Xilinx Zynq SoC and controlled from an application running on the ARM processor. Finally the execution time and power consumption of the two hardware accelerators is presented and compared to a software execution of the same test-cases. The software execution is performed on the ARM processor of the Xilinx Zynq SoC. The results demonstrates that the hardware accelerators achieve a much faster execution time than the software execution and that a reduction in energy consumption is obtained.

2 Platform

For this project the Zedboard platform [10] was used. The main features of this board, with regard to this project, is the Xilinx Zynq-7000 SoC xc7z020 and 512MB of DDR3 memory mounted on the board, as well as a current-sense resistor on the power supply.

2.1 Xilinx Zynq

Zynq SoC is a product line from Xilinx where a processor and programmable logic is implemented on the same silicon die. More specifically it is an ARM Cortex-9 dual-core processor connected to various sized amounts of programmable logic dependent on the chip model. In addition to the ARM Dual-core processor there is also an ARM NEON unit, which is a general-purpose Single-Instruction-Multiple-Data(SIMD) engine.

The combination of a general-purpose processor and programmable logic on the same silicon die allows realization of hardware accelerators with at higher data-throughput, as the communication bus between the two units is as short as possible.

The three most common types of hardware accelerator architectures are depicted in Figure 1. Each of these have their own pros and cons, with type 'C' being the fastest architecture due to the communication bus between CPU and accelerator being physically short. Reducing the length of a bus reduces the capacitance of the bus, which in return makes it easier to obtain a higher data throughput on the bus. A normal computer would allow implementation of the type 'a' architecture through the PCIe-bus while the Zynq SoC makes a type 'c' architecture implementation possible.

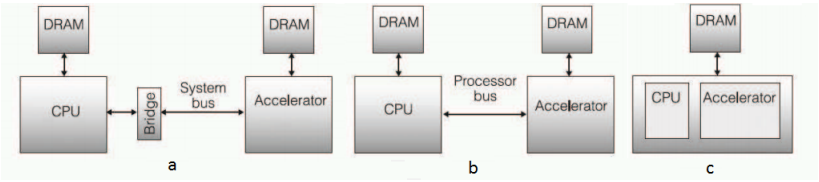

Figure 1: The three general hardware accelerator structures (source: [6])

The Zynq SoC system diagram is depicted in Figure 2. As it can be observed the ARM Cortex-9 processor is connected to the programmable logic of the Zynq SoC through AXI-interconnects and Extended Multi-use I/O (EMIO). EMIO will not be used in this project, but is useful if one want to use some of the built-in communication modules such as SPI, I2C etc. to communicate with external hardware or hardware implemented in the programmable logic. AXI-interfaces are described in section 2.2.

2.2 AXI interfaces

The Xilinx Vivado software package from Xilinx is the recommended development environment when working with the Zynq SoCs. The Vivado environment is highly geared towards using Intellectual Property (IP) and block-based design

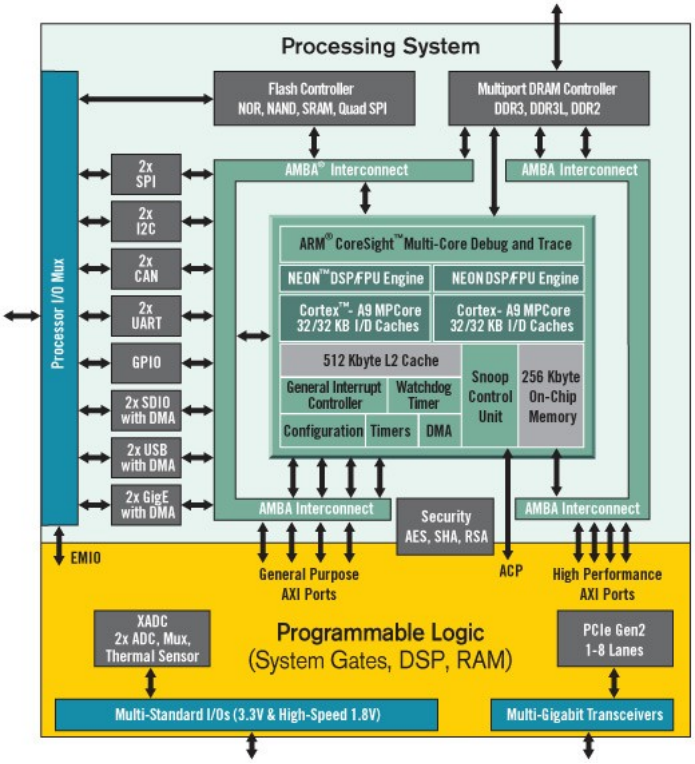

Figure 2: System diagram of the Zynq SoC with the dual-core ARM processor and the programmable logic (source: [8])

to increase productivity and lower development time. Vivado features several necessary IP blocks and templates for Zynq- and FPGA-designs, several of these blocks are related to or used for implementation of Xilinx's AXI-bus and configuration of the ARM processor.

Adding custom RTL to a project is done by creating IP-blocks from templates and using these in the block design of the project. As mentioned in section 2.1 the Zynq SoC is internally using Xilinx's AXI-bus to connect the ARM processor with the programmable logic of the Zynq SoC, and as such it is preferable to use the AXI-bus to connect the custom RTL to the processor of the system using the AXI-bus. Other connection possibilities exists though, these are however not as fast as the AXI-bus and it is thus up to the developer to choose a suitable bus for communication between the RTL and the ARM processor.

When creating a new IP peripheral in the Vivado design suite, the user have the choice of adding AXI-connections of three types, Full, Lite and Stream, these can be added either as a slave or master connections. When creating the peripheral a template is given to the user with the basic control logic for the added AXI-bus'. The three different AXI-bus types available to the peripheral is shown in Figure 3.

In this project the AXI-lite and AXI-stream bus' were used. The periph-

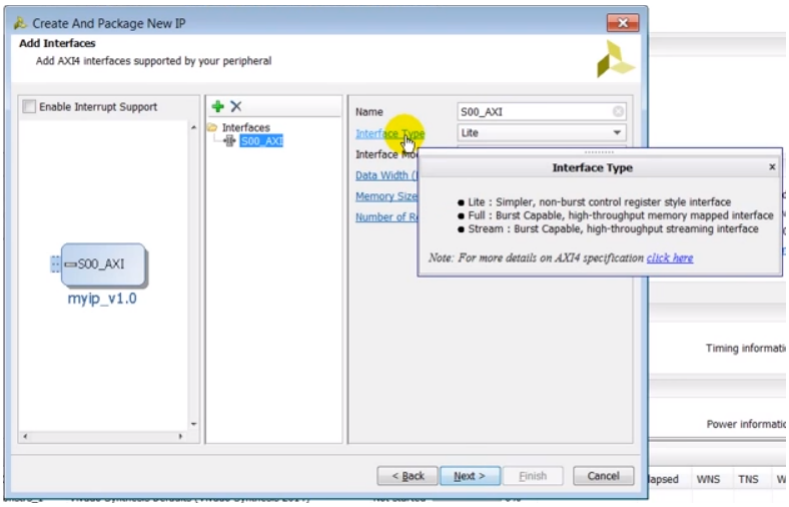

Figure 3: Types of AXI-interfaces available when creating a new peripheral in Xilinx Vivado

eral template provided with the AXI-lite slave interface is a simple register style interface, where the user upon creation of the peripheral can specify how many registers should be available, limited to the range of 8-512. In a slave-configuration of the AXI-lite interface, the registers in the template can be written to and read from by both the master and slave, but reset and the clock is provided by the master.

Only the connections of the template for the AXI-stream interface were used in this project, the control logic was not necessary in the implementation, but was provided upon generation of the peripheral with the AXI-stream interface.

2.3 ZedBoard power measurements

For power measurements a shunt-resistor, mounted on the power supply of the ZedBoard, will be used. The shunt-resistor is a $10m\Omega$ resistor in series with the input supply line to the board, this resistor can be used to measure the current draw of the whole board. The current draw can be used to calculate the power consumption of the board using Ohm's law and the supply voltage. As the power supply voltage is 12V, the power consumption of the board can be obtained by measuring the voltage over the current sense resistor, and inserting the measured value into equation 1.

$$P = \frac{V_{measured}}{10m\Omega} \cdot 12V \tag{1}$$

The shunt-resistor is connected in parallel with header 'J21' allowing easy measurement of the voltage over the resistor. The location of 'J21' is shown in Figure 4.

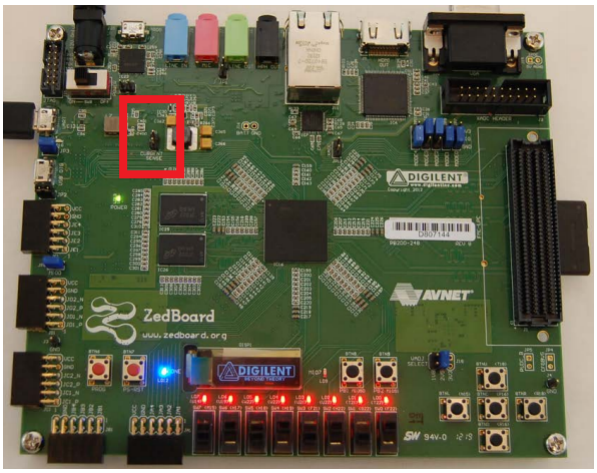

Figure 4: Location of 'J21' on the Zedboard

3 Implementation

In this project two different hardware accelerators based on previous work [5] were implemented, a Telco processor and a Monte Carlo processor. The main difference between these two accelerators is that the Telco processor requires a stream of data while the Monte Carlo processor simply need some initial values and a go-signal. This requires two widely different implementations on the Zynq platform, which will be shown in the following subsections. Furthermore the configuration of the ARM processor is also explained, as the ARM-processor was used for both the hardware and software execution of each benchmark.

Implementation of both processors was done in the Xilinx Vivado development environment [7], as this is the intended development environment for the Zynq SoC. The Vivado software package is shipped with several Intellectual Property packages for development on different FPGAs, but most importantly several IPs related to AXI-bus communication. In the implementation of both hardware accelerators, AXI-related IP was used for connecting the CPU and accelerator, as well as connecting the Telco processor with the external RAM.

The implementation of each of the hardware accelerators consists of a hardware implementation of the specific ASP and an embedded software application running on the ARM processor controlling the accelerator.

3.1 Monte Carlo processor

The Monte Carlo processor is in terms of interface with the ARM CPU the more simple of the two ASPs to implement. This is due to the Monte Carlo processor only requiring an initial set of parameters and a go-signal to be run, and thus the Monte Carlo processor can be controlled though use of a relatively small set of registers.

In the following subsections the hardware implementation and software application development for the Monte Carlo accelerator will be described.

3.1.1 Hardware implementation

Implementation of the Monte Carlo processor was split up into three tasks; packaging the Monte Carlo processor into an IP package with an AXI-interface, configuration of the ARM processor system, and connection of the Monte Carlo IP and the processor system.

Creating the Monte Carlo IP was done in the Vivado design suite, where Vivado was used to generate a template for an AXI4-peripheral with an AXI-lite interface operating in slave-mode. The AXI-lite interface was configured to have 16 32-bit registers, this amount of registers is sufficient for control of the Monte Carlo processor. After this the top-level module of the Monte Carlo ASP was implemented into the peripheral, where the input clock to the ASP was connected to the global clock of the programmable logic, the reset of the ASP was connected to a register in the peripheral. The reset was connected to a register instead of the global reset, such that the ASP can be reset without resetting the entire programmable logic fabric on the Zynq SoC. The seeds for the random generator in the Monte Carlo ASP was connected to a range of registers. Then the output and status signals of the Monte Carlo ASP was connected to a different set of registers, for the processor to read from, and the

logic controlling the registers was modified to update correctly when status and result signals changed. Of the 16 32-bit registers, 9 was used, the allocation of these are depicted in table 1.

Register	Application
0	Control signals
1	Result out from ASP
2	Status signals from ASP
3	HI 32-bit of no. iterations
4	LO 32-bit of no. iterations
5	Seed 1 for random number generator
6	Seed 2 for random number generator
7	Seed 3 for random number generator
8	Seed 4 for random number generator

Table 1: Allocation of registers in the Monte Carlo IP

As it can be observed from table 1, the constants used for the simulation is not included, only the seeds for the random generator is, there is however plenty of registers available to re-wire the constants into registers.

Finally the Monte Carlo peripheral was synthesized and packaged into an IP entity.

Configuration of the ARM processor system was done by creating a new project in Vivado and adding an instance of the processor system IP to the system design. The processor IP was then configured to have only the necessary modules to connect with the generated Monte Carlo IP. This means that the processor system was configured to have a single AXI-bus connection to the programmable logic, and to provide a clock running at 50MHz (this was the highest frequency achievable without timing errors) to the programmable logic.

After this the Monte Carlo IP was added to the design, and connected to the processor system and some external pins on the Zynq SoC. An automatic connection wizard of Vivado was run, this added a couple of AXI related IP blocks to the system, blocks which synchronize the reset signals and connect the IP AXI bus the AXI bus of the processor system. The external pins used by the Monte Carlo IP are connected to the LEDs of the Zedboard and used as status indicators. The system overview as seen in Vivado is depicted in Figure 5.

3.1.2 Software implementation

The software application developed for controlling the Monte Carlo processor was based on work from [3], which shows how to write to and read from registers in an AXI4-lite peripheral. The pseudo-code of the final application is shown in Figure 6.

As it can be observed from the pesudo-code, the software application is polling the status register of the Monte Carlo processor to determine whether the calculations are done or not. Not shown in the pseudo-code is the functions responsible for measuring execution time of the application. The time-taking is however started just before the go-signal is given and stopped right after the done-signal is received.

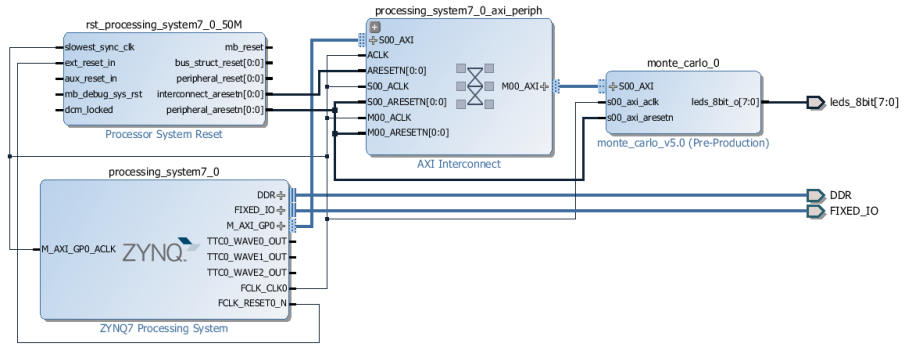

Figure 5: Top-level system overview of the Monte Carlo implementation on Zynq as seen in Vivado

```
init_platform();
reset_hardware_accelerator();
write_values_to_control_registers();
send_go_signal();
while(!done){
        done = register_2;
}
print(result);
cleanup_platform();
return 0;
```

Figure 6: Pseudo-code for running the Monte Carlo accelerator

3.2 Telco processor

As the Telco Processor in other works [5] is constrained by the data transfer rate, the implementation of the Telco processor is focused on implementing Direct Memory Access (DMA) in order to obtain a high data-transfer rate.

In the following subsections the hardware implementation and software application development will be described.

3.2.1 Hardware implementation

The hardware implementation of the Telco processor was done in roughly four stages; first the processor system was defined, afterwards DMA-transfer capability was added, then the Telco processor was implemented into an IP peripheral, finally the Telco IP was integrated into the system design.

First off a new project was created in Vivado, and a processor system IP was added and configured. The processor system was configured to have a High

Performance AXI slave and a normal AXI master connection, and a 100MHz clock for the programmable logic (the clock frequency was later reduced due to unmet timing constraints).

When the processor had been defined, DMA capability was implemented as described in [4], where data in the external memory simply, using DMA, is looped back to the memory through a FIFO. A system diagram is shown in Figure 7 and the Vivado design is shown in Figure 8.

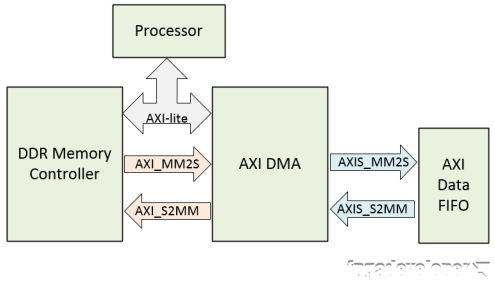

Figure 7: System diagram of data loop-back on Zynq using DMA, source [4]

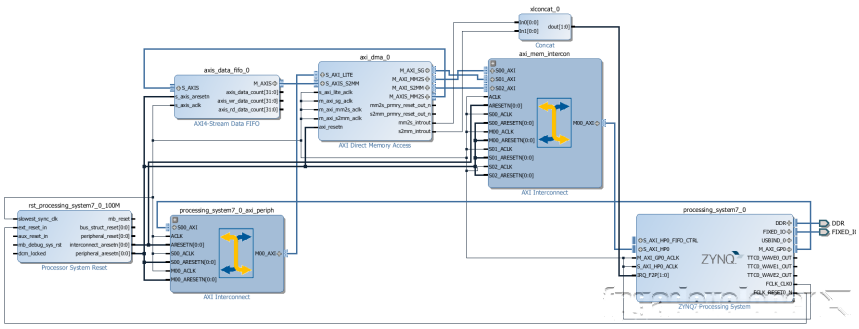

Figure 8: Vivado design of data loop-back on Zynq using DMA, source [4]

The Telco processor was implemented as a peripheral IP like the Monte Carlo processor but with a different set of input and outputs. An AXI4 peripheral was generated, but instead of a single AXI-lite interface, two AXI-stream interfaces was added instead, one slave and one master. A top-level HDL file of the peripheral was generated by Vivado along with the control logic of the two interfaces. In this implementation the control logic was however bypassed, and the Telco processor was implemented directly between the stream data input and data output of the peripheral.

This implementation scheme circumvent the need to edit and/or create control logic in the IP and lets the AXI IP on either end of the Telco IP block control the transfer.

The AXI control signals were wired directly from input to output of the peripheral. The *clock* and *reset* signal to the Telco processor was wired to the *clock* and *reset* signal of the AXI slave interface. Finally the clock *enable* signal to the Telco processor was wired to a logic 'AND' of the *valid* signal from the

AXI master interface and the *ready* signal from the AXI slave interface. This configuration is depicted in Figure 9.

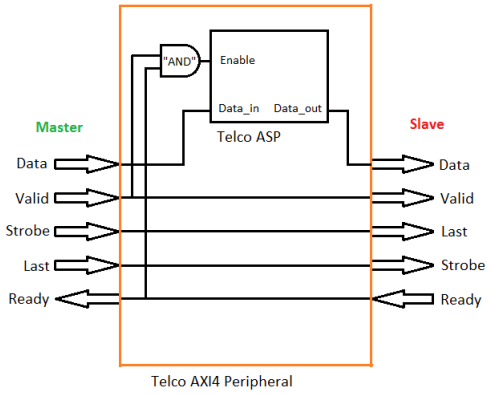

Figure 9: Implementation of Telco processor in an AXI4-peripheral IP

The logical "AND" of the *valid* and *ready* signal is used due to how the AXI-stream bus works. It is not enough to use just a single one of the signals as the data on the *data* line only gets updated when both signals are valid. If one were to use the *valid* signal only, one might process the same data twice, this will introduce an error in the final result of the Telco processor as there is an accumulator in the last stages of the pipeline. The waveforms of AXI-stream signals are illustrated in Figure 10, where it can be observed how data on the *TDATA* line only change on rising edge of the clock when both *valid* and *ready* are asserted.

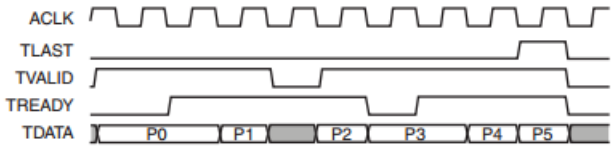

Figure 10: Waveforms of AXI-stream signals during a transfer, source [9]

Because the Telco ASP is pipelined, the data being processed will be delayed an amount of transfers equal to the depth of the pipeline. The pipeline depth of the Telco ASP is 11, which results in that the last 11 data instances of a data set never would be fully processed nor transmitted back to the system memory, if the application running does not take this into account. This problem in the implementation is handled by software through the use of zero-padding of the data-set, more precisely zero-padding by 11 instances. Finally, the Telco peripheral was synthesized and packaged into an IP instance for use in the top-level design of the project.

In the final stage of implementing the Telco processor, the Telco IP was inserted on the output of the FIFO instead of replacing the FIFO as originally intended with the design from [4]. This will create a buffer for the system,

allowing continued transmission of data even when either the transfer to or from the external memory is not ready, but only as long it is only one of the instances being occupied and the FIFO not being full. The depth of the FIFO was set to 1024. The final design in Vivado with the Telco IP between the FIFO and DMA unit is shown in Figure 11.

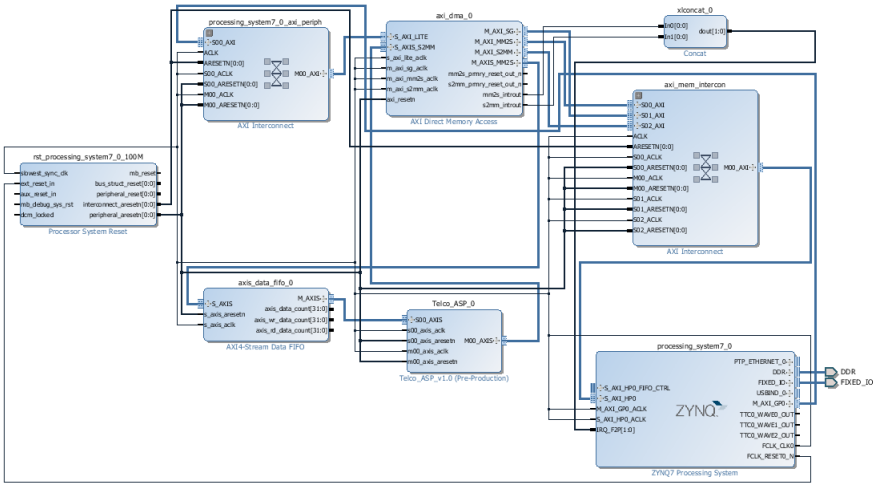

Figure 11: System design of the Telco hardware accelerator on Zynq, as seen in Vivado

As it can be observed from the final system design in Figure 11, DMA capability introduces more complexity to the system design, as a DMA controller and a memory interconnect must be included in the design, Xilinx does however provide these blocks for easier development of DMA capable systems.

3.2.2 Software implementation

The software used to control the Telco processor was developed using the work in [9]. This work provides a sample application which uses DMA to send and receive data. A few functions were slightly modified and a few lines of code were added to the application, in order to increase the amount of transactions and to modify what data was sent, as well as addition of the time-taking code. The pesudo-code for the final application is depicted in 12.

The timer measuring the processing time of the Telco accelerator is begun before the initialization of the platform and the DMA, and ended after all the data have been received. The test of the Telco processor was done by generation of test-data instead of reading test-data from a file. The time-taking includes the generation of test-data to compensate for the lack of reading test-data, this design choice will be explained further in the discussion.

3.3 Hardware for software execution

In order to quantify the performance of the Telco and Monte Carlo processors, a software execution was to be made as well, even though no hardware was

```
init_platform();
init_DMA();
build_tx_data_buffer();
start_DMA();
while(!tx_done){
        tx_done = status();
}
receive_data_to_data_buffer();
while(!rx_done){
        rx_done = status();
}
check_result();
cleanup_platform();
return 0;
```

Figure 12: Pseudo-code for running the Telco accelerator

necessary to do this the processor system still needed to be defined. This was done in Vivado as well, where the processor system IP was added to a design, and an automatic connection wizard was run to connect the external reset, clock and memory signals to the correct pins on the Zynq SoC. The final design as seen in Vivado is depicted in Figure 13.

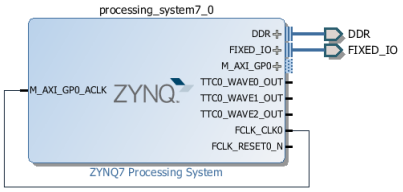

Figure 13: System design of a pure processor system (no programmable logic), as seen in Vivado

4 Results

For each of the benchmarks the accelerator was compared with a software execution on the ARM processor on the Zynq SoC. On all of the runs only a single core of the dual-core processor was utilized. The specifications for the processor are listed in table 2.

Name	Value
CPU frequency	666.667MHz
No. of cores used	1 (core 0)
Memory	512MB DDR3

Table 2: ARM processor specifications for every implementation

Common for all of the benchmarks is that only the execution time was measured, the time and energy used to program the Zynq SoC is not included in the measurements.

In the following subsections the results for the two hardware accelerators will be listed and explained.

4.1 Monte Carlo benchmark

In this subsection the results from the implementation of the Monte Carlo processor will be presented and compared to the results from the software driven benchmark.

4.1.1 Implementation

The Monte Carlo processor was implemented to run on the programmable logic at a frequency of 50MHz, in order not to violate timing constraints. The implementation utilized only a few resources on the programmable logic available, the amount of utilized resources are listed in table 3.

Resource	Utilized units	Utilization [%]
FF	3043	2.85%
LUT	3436	6.45%
Memory LUT	89	0.51%
I/O	8	4.00%
DSP48	8	3.63%
BUFG	2	6.25%

Table 3: Monte Carlo processor programmable logic resource utilization

4.1.2 Functionality

The Monte Carlo processor was successfully implemented and performed the number of iterations that it was instructed to.

There is however a minor flaw in the developed software. The flaw is that the go-signal for the Monte Carlo processor is issued before the time-taking procedure is begun, resulting in that a number of iterations are run without

being timed. The effect of this error is that in order to measure the execution time, a very large number of iterations have to be performed, such that the computation time surpasses the time it takes for the software to reach the point where the time-taking procedure is begun.

This way of measuring execution time may appear to be inaccurate, this was accounted for by taking several measurements of the same number of iterations and observing how much the measurements deviated, the deviation from run to run was $2,200ns$ at most.

4.1.3 Execution time

The total execution time and execution time per element for different amounts of iterations is listed in table 4.

Platform	No. iterations	$t_{tot}[s]$	$t_{element}[ns]$
ASP			
	$<$ 1.68E+07	0.00	0.00
	8.05E+08	7.10	8.81
	1.61E+08	23.20	14.40
	2.68E+08	44.68	16.60
	4.03E+08	71.52	17.80
SW			
	10,000	0.0021	209.20
	100,000	0.0216	216.20
	1,000,000	0.2160	216.00
	10,000,000	2.1600	216.00

Table 4: Execution time for different number of iterations on the Monte Carlo processor and the software implementation

As it can be observed in table 4 the execution time for anything less than 16 million iterations on the hardware accelerator yields an execution time of 0s. The actual time measured was $\sim 650ns$ regardless of whether it was for 10 or 10,000,000 iterations. The 650ns is way too little for the Monte Carlo processor being able to compute 1.68 million iterations, as the Monte Carlo processor has a clock-frequency of 50MHz and should need 1.68 million clock cycles to compute all the iterations (plus a few clock cycles to fill up and empty the pipeline). 16.8 million iterations should yield a theoretical minimal execution time of:

$$\frac{16,800,000}{50MHz} = 0.336s$$

The reason for these numbers is that the go-signal is issued before the execution time measurements begin. The fact that the go-signal is issued before measurements begin was deducted from the plot in Figure 14, where the execution time for small numbers of iterations is near zero; At larger numbers of iterations the execution time scale as expected, e.g. linear.

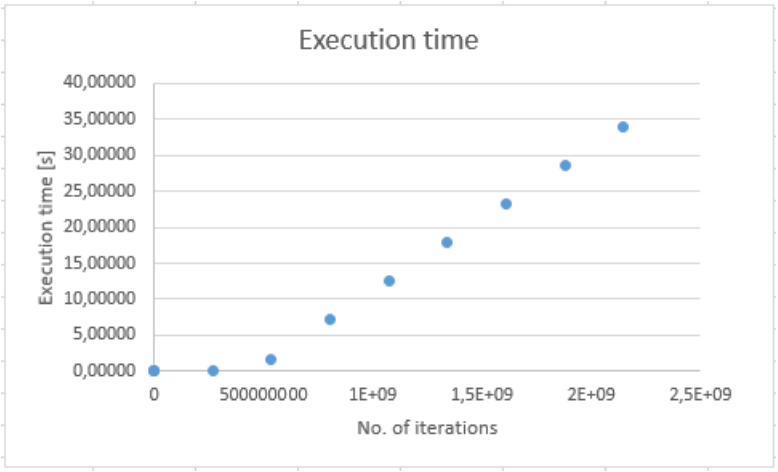

Figure 14: Execution time of the Hardware accelerated Monte Carlo benchmark as a function of no. of iterations

The expected execution time of the Monte Carlo processor was:

$$t_{exec} = t_{init} + t_{element} \cdot n_{elements} \tag{2}$$

Where:

$$t_{element} = 1/frequency$$

.
The first many iterations is completed before time measurements is begun and is thus not included in the measurements. Because the execution time seem to scale correctly at a number of iterations larger than 600 million, it is possible to approximate the execution time by using the difference between two measurements.
Using the difference between two measurements of different amounts of iterations provide the expected increase in execution time. As an example the number of iterations and execution time of two executions as well as the difference between them is listed in table 5.

	No. of iterations	Execution time
	$8.05 \cdot 10^8$	$7.10s$
	$1.07 \cdot 10^9$	$12.46s$
Difference	$2.65 \cdot 10^8$	$5.36s$

Table 5: Execution time for $2.65 \cdot 10^8$ iterations in the HW driven Monte Carlo benchmark as a function of the difference between two different benchmark runs.

The time per iteration for the two individual executions would result in that the Monte Carlo processor ran faster than the clock it was provided with, while the time per iteration from the difference in no. of iterations and execution time, results in a frequency a little less than the one provided to the Monte

Carlo processor. More precisely the frequency of iterations when using the difference between two simulations is:

$$f_{iterations} = 1/t_{iteration} = 1/20.23ns = 49.43MHz$$

$49.43MHz$ is a little less than the frequency of the clock provided to the Monte Carlo processor, the deviation from the clock frequency is low enough to be due to rounding of measurements. Using $1/49.43MHz$ as the execution time per element, the achievable speed-up for the Monte Carlo processor can be approximated to be:

$$Speed - up = \frac{216ns}{\frac{1}{49.43MHz}} = \frac{216.0ns}{20.23ns} = 10.68$$

4.1.4 Energy consumption

Power consumption of the Zedboard was measured during the hardware accelerated and software driven Monte Carlo benchmarks. The measured power consumption and difference between the hardware accelerated and software driven benchmarks is listed in table 6, the power measurement for the longest runs were used to compute the average power consumption.

$P_{ASP}[W]$	$P_{SW}[W]$	$P_{diff}[W]$	P_{ratio}
3.819	3.784	0.035	1.009

Table 6: Instantaneous power consumption for the Monte Carlo ASP and software driven test, based on the average power consumption for the longest run of each implementation

As it can be observed from the last column of table 6 the difference in power consumption is very low, thus the difference in energy consumption rely heavily on the execution time for the Monte Carlo benchmark. Using the time per iteration for the SW and HW implementation of the Monte Carlo benchmark allow comparison of energy consumption. The energy per iteration is given by the equation:

$$E_{iteration} = P \cdot t_{iteration}$$

Energy usage per iteration for the SW and HW implementation is listed in table 7, along with the difference and ratio of energy consumption between the two implementations.

$E_{iteration,ASP}[nJ]$	$E_{iteration,SW}[nJ]$	$E_{diff}[nJ]$	E_{ratio}
77.26	817.34	740.08	10.58

Table 7: Energy per element for the Monte Carlo ASP and software driven test

As expected the ratio in energy consumption is very close to the ratio for execution time. Slightly lower due to the slightly higher power consumption.

4.2 Telco benchmark

The following subsections will present and explain the results from the hardware implementation of the Telco processor, and the measurements of execution time and power consumption of the Telco ASP and the software execution.

4.2.1 Implementation

The Telco processor was implemented to run on the programmable logic at a frequency of 58.824MHz as higher frequencies led to violation of timing constraints. The Telco processor utilized only a fraction of the available resources on the programmable logic, the amount of utilized resources are listed in table 8.

Resource	Utilization	Utilization [%]
FF	5953.0	5.59%
LUT	5916.0	11.12%
Memory LUT	242.0	1.39%
I/O	4.0	2.00%
DSP48	3.5	2.50%
BUFG	1.0	3.12%

Table 8: Telco processor utilization of programmable logic resources

4.2.2 Functionality

The implementation of the Telco processor was working as intended with correct calculations. As expected the returned results was in a delayed fashion, such that the first 11 results was zeros due to the empty pipeline, this was expected and thus compensated for by letting the Telco processor process 11 additional calls of zero call length.

4.2.3 Execution time

The execution time for the Telco processor and the software driven test is listed in table 9 for different amounts of calls.

No. of calls	Telco ASP	Software	Speed-up
10,000	0.00058 s	0.61 s	$\sim$ 1,060
100,000	0.00411 s	6.15 s	$\sim$ 1,495
1,000,000	0.03448 s	61.39 s	$\sim$ 1,555

Table 9: Execution time for the Telco benchmark for both the Telco ASP and software driven test

As listed in table 9, the Telco ASP achieve a very significant speed-up as it in all three test-cases is more than a thousand times faster than the software-driven benchmark. The execution time per element on the ASP is depicted as a plot in Figure 15, with execution time per element as a function of number of calls.

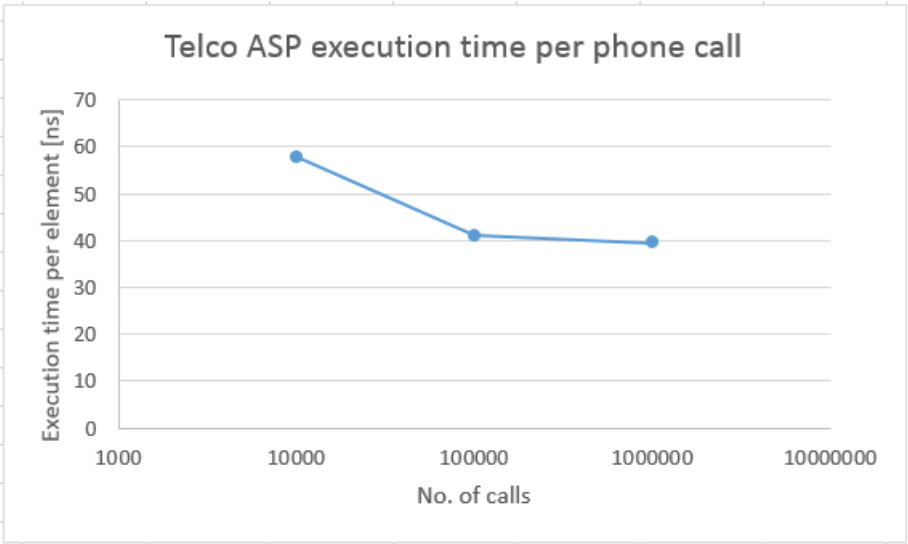

Figure 15: Execution time per phone call as a function of total no. of phone calls, on the Telco processor.

The plot reveals a decrease in execution time per phone call on the Telco processor, when processing larger amounts of phone calls. The decrease is however at a decreasing rate, indicative of that there is a limit for minimum achievable execution time per phone call or the existence of a point of diminishing returns.

From the execution time per element the frequency of processed elements per second can be derived and compared to the frequency of the Telco processor, this will indicate the how much of the time is spent on calculations 3.

$$f_{calculations} = \frac{1}{t_{element}} \tag{3}$$

The resulting frequencies of calculations are listed in table 10.

No. of calls	$f_{calc}[MHz]$	% of f_{PL}
10,000	17.27	29.36%
100,000	24.32	41.34%
1,000,000	25.33	43.06%

Table 10: Rate of elements processed per second in the Telco processor.

From table 10 it can be observed that the Telco-pipeline is active for less than half of the clock-cycles during computing, which is indicative of that the Telco accelerator is limited by the DMA, external memory or the ARM processor system.

4.2.4 Energy consumption

Power consumption of the Zedboard was measured while the Telco benchmarks ran in order to determine the amount of energy used by the Telco ASP and the

ARM-processor. The measured power and energy consumptions of the Telco ASP and the software run benchmark is listed in table 11.

No. of calls	$P_{ASP}[W]$	$P_{SW}[W]$	$P_{diff}[W]$	P_{ratio}
10,000	3.719	3.778	0.059	1.016
100,000	3.716	3.779	0.063	1.017
1,000,000	3.723	3.781	0.058	1.016

Table 11: Power consumption for both the Telco ASP and software driven test

The last column of the table shows the difference in average power consumption of the ASP and CPU driven benchmarks. These numbers indicate that there is a small, but rather consistent, difference in power consumption between running the benchmark on the ASP and the CPU. From the execution time and power consumption, the energy usage by the benchmarks can be derived, energy consumption for each phone call processed is listed in table 12.

No. of calls	$E_{ASP_{element}}[\mu J]$	$E_{SW_{element}}[\mu J]$	$Ratio$
10,000	0.215	231.48	1076.65
100,000	0.153	232.41	1519.02
1,000,000	0.147	232.08	1578.78

Table 12: Energy usage per element processed in the Telco benchmark for both the Telco ASP and software driven test

The energy usage ratio between the ASP and CPU is highly correlated to the speed-up gained by the ASP. This is also shown by the plot in Figure 16.

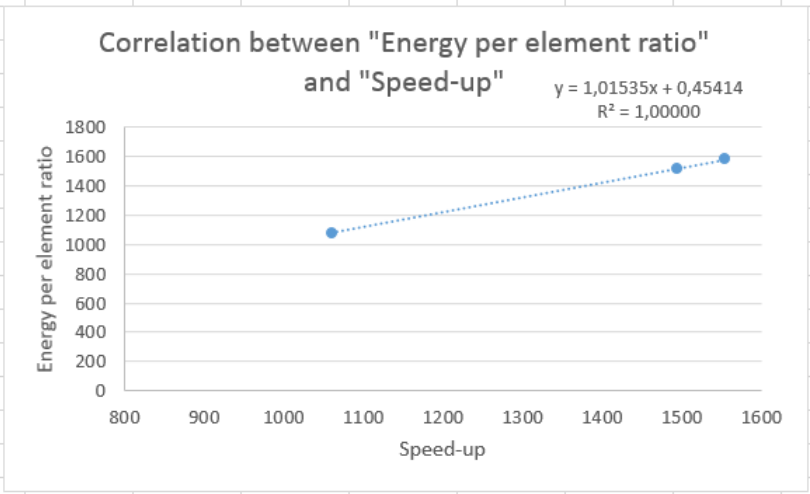

Figure 16: Correlation between Energy usage ratio and speed-up of the Telco ASP

5 Discussion

5.1 Power measurements

In order to measure the energy usage of the different implementations, the power consumption of the Zynq SoC was to be measured during the runs, this is however not possible without modifying the ZedBoard, and thus the measured power consumption is of the entire board. This introduces a couple of problems, one being that the measurements are more receptive to noise, a second problem is that the difference in power consumption between implementations is difficult to asses, as the contribution from peripherals to the power consumption is unknown.

One known error in the measurements from the implementations in this work, is that some of the user-controllable LEDs on the board was used in the hardware implementations but not in the software implementations, giving a difference in power consumption, this difference is expected to be in the range of tens of mW.

Another factor affecting the power consumption measurements is the multimeter used to measure the current draw of the ZedBoard. The used multimeter was an Agilent 34461A. This multimeter is able to filter the noise if the sampling rate is one Power Line Cycle (50Hz) or less, but as the execution time of the benchmarks are rather small a higher sampling rate was used to measure the current draw, resulting in that the measurements are affected by noise. The interpretation of a measurement is depicted in Figure 17, where the noise can be observed as well.

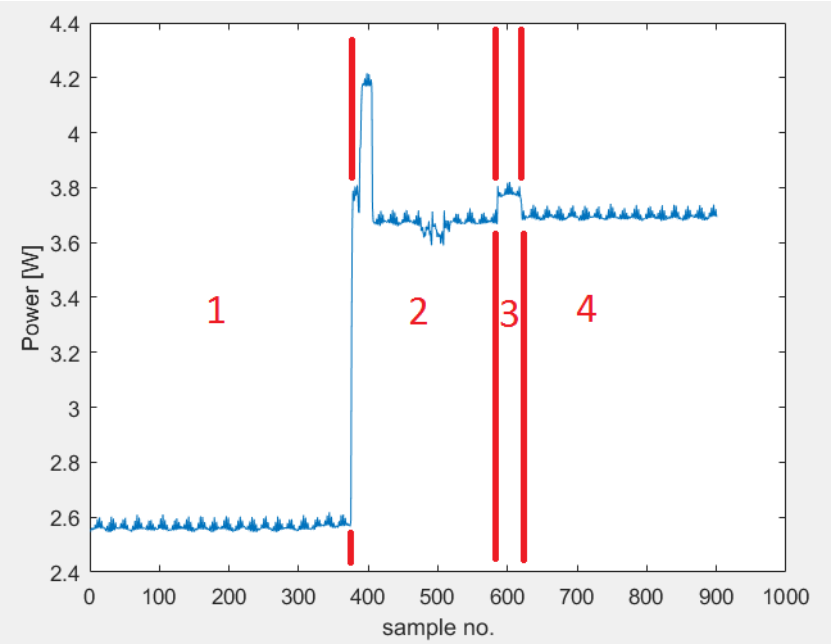

Figure 17: Interpretation of ZedBoard power consumption during SW benchmark, 1. Zynq SoC is idling, 2. SoC is programmed with executable for the CPU, 3. execution of benchmark, 4. card returns to idling

Energy usage was calculated by taking the mean power consumption during the execution of the benchmarks and multiplying it by the execution time measured by the software application, as the software application has a higher resolution in time(nanoseconds opposed to milliseconds).

5.2 Monte Carlo processor

The Monte Carlo processor was implemented successfully on the Zynq platform and performed the Monte Carlo simulations as instructed. There was a minor flaw in the software, which had an effect on the measurements of the execution time. The go-signal for the Monte Carlo processor was issued before the time-taking procedure is begun, resulting in that the measurements of execution time does not directly represent the actual execution time. Taking the difference between two sets of measurements of sufficient size allows determination of execution time for each element.

The Monte Carlo processor had a computation time per iteration that was 10.68 times less than that of the software driven benchmark. The energy consumption per iteration with the Monte Carlo processor was 10.58 times less than that of the SW driven benchmark. The energy consumption ratio is highly correlated to the execution time, as the difference in power consumption between the HW and SW implementation was very small.

Worth noting is that only one of the cores on the ARM processor was used for calculations and the SIMD floating-point NEON unit of the ARM CPU was not utilized, therefore the achievable speed-up with the single core Monte Carlo

processor may be much smaller, or non-existent. There is however plenty of space on the Zynq PL to implement a multi-core Monte Carlo processor; A multi-core Monte Carlo hardware accelerator may be able to sustain a speed-up against the NEON module of the ARM processor.

As the power consumption of the Monte Carlo processor and the software implementation was very small (0.9%), the difference in energy consumption is highly dependent on the execution time of the benchmarks. This was confirmed by the results, and the energy consumption ratio only deviated a very little from the execution time ratio.

5.3 Telco processor

The Telco processor was successfully implemented on the Zynq platform and worked as intended on the hardware side, but left room for a few improvements. The constants used in the Telco processor were fixed in the implementation done in this work, but it would be rather simple to make a new implementation where an AXI-lite slave connection on the Telco IP peripheral would be used to control the processor. The software did not achieve the desired functionality, as the intention was that the calls to be processed by the processor should be read from a file and the cost of the calls written back, this was not achieved, instead a set of test-data was generated and loaded into the memory and read back to the memory. This was however the solution for both the Telco processor driven and software driven benchmark, and therefore the Telco processor and software executed benchmarks can be compared to each other, as the reading and writing of files must be expected to be similar in both instances.

The execution time ratio between the ARM-processor and the Telco-processor was quite substantial, as the Telco processor achieved a speed-up of a 1,000 to 1,550, dependent on the amount of phone calls processed.

Whether the full data-bandwidth between memory, CPU and PL was fully utilized is unknown as the bandwidth was not measured, but results were indicative of that either the DMA, external memory or ARM CPU was limiting the performance of the Telco accelerator.

The power consumption of the Zedboard during the Telco processor driven and software driven benchmarks were really close. The lowest power consumption was achieved with the Telco processor implemented, the difference was however in the range of a mere $0.058 - 0.063W$ or $1.6 - 1.7\%$. It must however again be remarked that the power measurements were of the entire Zedboard and not only the Zynq chip, so as mentioned a rather large bias may be present.

The difference in energy usage between the Telco processor and software driven benchmarks is highly correlated to the execution time, due to the rather small difference in power consumption. The result of this is that the Telco processor is between $1,076 - 1,578$ times more energy efficient than the software execution for the tested amounts of phone calls.

An important thing to note is that only one core of the ARM-processor was utilized in the measurements, thus the speed-up may be closer to half of the measured speed-up, as it is very likely that the processor is limited by it is processing power and not by bandwidth to the memory. The impact on power consumption of using both cores of the ARM processor is unknown, but it is certain that the power consumption of the entire board won't be doubled, due to the bias in present measurements.

6 Conclusion

In this work two different hardware accelerators were to be implemented on a Zynq SoC using a ZedBoard as the platform. The execution time and power consumption of the two hardware accelerators were to be measured and compared to a software execution performing the same task, in order to quantify the performance of the hardware accelerators.

The implementation of the Telco and Monte Carlo processors were both successful, and they performed calculations as instructed.

The Telco processor implementation was based on previous work [5], and only few modifications were made for the processor to work with the Zynq SoC and the Xilinx Vivado development environment.

The comparison between the Telco processor and ARM CPU, showed that the Telco processor was much faster with regard to execution time as it achieved a speed-up of up to 1550. The speed-up in execution time also made the Telco processor much more energy efficient as the difference in power consumption between the Telco processor and ARM CPU was very small. It should however be noted that the power measurements are biased and that the Telco processor may not be 1580 times as energy efficient as the ARM CPU, which was the highest energy per element ratio achieved for the Telco processor.

The achieved step-up in performance of the Telco processor compared to the CPU is much larger in this work than in previous work with the Telco processor [5], but the platform is also different and the programming time of the FPGA was not included in the measurements of this work. Which opens up for some topics that was not investigated in this work, it would for example be interesting to find out what the restricting factor is on the Zynq platform, as the hardware accelerator structure is different than the one in [5], where the bus data-bandwidth between CPU and RAM was the restricting factor. Furthermore, it would be interesting to measure the time it takes to program the FPGA, as this was a very restricting factor for the achievable speed-up in [5].

The Telco processor implementation in this work have some room for improvement, as the call-rates in the used implementation is fixed, and a real application would like to be able to change these rates using the software instead of compiling a new bit-stream. Also the software application responsible for running the Telco processor could be improved, right now the data sent to the Telco processor is generated by the application, the data should be read from a file in order to be usable in a real-life application. The current implementation is however good enough for benchmarking the system performance.

The implementation of the Monte-Carlo processor was successful as the processor ran as intended. The measurements in execution time revealed a minor flaw in the software controlling the processor, as the go-signal was issued before the time measurements were begun. However, taking the difference in iterations and execution time between two runs of sufficient size on the Monte Carlo processor, revealed that the processor ran the iterations at a rate close to the expected frequency.

The point with the Monte Carlo benchmark is however to test the floating-point performance of a system, and as the NEON unit of the ARM processor was not utilized for the benchmarking, the comparison of the speed of the Monte Carlo processor and the speed of the software execution of the benchmark in

this work does not represent the full capability of the ARM CPU.

From the results achieved in this work it can be concluded that, on the Zynq platform, BCD applications can benefit extensively from being run on a hardware accelerator, both in terms of execution time and energy usage per calculation, even though the power consumption was not lowered. For the Monte Carlo processor it is limited what can be concluded as NEON unit in the ARM CPU was not utilized. With the implementation in this work the single core Monte Carlo processor out-performs a single core on the ARM CPU by a factor of 10.6.

In future work the performance of the Zynq SoC and ZedBoard should be investigated in depth, in order to determine the data bandwidth between the ARM processor, external memory (RAM) and the programmable logic, as well as the dual-core and NEON unit performance. This should make it possible to determine how well the hardware accelerators utilizes the available resources.

Bibliography

[1] Adrian Bot, Sergiu Pogacian, and Bogdan Belean. Fpga based hardware architectures for high performance computing applications. *Proceedings - 2012 5th Romania Tier 2 Federation Grid, Cloud and High Performance Computing Science, RQ-LCG 2012*, pages 11–14, 2012.

[2] Altera Corporation. Accelerating high-performance computing with fpgas. *N/A*, pages 1–8, 2007.

[3] Jan Gray. How to design and access a memory-mapped device in programmable logic from linaro ubuntu linux on xilinx zynq on the zedboard, without writing a device driver – part one. `http://fpga.org/2013/05/28/how-to-design-and-access-a-memory-mapped-device-part-one/`, 2013.

[4] Jeff Johnson. Using the axi dma in vivado. `http://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html`. [Online; accessed 06-November-2015].

[5] Jakob Kenn Toft and Alberto Nannarelli. Energy efficient fpga based hardware accelerators for financial applications. *Proceedings of 32nd Norchip Conference*, 2014.

[6] Sanjay Patel and Wen Mei W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4–12, 2008.

[7] Xilinx. Xilinx vivado Product page. `http://www.xilinx.com/products/design-tools/vivado.html`. [Online; accessed 03-November-2015].

[8] Xilinx. Xilinx Zynq SoC. `http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf`. [Online; accessed 03-November-2015].

[9] Xilinx. Ug761 - axi reference guide. PDF, 2011.

[10] ZedBoard. Zedboard Product page. `http://zedboard.org/product/zedboard`. [Online; accessed 03-November-2015].