

Design of Power Efficient FPGA based Hardware Accelerators

Jonas Stenbæk Hegner
s052574

DTU



Kongens Lyngby 2012
IMM-MSc-2012-133

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-133

Summary (English)

The aim of this thesis is to use an FPGA to speed up computations, and to analyse performance trade-offs with respect to latency, throughput and energy consumption.

To do this an option pricing algorithm was chosen as the test case.

The pricing algorithm was implemented in C and the latency tested on a desktop PC.

Then a soft core processor system was implemented, to run the application and to measure the number of cycles it took to do the calculations of option pricing.

The third experiment involved a Application Specific Processor, designed to implement the option pricing algorithm in hardware.

The latency of these systems was measured, and the power consumption was measured for the soft core processor and the ASP.

Finally the three systems were compared with respect to their individual energy consumption. The results clearly show a speed up when comparing Application Specific processors with CPUs, both desktop CPU and soft core processors.

The energy consumption was also lower for the ASP and when using ASPs in parallel an even greater reduction is achieved.

Summary (Danish)

Formålet med det project er bruge en FPGA til at accelerere beregninger. Og derudover at analysere ulemper i forhold til latenstid og energiforbrug. Til dette formål er valgt en algoritme, som udregner optionspriser, som en test applikation. Algoritmen blev implementeret i C og latenstiden teste på PC. Herefter blev et soft core processor system implementeret, så applicationen kunne køres og antallet af clock cykler, der skal til for at køre simulationen, kunne måles. Endelig blev algoritmen implementeret i hardware på FPGA. Latenstiden samt effektforbruget blev målt for de to FPGA systemer. Til slut blev de tre systemer sammenlignet med hensyn til latenstid.

Og de to FPGA systemer blev sammenlignet med hensyn til energiforbruget. Resultaterne viser der opnå en besparelse i latenstid ved at bruge ASPen sammenlignet med de to processore. Også energiforbruget kan reduceres with brug af en ASP. Ved at benytte flere ASPer i parrallel kan energiforbruget reduceres yderligere. Samtidig med at latenstiden falder.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with power and latency in implementations in FPGAs and CPUs.

Advisor: Alberto Nannarelli, Informatics and Mathematical Modelling at the Technical University of Denmark.

Lyngby, 31-August-2012

Jonas Stenbæk Hegner
s052574

Acknowledgements

I would like to thank my advisor Alberto Nannarelli, DTU for all the help and discussions of ideas, he provided during the Work on this thesis. His advice and friendly conversations has been invaluable.

I would also like to thank Edward A. Todirica, DTU for his guidance with respect to tools.

And thank to Joakim Sindholt, DTU for his contribution in form generation of some of the modules.

Contents

Summary (English)	i
Summary (Danish)	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
1.1 Project overview	2
2 Background	3
2.1 Accelerators	3
2.2 Power and energy	5
2.3 Power	5
2.4 Energy	7
2.5 FloPoCo	7
2.6 Development board	8
2.6.1 Power monitor	9
3 Application	11
3.1 Option Pricing	11
3.2 Monte Carlo Simulation	12
4 Soft core processor system	15
4.1 Design	15
4.2 Implementation	16
4.2.1 Soft Core Processor	16
4.3 Test	21

4.3.1	Gaussian random function	21
4.3.2	Cycle counter test	24
5	Application specific processor system	27
5.1	Design	27
5.2	Implementation	29
5.2.1	FPMul_binary32_100	29
5.2.2	FPAdd_binary32_100	29
5.2.3	FPExp_binary32_100	29
5.2.4	FPAdd3_binary32_100	30
5.2.5	Random number generator	31
5.2.6	Floating Point Accumulator	32
5.2.7	Module Latency	32
5.3	Test	34
6	Experiments	37
6.1	Monte Carlo simulation on PC	37
6.1.1	Monte Carlo Simulation on PC results	38
6.2	Monte Carlo simulation on Microblaze	38
6.2.1	Monte Carlo simulation on Microblaze results	39
6.3	Application Specific Processor	40
6.3.1	Application Specific Processor results	40
6.4	Energy consumption	42
6.5	Summary of Power and Energy	43
7	Conclusion	47
7.1	Improvements and Further Experiments	48
	Bibliography	50
A	How To	53
A.1	Testing with chipscope pro	53
A.1.1	Example	54
A.2	Working with XMD	59
A.2.1	Download Application Executable or Data File	60
B	Source Code	63
B.1	Source Code for Monte Carlo Simulation	63

CHAPTER 1

Introduction

As financial computing and computational heavy applications become more and more common, the need for low latency solutions is increasing.

ASICs can be used to reduce the processor load and low power, but they lack flexibility, in a market that is changing rapidly. The time it takes to implement the ASIC chip means it can be obsolete, when it is ready to be deployed.

More and more companies are getting into hardware acceleration, both with graphical processing units and FPGAs. This thesis investigates implementation of a system used in financial computing, specifically to calculate profit of certain trades.

Three systems performing the same task are tested with respect to latency. And two of these are also tested with respect to the power and energy consumption and the impact of parallelization on the latency and power consumption.

During the work on this project two systems has been developed on FPGA, one system using a general purpose soft core processor, and another system as a processor designed to implement the specific calculations on the FPGA. Also a standard PC has been tested with the calculations with respect to latency. The application, a Monte Carlo simulation used in financial computing has been implemented in C and as a processor in hardware.

1.1 Project overview

The one system using a soft core processor runs the Monte Carlo simulation implemented in c, while executing the C program the power and execution time was measured. Then the power and energy consumption was calculated. The same was done for the hardware based Monte Carlo simulation. As for the PC testing, only the execution time was measured running the simulation.

This report describes the design, implementation and experiments of the before mentioned systems. As well as the testing. It also describes some tools needed to do the implementation and testing.

The thesis is organized as follows:

Chapter 2 describes the power measurements in FPGAs as well as the board used to do the project. Lastly it describes a core generation tool called Flopoco, used to generate arithmetic units.

Chapter 3 describes the Monte Carlo simulation application, and some basic background on this specific area of financial computing called option pricing.

Chapter 4 outlines the design implementation and testing of a soft core processor system using the core provided by the vendor.

Chapter 5 describes the application specific processor in both design, implementation and testing.

Chapter 6 outlines the experiments done on the different systems and lastly the results of the experiments.

Project files are included on a CD, also included on the CD is application notes and manuals used for the various parts of the project. The documentation for the Xilinx ML550 board is also included.

Background

2.1 Accelerators

CPUs are general purpose units which can be used for many different things, the problem with this is that not all calculations are very efficient. For example, it is common to offload computer graphics calculations to a Graphics Processing Unit, or GPU. This is one form of hardware accelerator. The purpose of this is to increase the overall performance of the system because the GPU is made specifically for the kind of operations that are used in computer graphics.

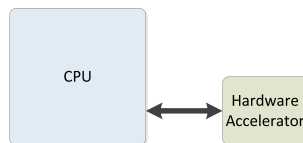


Figure 2.1: CPU to Hardware Accelerator

These hardware accelerators are faster than the general purpose units but also have a narrower field of use. They are also not flexible, when the chip is made, it is made for a very specific task, and to change it requires a whole new chip. Implementing hardware acceleration into an FPGA gives the designer the ability

to change parts of the accelerator if needed. In many cases an Application Specific Processor or ASP implemented in an FPGA may lead to lower latency, but development time is not much shorter than for ASICs, but due to its ability to be reconfigured it is much more flexible and cost effective than ASICs.

There are two different approaches commonly used in financial computing, one is a hybrid approach, where the FPGA is connected to the host PC via network, the PC sends the data that are to be used in the calculations, other calculations are done on the PC. The other is pure FPGA computing, where the FPGA computes all values.

One of the biggest concerns in financial computing is the latency of the calculations and the variations in latency, also called jitter. When using a PC to do the calculations, other processes delaying calculations may cause jitter. In pure FPGA setups there is almost no jitter, and potentially lower latency [Loc12].

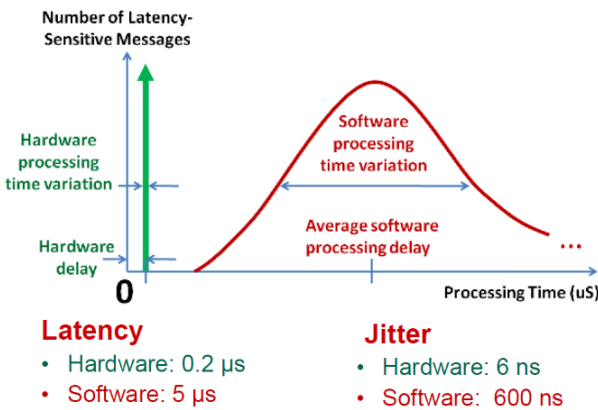


Figure 2.2: FPGA vs Software

As can be seen in figure 2.2 [Loc12] in software a lot of things add to the jitter, like cache misses, other running processes interrupting. These factors can greatly increase the latency.

This is not the case in hardware processors with specific purposes, because they only have to do one thing.

The biggest downside of doing the application specific processor is the development time. As illustrated in figure 2.3 [Loc12] the low latency FPGA solutions have considerably longer development time.

This is the advantage with software implementations, the development time can be as low as days compared to weeks with FPGAs.

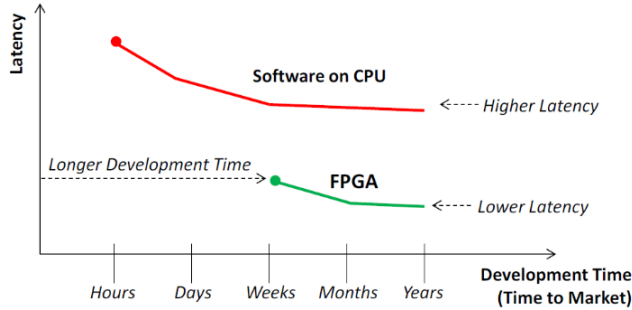


Figure 2.3: Development time vs latency

2.2 Power and energy

This part describes the calculation of power and energy consumption in a system on FPGA boards.

2.3 Power

When considering power consumption in FPGAs, four parameters are important.

- Total power P_{tot} - The total power consumed by the system.
- Bias power P_{bias} - The power used by to bias the FPGA(configuration, memory and interconnects).
- Static power P_{stat} - The power consumed when the system is inactive.
- Dynamic power P_{dyn} - The power consumed by the switching activity in the chip.

The dynamic power can be written is.

$$P_{dyn} = E_{pc(dyn)} \cdot f \quad (2.1)$$

Where $E_{pc(dyn)}$ is the dynamic energy dissipated per cycle and f is the frequency. The FPGA configuration also contributes to the total power, this is what we call bias power P_{bias} , this is measured without downloading a bit stream to the

FPGA(no configuration).

The equation for the biased total power is:

$$P_{tot(bias)} = P_{bias} + P_{stat} + E_{pc(dyn)} \cdot f \quad (2.2)$$

This means to get the total power of the system implemented on the FPGA alone, we need to unbiased the total power. This will make the power figures independent of the FPGA type and size.

$$P_{tot} = P_{tot(bias)} - P_{bias} \quad (2.3)$$

As we can see from 2.2 the total power also depends on the frequency.

Lets call the power measured when reset is pressed P_{reset} and the clock is stopped. So the equation for P_{reset} is:

$$P_{reset} = P_{bias} + P_{stat} \quad (2.4)$$

When measuring power on the FPGA, the total power and the power consumed when reset is pressed, is measured. If we isolate P_{stat} in the equation 2.4 we get:

$$P_{stat} = P_{reset} - P_{bias} \quad (2.5)$$

we can now write the equation for the dynamic power .

$$P_{dyn} = P_{tot} - P_{stat} \quad (2.6)$$

An example of the power calculation is shown below.

The measured values are

First we calculate the unbiased P_{tot} and the P_{stat}

Frequency [MHz]	P_{reset} [mW]	P_{bias} [mW]	$P_{tot(bias)}$ [mW]
100MHz	254.32	212.06	300.49

Table 2.1: Example power values

$$P_{tot} = P_{tot(bias)} - P_{bias} = 300.49 - 212.06 = 88.43 \text{ mW} \quad (2.7)$$

$$P_{stat} = P_{reset} - P_{bias} = 254.32 - 212.06 = 42.26 \text{ mW} \quad (2.8)$$

We can now calculate the dynamic power

$$P_{dyn} = P_{tot} - P_{stat} = 88.43 - 42.26 = 46.17 \text{ mW} \quad (2.9)$$

2.4 Energy

The power consumption is the power the system uses on average during operation. Another parameter is the energy used to perform the execution of a given operation.

$$E = P \cdot t_{op} \quad (2.10)$$

One system could use less power than another, but if the execution time is longer, the system is still using more energy. To determine the energy used we first need the energy per clock cycle E_{pc} and the number of cycles used in a given application. The total energy per cycle can be written as:

$$E_{pc_tot} = \frac{P_{tot}}{f} = P_{tot} \cdot T \quad (2.11)$$

This gives us the equation for the total energy used for a application.

$$E_{tot} = E_{pc_tot} \cdot n_{cycles} \quad (2.12)$$

Where n_{cycles} is the total number of cycles used during execution. With cycle time $T = \frac{1}{f}$.

2.5 FloPoCo

In this project many different arithmetic units are used, to cut down on development time for standard components, an arithmetic core generation tool can be used. One tool that can be used to create these unit, is FloPoCo developed by Florent de Dinechin and his group at university of Lyon, France. This section will describe this tool and its use.

FloPoCo as a tool, is capable of producing synthesizable VHDL code especially suitable for FPGAs, using a command line interface. The tool includes options

to specify target hardware and frequency. It can generate whole pipelined datapaths or single arithmetic operators.

Below is an example of the generation of a floating-point adder.

FloPoCo command for an adder is:

```
./flopoco FPAdder wE wF
```

Where wE is the number of bit of the exponent and wF is the number of bits in the fraction. To get a 32 bit single precision floating-point adder, on a Virtex-5 at 100MHz the command would look like this:

```
./flopoco FPAdder 8 23 -frequency=100 -target=virtex5
```

This will produce the synthesizable VHDL code in a single file.

2.6 Development board

The FPGA development board used in this project is the Xilinx ML550 with a Virtex-5 FPGA chip.

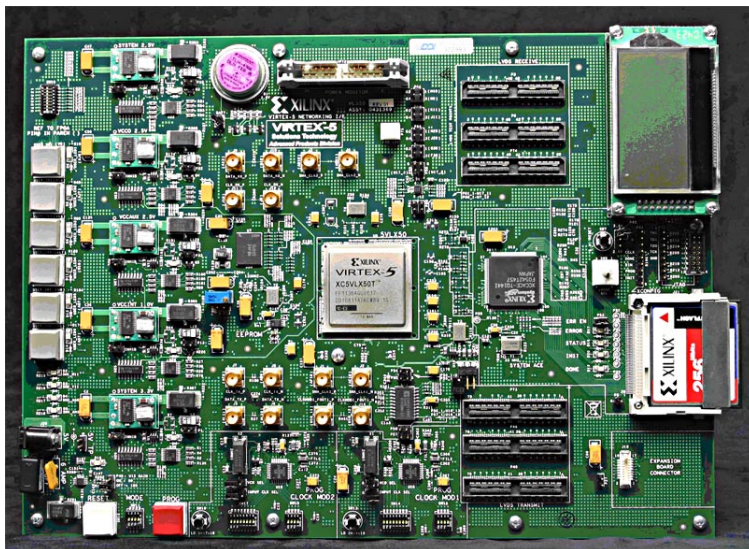


Figure 2.4: Xilinx ML550 FPGA development board

This board was chosen for its physical power monitor connectors and the size of the FPGA chip. This enables measurements of power consumption through the use of a multimeter.

Some features of the board are listed below:

- Xilinx Device: XC5VLX50T-FFG1136 FPGA
 - 7200 slices
 - 6-input LUTs
 - 48 DSP slices
 - 2160 Kb block RAM
- on-board clock oscillators up to 250MHz
- Power monitor connector for powers measurements
- JTAG interface

2.6.1 Power monitor

The Xilinx ML550 board features a power monitor connector for measuring the voltages of the different power supplies on the board. The power connector gives access to three voltages:

- V_{aux} powers things like JTAG and DCM.
- V_{int} is the core voltage, powers the chip.
- V_{cco} is connected to I/O

The voltage important in this project is V_{int} . The schematic of the power monitor connector for the fpga chip is shown below in figure 2.5. To get the reset power P_{reset} the power is measured as above but with the reset pressed.

To measure the total biased power, first the bit stream is downloaded, so the system is running. With a multimeter measuring the voltage over pin 2 and 3 in figure 2.5, between these two is a 10.0mOhm kelvin resistor. This together with the reference voltage of V_{intmon} , gives us the power calculated by:

$$P_{tot(bias)} = V_{int(s+s-)} \cdot V_{intmon} / R \quad (2.13)$$

To measure the bias power P_{bias} , the same measurement as for the total biased power is used, but without downloading the bit stream. We can then measure the unconfigured FPGA.

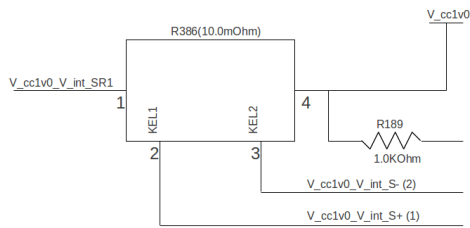


Figure 2.5: Power monitor schematic

Application

As the financial analysis becomes more and more complex, financial computing is getting more important. Financial computing often involves large models. Areas like option pricing can greatly benefit from hardware acceleration. In this chapter option pricing using Monte Carlo simulation is described. This is the application that has been used to do the power consumption experiments in fpga.

3.1 Option Pricing

Options are contracts between an owner of an financial asset and another party, giving the second party the right or option to trade that asset at an agreed upon price, on or before an expiration date, in Europe option can only be exercised on the expiration date. In this case we will only consider European options.

The agreed upon price that is to be paid if the option is exercised is called the strike price. The price of the asset at the moment the option is bought is called the security price. The real price of the asset can fluctuate. So if at the expiration date the real price is higher than the strike price the owner of the option can make a profit, if not then he can choose not to exercise the option.

The difference between the initial security price and the strike price is paid at

the point when the option is bought.

For example A has a stock priced at 50€(the security price at the moment) B can buy the option to trade that stock in 1 year, at the strike price of 60€. Then if after a year the stock security price has increased to for example 70€, B will exercise the option and make a profit of 10€. If on the other hand the stock security price is decreased to 30€ after a year, B will not exercise the option. B's loss in this case will only be the difference between the strike price and the initial security price 10€. A makes a profit of 10€ either way.

To try to predict the profit of an option, several methods have been used, the method described here is parametric Monte Carlo simulation.

3.2 Monte Carlo Simulation

The algorithm for the Monte Carlo simulation [EL04] of option pricing is done as shown in the pseudocode below:

Algorithm 1 Algorithm for Monte Carlo simulation

S_0 = Security Price

K = Strike Price

r = Risk free interest Rate

σ = Security Volatility

t = Time to expiration

n = Number of iterations

r = Risk free interest rate

sum = 0

t = Time to expiration

n = Number of iterations

vsqrtdt = $\sigma\sqrt{t}$

drift = $\left(r - \frac{\sigma^2}{2}\right)t$

expRT = $e^{-r \cdot t}$

for $i = 1 \rightarrow n$ **do**

$S_t = S_0 \cdot e^{drift + vsqrtdt \cdot V_{random}}$

if $(S_t - K) < 0$ **then**

$sum = sum + (S_t - K) \cdot expRT$

end if

end for

return sum/n

σ , r and t are constant, this means drift, expRT and vsqrdt can be calculated in advance, before the execution of the algorithm.

Before the implementation of the algorithm a number of modification, to minimize the execution time has been made. As can be seen from the pseudo code above, the algorithm can be modified by moving some of the multiplications around, this means less cycles to get the final result. For example if K is divided by S_0 so that $K_1 = K / S_0$, the multiplication S_0 can then be removed from the for loop. similarly expRT can be removed from the loop. in the end we can have a modified algorithm shown in algorithm 2 These modifications also

Algorithm 2 Optimized Algorithm for Monte Carlo simulation

```

S0 = Security Price
K = Strike Price
r = Risk free interest Rate
σ = Security Volatility
t = Time to expiration
n = Number of iterations
vsqrdt = σ√t
drift = (r - σ2/2) t
r = Risk free interest Rate
sum = 0
t = Time to expiration
n = Number of iterations
K1 = Strike Price / S0
final = S0 · expRT / n
for i = 1 → n do
  St = edrift+vsqrdt·Vrandom
  if (St - K1) < 0 then
    sum = sum + (St - K1)
  end if
end for
return sum · final

```

makes the implementation of the algorithm simpler. The initial c implementation can be seen in appendix This algorithm uses standard gaussian distributed numbers. The standard gaussian distribution has a variance of 1 and a mean of zero.

Soft core processor system

4.1 Design

To get consistent cycle count we need to implement a CPU in the FPGA, this way we have no other processes to interfere with the execution, as would be the case with a desktop PC. To run programs written in C on a FPGA, a soft core processor is needed. To calculate the energy used by the system to run a given application, we need a cycle counter. The ML550 board used, also requires a clock modulator. The system is shown in figure 4.1.

The operations that are used in the applications for this project, are all single precision floating point.

The cycle count of the applications is unknown, assuming the execution time does not exceed 30min at 100MHz. We would need:

$$cycles = \frac{30 \cdot 60}{10^{-8}} = 180000000000 \quad (4.1)$$

$$N_{bits} = \frac{\log(180000000000)}{\log(2)} = 37.3bits \quad (4.2)$$

The cycle counter is attached with dotted wires in figure 4.1 because it is only present when cycle count experiments are done.

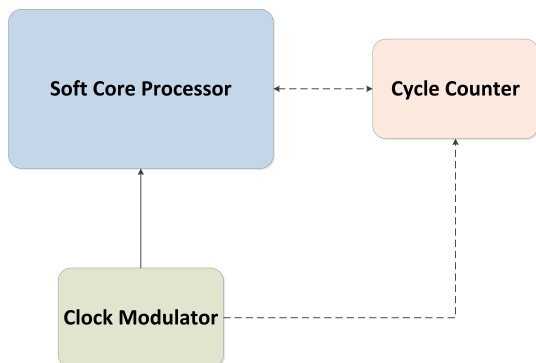


Figure 4.1: Soft core processor diagram

4.2 Implementation

4.2.1 Soft Core Processor

Because the simulations in this project are all single precision, a single precision processor can be used for the C implementation. As a CPU for the C experiments on FPGA, the soft-core processor Microblaze from Xilinx was chosen. Listed below are a few features of the processor.

- 32 bit RISC architecture
- Optional hardware FPU
 - Supports addition, subtraction, multiplication, division, comparison, conversion and square root
- Small area
- Expandable with custom peripherals
- Programming language C/C++
- Supports linux OS
- JTAG debug module

The processor is implemented with 64KB instruction and data memory connected via the local memory bus, this is the fastest memory in Microblaze and there are no cache misses to create jitter.

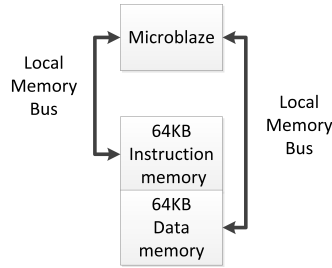


Figure 4.2: Microblaze Architecture

4.2.1.1 Cycle counter

Since the Microblaze uses 32bit registers, the minimum number of register we need to store 37 bits is 2, therefore makes sense to use a 64bit counter. Either way we would need 2 32bit register in the Microblaze. A simple implementation can be used.

4.2.1.2 Clock modulator

Because the ML550 board only has differential clock, a clock modulator is used to convert the differential to a single ended clock. The clock modulator used is an IP core from ISE design suite.

4.2.1.3 Complete system

The complete system is shown in the figure [4.3](#)

When the cycle count is measure a configuration as shown in figure [4.4](#) is used. In this configuration the Microblaze has 3 32 bit registers connected to the outside of the processor.

- Control register at 0xCH400000 - this contains the reset and enable signal for the counter. writing 0x00000002 resets the counter. writing 0x00000001 starts the counter.
- Count MSB register at 0xCH400004 - This contains the most significant bits of the counter.

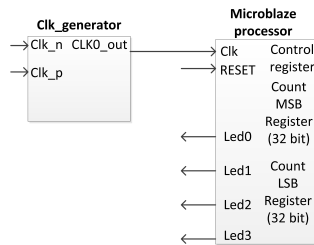


Figure 4.3: Finished soft core processor system

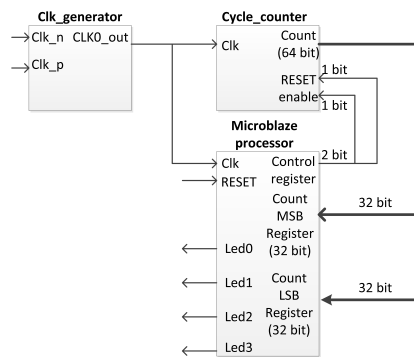


Figure 4.4: Finished soft core processor system for cycle experiments

- Count LSB register at 0xCH400008 - This contains the least significant bits of the counter.

4.2.1.4 Applications

The simulation that is run on the Microblaze is a C implementation of the algorithm in 3.2.

The body of the c code is quite straight forward, the challenging part is the gaussian floating point random number generator. Because C does not have a native function for gaussian distributed random numbers, two different methods were implemented.

4.2.1.5 Random Number Generation

Two different method for random number generation were implemented and analysed. One method commonly used in C, and another method more suited for FPGA implementation. Both method approximate standard gaussian distributions.

4.2.1.6 Random numbers using Box-Muller transformation

This method uses the normal random function in C `rand()`, the generated numbers are then transformed using the Box-Muller transforms.

$$\begin{aligned} Z_0 &= \sqrt{-2 \ln U} \cos(2\pi V) \\ Z_1 &= \sqrt{-2 \ln U} \sin(2\pi V) \end{aligned} \tag{4.3}$$

Where U and V are uniformly distributed random numbers. This method is not suited for FPGA implementation because of the many different and slow operations, like square root, logarithm, sine and cosine. A comparison of the execution time for the two methods can be seen in the test section The source code is shown below:

```

#define PI 3.141592654
float gaussian_ra()
{
    static float U, V;
    static int phase =0;
    float Z;

    if (phase == 0){
        //generate 2 random floats in range ]0,1[
        U = (rand() +1.)/(RAND_MAX+2.0);
        V=rand()/(RAND_MAX+1.);
        //perform Box-Muller transforms
        Z=sqrtf(-2.*logf(U))*sinf(2.*PI*V);
    }else
        Z=sqrtf(-2.*logf(U))*cosf(2.*PI*V);

        phase=1-phase;

    return Z;
}

```

Figure 4.5: C function for random number generator using Box-Muller

4.2.1.7 Random numbers using LFSR

One way of generating uniformly distributed pseudo random numbers in FPGAs, is to use linear feedback shift registers. These can be configured with a predefined word length, the period of the register is then $P_{LFSR} = 2^n - 1$ [Alf96].

In this random number generator, the LFSR is used to generate the fraction of the floating point number, therefore an LFSR of 23 bits is needed, the period is therefore $2^{23} - 1 = 8388607$.

Because the LFSR is uniformly distributed, 4 generated numbers are averaged, the resulting distribution is gaussian. When generating floating point numbers centred around zero directly, the output number can be subnormal, the modules used later in the section 5 are not compatible with subnormal numbers, the generator needs to adjust for this. To do this numbers are generated initially in a positive range away from zero. To limit the range of the numbers, to a small range of positive floating point numbers, the exponent is set to a specific number in this case the exponent is set so that the range is [2.0:4.0].

The range is then offset by subtracting 3.0, which gives a range of [-1.0:1.0]. In the last stage the range is expanded to [-3.5:3.5], this is done to get a variance of approximately 1. In the test section some tests of the random number function is shown. The source code for the C implementation can be seen below:


```

uint32_t lfsr1 = 455u;
uint32_t lfsr2 = 68787u;
uint32_t lfsr3 = 8u;
uint32_t lfsr4 = 98u;

//gaussian_ra generates floating numbers with gaussian distribution

float gaussian_ra()
{
    uint32_t res, res2, ran_temp;
    float Z;
    res=0;
    // generate 4 random numbers from the linear feedback
    //shift registers
    lfsr1 = (lfsr1 >> 1) ^ (-(lfsr1 & 1u) & 0x00420000u);
    lfsr2 = (lfsr2 >> 1) ^ (-(lfsr2 & 1u) & 0x00420000u);
    lfsr3 = (lfsr3 >> 1) ^ (-(lfsr3 & 1u) & 0x00420000u);
    lfsr4 = (lfsr4 >> 1) ^ (-(lfsr4 & 1u) & 0x00420000u);
    //calculate the average to get gaussian distribution
    res+=lfsr1+lfsr2+lfsr3+lfsr4;

    //compile changes this to a shift operation
    res2=res/4;

    //set exponent of random number, to get a range
    //of [2:4]
    ran_temp=res2 & 0x007fffff | 0x40000000;
    //offset range to [-1,1[ and then expand to [-3,5:3,5[
    Z=(*(float*)&ran_temp-3.0)*3.5;

    return Z;
}

```

Figure 4.6: C code for random number generator using LFSR

4.3 Test

4.3.1 Gaussian random function

To test the mean and variance of the random number function, 1000000 random numbers are generated and the variance and mean is calculated in Matlab. Further more a histogram is produce to confirm the bell shape of the distribution. The variance should be close to 1 and the mean should be close to zero.

4.3.1.1 Random numbers using Box-Muller

The histogram is shown in figure 4.8:

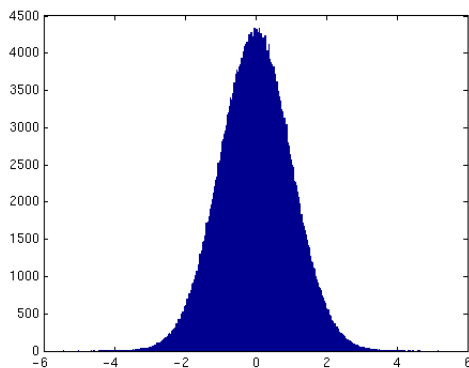


Figure 4.7: Histogram for Box-Muller random number generator

As can be seen the histogram is reasonably bell shaped. The mean and variance is shown in the table below: With mean close to zero and variance close to 1,

Mean	0,0010
Variance	0,9987

Table 4.1: Mean and variance of Box-Muller random number generator

the numbers generated are close to standard gaussian distribution.

4.3.1.2 Random numbers using LFSR

The histogram is shown in figure 4.8:

As can be seen the histogram is reasonably bell shaped. The mean and variance are shown in the table below: With these numbers the numbers generated are

Mean	0,000011916
Variance	1,097

Table 4.2: Mean and variance of LFSR random number generator

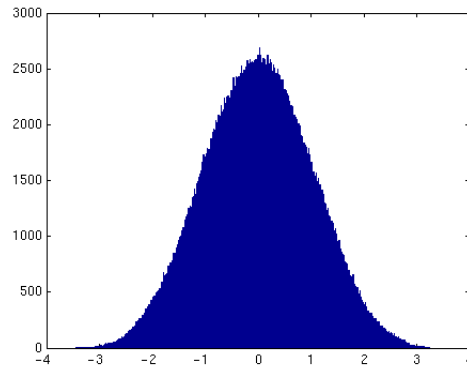


Figure 4.8: Histogram for LFSR random number generator

even closer to a standard gaussian distribution compared to the Box-Muller method. To obtain more precision more LFSRs are needed so that the average could be over more values, for example an average of 16 vaules. But because of the size restriction of the FPGA chip, this is not feasible. The other way is to generate more values from the four, but this would lower the throughput. Since system in this project is more a proof of concept, than an actual product, the random generator is using 4 LFSRs.

4.3.1.3 Random generator time test

A test of the execution time was done on a desktop pc, at two different frequencies and with as little other processes running as possible, to give more consistent results. Specifications of the pc was:

- Processor: Intel Core2duo e6600
- Memory: 4Gb RAM
- OS: Xubuntu 12.04
 - kernel 3.2.0-31-generic-pae
 - gcc 4.6.3

Each method is run 10000000 and 100000 times, and at two different cpu frequencies, while the time is recorded. This is done 10 times and the average is taken.

Frequency [MHz]	LFSR		BOX-Muller		Ratio
	T_{exec} [ms]	n_{cycles}	T_{exec} [ms]	n_{cycles}	
n=100000					
1600	2.5	4000000	23	36800000	8.97
2400	1.6	3840000	15.4	36960000	9.16
Average		3920000		36880000	9.07
n=1000000					
1600	25	40000000	229.1	366560000	9.2
2400	17	40800000	152.5	366000000	9.63
Average		40400000		366280000	9.41

Table 4.3: Execution time test for the random number generators

As can be seen from table 6.1, the LFSR method is almost a factor 10 more efficient, the impact of the two methods on the Monte Carlo simulation can be seen in sections 6.2 and 6.1.1.

4.3.2 Cycle counter test

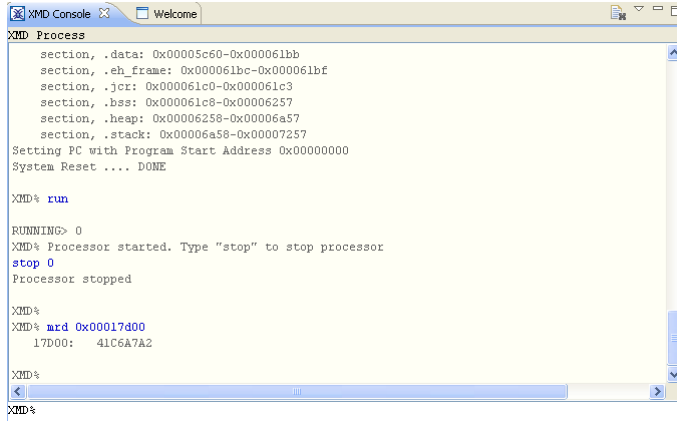
Test of the system has been done in Xilinx SDK, the software development kit for Microblaze. The test is done with a simple program that increments numbers in a for loop, this test can then be executed for a number of iterations of the for loop. The cycle count can then be read with the command mrd, followed by the register number of the counter, in the Xilinx Microprocessor Debugger XMD. Below is a table of the obtained cycle counts. As can be seen in table 4.5 as the

Number of iterations	Plain for loop	For loop with j++	For loop with j++ b++	For loop with j++ b++ g++
10	140	190	240	290
100	1220	1720	2220	2720
1000	12020	17020	22020	27020
10000	120020	170020	220020	270020
100000	1200020	1700020	2200020	2700020

Table 4.4: Cycle Counts from Microblaze cycle tests

number of iterations is increased the number of cycles is increased linearly. The same is the case when the number of calculations per cycle is increased. The numbers are consistent.

The other test performed is a test of the Monte Carlo simulation with 100000 iterations on the Microblaze. First for the Monte Carlo simulation using Box-Muller method as shown in figure 4.9 The other random generator for Monte



```

XMD Process
section, .data: 0x00005c50-0x000061bb
section, .eh_frame: 0x000061bc-0x000061bf
section, .jcr: 0x000061c0-0x000061c3
section, .bss: 0x000061c8-0x00006257
section, .heap: 0x00006258-0x00006a57
section, .stack: 0x00006a58-0x00007257

Setting PC with Program Start Address 0x00000000
System Reset .... DONE

XMD% run

RUNNING> 0
XMD% Processor started. Type "stop" to stop processor
stop 0
Processor stopped

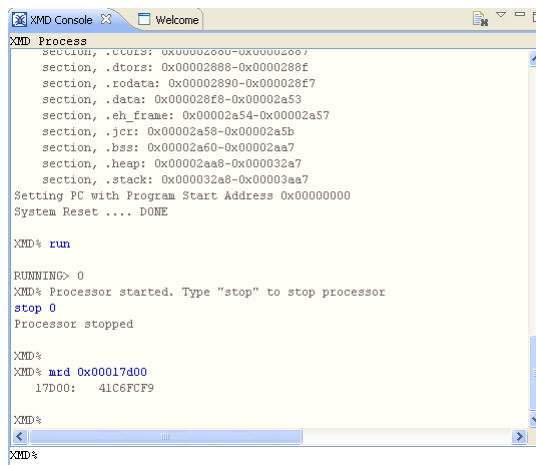
XMD%
XMD% mrd 0x00017d00
17D00: 41C6A7A2

XMD%
XMD%

```

Figure 4.9: ChipScope test of Monte Carlo simulation using Box-Muller on Microblaze

Carlo simulation using LFSR method, the result is shown in figure 4.10 The



```

XMD Process
section, .ctors: 0x00002890-0x00002897
section, .dtors: 0x00002898-0x0000289f
section, .rodata: 0x000028a0-0x000028f7
section, .data: 0x000028f8-0x00002a53
section, .eh_frame: 0x00002a54-0x00002a57
section, .jcr: 0x00002a58-0x00002a5b
section, .bss: 0x00002a60-0x00002aa7
section, .heap: 0x00002aa8-0x000032a7
section, .stack: 0x000032a8-0x00003aa7

Setting PC with Program Start Address 0x00000000
System Reset .... DONE

XMD% run

RUNNING> 0
XMD% Processor started. Type "stop" to stop processor
stop 0
Processor stopped

XMD%
XMD% mrd 0x00017d00
17D00: 41C6FCF9

XMD%
XMD%

```

Figure 4.10: ChipScope test of Monte Carlo simulation using LFSR on Microblaze

simulation has also been run on a desktop PC, the results converted to decimal is: The small variations are due to the fact that they use different seeds, and maybe some differences in the rounding.

Test	Result
PC LFSR	24.873938
Microblaze LFSR	24.873522
PC Box- Muller	24.591898
Microblaze Box-Muller	24.831852

Table 4.5: Results of Monte Carlo Simulation for 100000 iterations

The maximum frequency of the system is 100MHz because the Microblaze is configured with a input frequency of 100MHz. The highest frequency the Microblaze can be configured with is 125MHz

Application specific processor system

5.1 Design

To create a application specific processor implementing the algorithm described in algorithm 2, the floating point operations needed are

- Multiplier
- Adder
- Exponential function
- Accumulator
- Gaussian random number generator

The complete datapath of the application specific processor computing option price by Monte Carlo simulation is shown in figure 5.1.

Besides the datapath a control unit is needed to start and stop the execution, for the control unit to do this a counter is needed.

The complete system looks like this figure 5.2

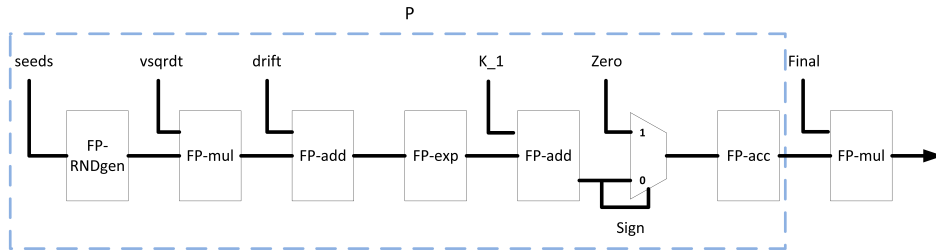


Figure 5.1: Application specific processor datapath

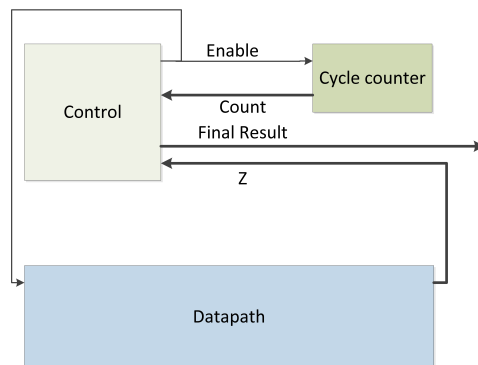


Figure 5.2: Complete Application specific processor system

5.2 Implementation

The modules of the operations multiplication, addition, exponential function and accumulator can be generated with FloPoCo and gaussian random number generator can be implemented following the LFSR method described in section 4.2.

5.2.1 FPMul_binary32_100

This is a standard binary32 multiplier, as described in [EL04] pages 435-442 The module was created in FloPoCo, with a target frequency of 100MHz and target platform Virtex-5.

The top level has the following ports:

Name	Direction	Description
CLK	input	100Mhz clock input
X	input	First operand
Y	input	Second operand
Z	output	Result of the multiplication

Table 5.1: Ports of FPMul_binary32_100

5.2.2 FPAdd_binary32_100

This is a standard binary32 adder, as described in [EL04] pages 417-429 The module was created in FloPoCo, with a target frequency of 100MHz and target platform Virtex-5. The top level has the following ports:

5.2.3 FPExp_binary32_100

The binary32 exponential function is actually a specialized polynomial evaluator. It uses tables to do the operation. It contains range reduction, polynomial

Name	Direction	Description
CLK	input	100Mhz clock input
X	input	First operand
Y	input	Second operand
Z	output	Result of the addition

Table 5.2: Ports of FPAdd_binary32_100

evaluation and reconstruction.

The top level has the following ports, shown in table 5.3

Name	Direction	Description
CLK	input	100Mhz clock input
X	input	Input operand
R	output	Result of the exponential operation

Table 5.3: Ports of FPExp_binary32_100

5.2.4 FPAdd3_binary32_100

This is a standard binary32 3 to 1 adder, is basically a FPAdd_binary32_100 with one extra input. The module was created in FloPoCo, with a target frequency of 100MHz and target platform Virtex-5. The top level has the following ports:

Name	Direction	Description
CLK	input	100Mhz clock input
X	Input	First operand
Y	Input	Second operand
Z	Input	Third operand
R	Output	Result of the addition

Table 5.4: Ports of FPAdd3_binary32_100

5.2.5 Random number generator

This module follows the same approach as the C function from 4.3.1.3. The architecture of the module can be seen below in figure 5.3

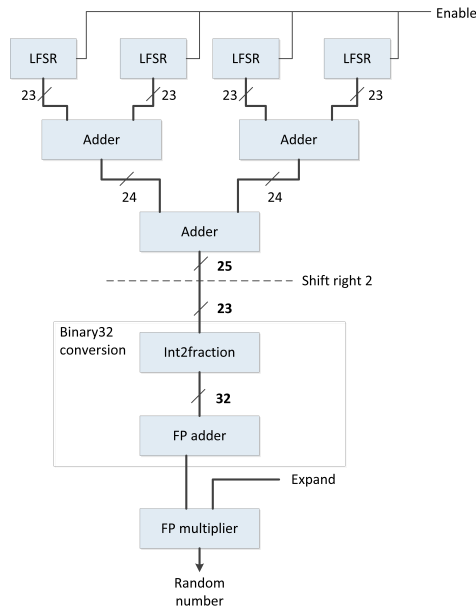


Figure 5.3: LFSR random generator architecture

5.2.6 Floating Point Accumulator

This is a binary32 accumulator it consists of two parts the accumulator, called LongAcc, which accumulates binary32 numbers to a 64 bit fixed point sum, and post normalisation unit, called LongAcc2FP, that transforms the fixed point sum to a binary32 number. The architecture is shown in figure 5.4.

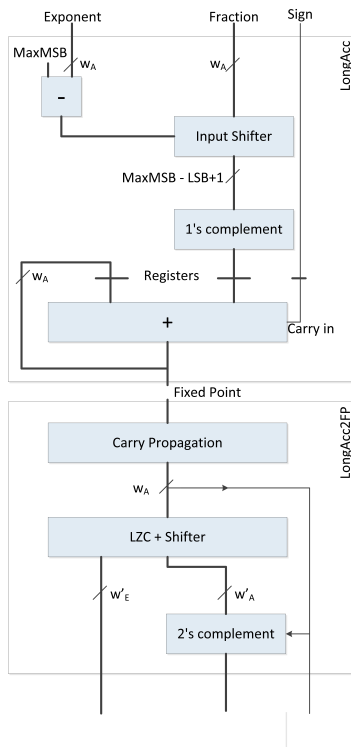


Figure 5.4: Floating point accumulator architecture

This module has the ports.

5.2.7 Module Latency

The latency for the modules used in the ASP is shown in table 5.6.

The system is pipelined so as to give a throughput of one iteration of the simulation per clock when the pipeline is filled. This means the simulation can be

Name	Direction	Description
CLK	input	100Mhz clock input
X	Input	Operand
newDataset	input	Indicates a new accumulation is started (unused)
data_out	Output	Accumulated pga sum in binary32
XOverflow	Output	Indicates the exponent of X is too big
XUnderflow	Output	Indicates the exponent of X is too small
ready	Output	Ready signal

Table 5.5: Ports of FPAdd_binary32_100

	Function	Latency
FPAdd	$x + y$	2
FPExp	e^x	3
FPMul	$x \cdot y$	2
FPAdd3:1	$x + y + z$	2
FPAcc	$x = x + y$	3
RNDgen	random	8

Table 5.6: Latency of the modules and the complete system

executed in the ASP in 100031 cycles.

5.3 Test

For the testing ChipScope was used to do the on board testing and Isim was used to do the simulation of the Monte Carlo simulation.

The Monte Carlo simulation was done with the following parameters in table 5.7.

Name	Decimal Value	Hex Value
n	100000	
S_t	0	
S_0	90	
r	0.1	
sigma	0.25	
t	2.0	
K	100	
K1	1.111	0x3F8E38E4
drift	0.137500	0x3E0CCCCD
vsqrdt	0.353553	0x3EB504F3
expRT	1.221403	0x3F9C56ED
final	0.004397	0x3B901521

Table 5.7: Monte Carlo simulation parameters

The output of the testbench can be seen below The test with Chipscope can be

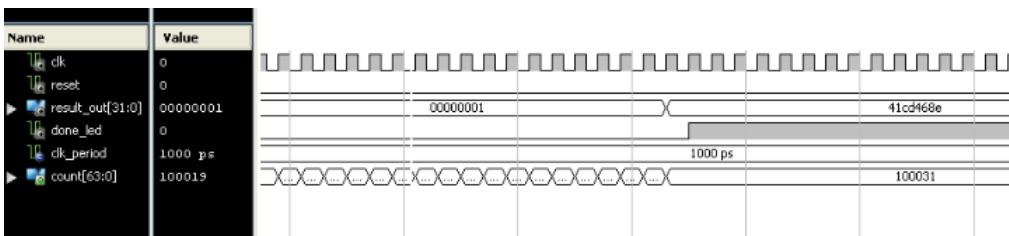


Figure 5.5: Test bench simulation of the ASP

seen in figure 5.6.

Converted to decimal we get the values in table 5.8.

This variation is probably due to rounding in the floating-point units. And

Experiments

This chapter contains a brief description of the experiments made in this project.

6.1 Monte Carlo simulation on PC

In order to compare Monte Carlo Simulation on Microblaze with the same simulation running on a pc, a few timing experiments was done.

For both Simulation with the Box-Muller method and LFSR the execution time was measured on a desktop computer. The pc had the following specifications:

- Processor: I5-3350m processor
- Memory: 8GB RAM
- OS: Xubuntu 32bit
- gcc version 4.6.3
- kernel 3.2.0-31

6.1.1 Monte Carlo Simulation on PC results

The Monte Carlo was also tested on a desktop pc with both random number generators. The simulation was run with 100000 iteration and the execution time measured. The same pc as on page 23.

Frequency [MHz]	LFSR		BOX-Muller		Ratio
	T_{exec} [ms]	n_{cycles}	T_{exec} [ms]	n_{cycles}	
1600	16	25600000	37.1	59360000	2.32
2400	11	26400000	24.3	58320000	2.21
Average		26000000		58840000	2.26

Table 6.1: Execution time test for the Monte Carlo simulation

The table above shows that changing the random generation method to LFSR based gives a factor 2 speed up on a desktop pc.

6.2 Monte Carlo simulation on Microblaze

When doing the experiments on Microblaze, first the bit stream is downloaded to the board. Then Xilinx SDK is used to download the executable to the Microblaze instruction memory.

The execution is started with the command and when execution is done the stop command is issued. If needed registers can be read after execution. See appendix for detailed information on downloading, execution and register read. The Monte Carlo simulation experiments is done to compare the soft core processor to the application specific processor. The experiments also aim to compare the two different ways of generating gaussian distributed random numbers described in 4.2.

These are the experiments:

- Monte Carlo with Box-Muller random generator
- Monte Carlo With LFSR random generator

To measure the cycle count each application is run for 100000 iterations of the simulation.

The power is measured as described in 2.6.1 and calculated as described in 2.2.

6.2.1 Monte Carlo simulation on Microblaze results

Microblaze	V_{mon} [V]	V_{s+s-} [mV]	$V_{s+s-rst}$ [mV]	P_{reset} [mW]	P_{stat} [mW]	P_{dyn} [mw]	$P_{tot(bias)}$ [mW]	P_{tot} [mW]
50MHz	0.995	3.180	2.488	247.556	35.494	68.854	316.410	104.348
100MHz	0.994	3,899	2.505	248.997	36.935	140.005	387.561	175.499

Table 6.2: Power measurements for Monte Carlo simulation on Microblaze

In this table 6.3 is shown the cycle counts for the microblaze experiments.

	n_{cycles}
Microblaze Box-Muller	511,826,387
Microblaze LFSR	163,056,258

Table 6.3: Cycle count for Microblaze

6.3 Application Specific Processor

In the application specific processor the cycle count is already known, because this is what the control unit uses as a stop condition. So the ASP experiments only measures power. This is done for 2 different configurations.

- A system with 1 x ASP
- A system with 4 x ASP

In figure 5.1 the part of the ASP that is repeated is marked with a blue line. The new configurations follow this schematic.

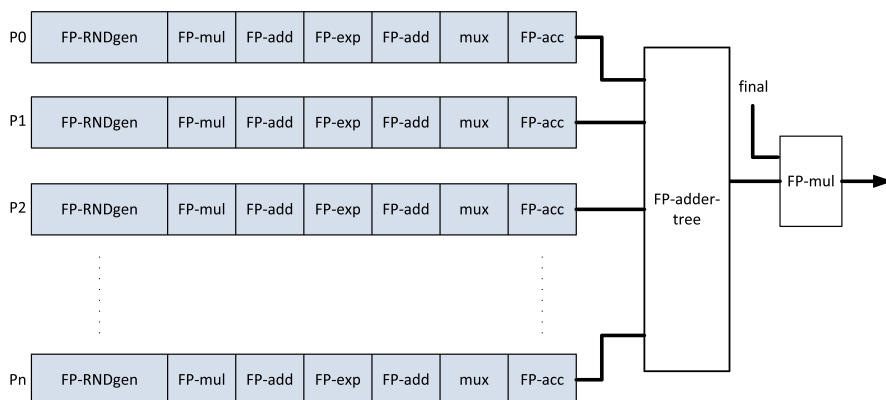


Figure 6.1: ASP multi path implementation

Where the FP-adder-tree consists of 3:1 FPAdd3_binary32_100 and 2:1 FPAdd_binary32_100. The chip on the board used is not big enough for the 8 x ASP and 16 x ASP experiments.

6.3.1 Application Specific Processor results

For the power experiments with ASP we the results shown in tables 6.9 and 6.5.

The figure 6.2 shows the power for the two different ASP configurations. As can be seen the power is increased proportionate to the frequency.

1 x ASP	V_{mon} [V]	V_{s+s-} [mV]	$V_{s+s-rst}$ [mV]	P_{reset} [mW]	P_{stat} [mW]	P_{dyn} [mw]	$P_{tot(bias)}$ [mW]	P_{tot} [mW]
50MHz	0.994	3.065	2.447	243.232	31.170	61.429	304.661	92.599
100MHz	0.996	3.759	2.457	244.727	32.655	131.165	374.396	162.335

Table 6.4: Power measurements for 1 x ASP

4 x ASP	V_{mon} [V]	V_{s+s-} [mV]	$V_{s+s-rst}$ [mV]	P_{reset} [mW]	P_{stat} [mW]	P_{dyn} [mw]	$P_{tot(bias)}$ [mW]	P_{tot} [mW]
50MHz	0.995	4.725	2.646	263.277	51.215	206.861	470.138	258.076
100MHz	0.994	6.890	2.644	262.814	50.752	421.589	684.866	472.804

Table 6.5: Power measurements for 4 x ASP

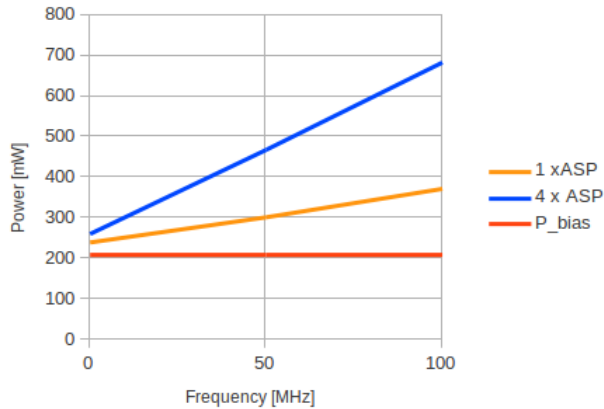


Figure 6.2: Power for different ASP configurations

The area of the different configurations and systems is shown in table 6.6, the increase in area is not completely linear, because of the of some minor differences in the configurations. For example the adder tree is not present in the 1 x ASP configuration. And for the multi ASPs several multipliers can be saved.

	Slices	LUTs	Flipflops	DSP units
Microblaze	1,200(2)	2,891(1)	2,305(1)	5(3)
1 x ASP	840(2)	2,674(1)	1,766(1)	8(4)
4 x ASP	3,078(6)	8,795(4)	4,974(2)	26(14)
8 x ASP	7,903(15)	24,327(12)	14,028(7)	50(26)
16 x ASP	16,256(31)	48,468(23)	28,211(14)	98(51)

Table 6.6: Area for different configurations(the values in () are percentage of the xc5vlx330t chip)

6.4 Energy consumption

From the results in sections 6.2 and 6.3 we get the energy consumption figures of the ASP and the Microblaze implementations.

50Mhz	$E_{pc(tot)}$ [pJ]	T_{exe} [μ s]	Energy [μ J]	Energy Ratio	Speedup
Microblaze Box-Muller	20.869	10,236,527.740	106,813.0	1:1	
Microblaze LFSR	20.869	3,261,305.160	33,917.6	1:3	3
1 x ASP	18.520	2,000.620	185.3	1:577	5118
4 x ASP	51.615	500.620	129.2	1:828	20447

Table 6.7: Comparison of ASP and Microblaze at 50MHz

As can be seen from tables 6.8 and 6.7 using the ASP approach can reduce the latency compared to both PC and Microblaze. And the system is low power.

If we for example compare the energy used for 50MHz and 100Mhz of 1 x ASP we would expect the energy used to be the same, as it is in ASIC, this is not the case. This is because the static power is very high compared to ASICs. From table 6.9 the static power is 30% of the total power for 50MHz and 20% for 100MHz. This affects the energy consumption since the contribution of the

100Mhz	$E_{pc(tot)}$ [pJ]	T_{exe} [μ s]	E_{tot} [μ J]	Energy Ratio	Speedup
Microblaze Box-Muller	17.550	5,118,263.870	89,825.5	1:1	1
Microblaze LFSR	17.550	1,630,652.580	28,617.8	1:3	3
1 x ASP	16.233	1,000.310	162.3	1:553	5118
4 x ASP	47.280	250.310	118.3	1:759	20447

Table 6.8: Comparison of ASP and Microblaze at 100MHz

static power will increase in the 50MHz system because the execution time is longer. This is illustrated in figure 6.3. As can be seen the **static power** only contributes half the time for the 100MHz system. The large static power is something we cannot change, it is part of the power in an FPGA.

The unbiased energy consumption of the 1 x ASP and the 4 x ASP is shown in figure 6.4. Again the static power contributes more when the execution time is longer.

ASP	P_{stat}	P_{tot} [mW]	$P_{stat}:P_{tot}$ [%]	E_{tot} [μ J]
50MHz	31.170	92.599	33.60	185.3
100MHz	32.655	162.335	20.11	162.3

Table 6.9: Ratio between 1 x ASP at two frequencies

6.5 Summary of Power and Energy

A summary table of the power and timing results is shown in the figure below. Here only the Microblaze simulation using the LFSR random number generator is used. The frequency is 100MHz.

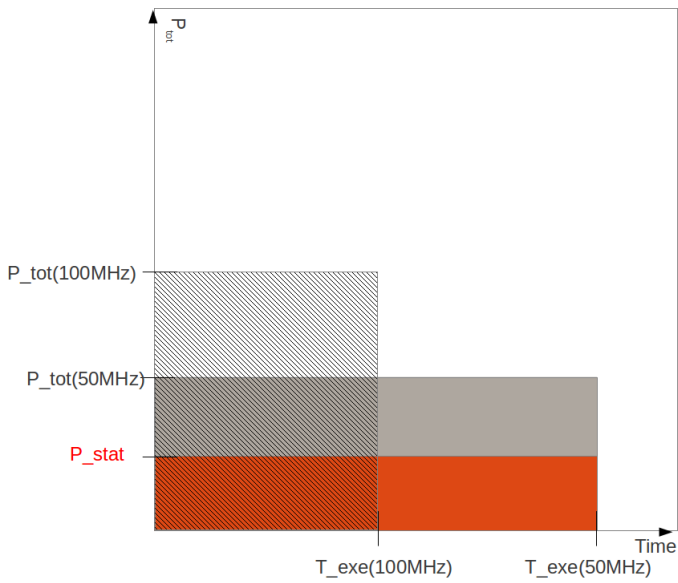


Figure 6.3: Graph of the energy consumed.

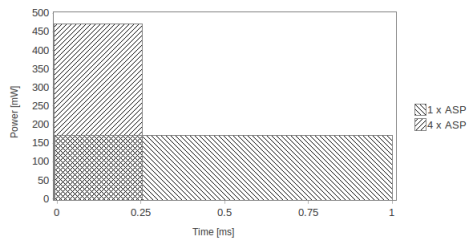


Figure 6.4: Graph of the unbiased energy consumed.

		Timing		Power		
100MHz	n_{cycles}	T_{exe} [μ s]	Speedup	P_{tot} [mW]	Energy [μ J]	Ratio
Microblaze	163,056,258	1,63,652.58	1	175.5	8982.49	1:1
PC	260000	11,000	6	N.A.	N.A.	-
1 x ASP	100031	1,000.31	1630	162.5	162.3	1:553
4 x ASP	25031	250.31	6515	472.8	118.3	1:759

Table 6.10: Summary of power and timing results

Conclusion

The aim of this thesis is to use an FPGA to accelerate an algorithm. And to do an analyse of performance trade-offs with respect to latency/throughput and to investigate energy consumption in the FPGA. In order to obtain these measurements, an algorithm for option pricing was used as a test case.

In this project a soft core processor system was implemented and a C application implementing option pricing using Monte Carlo simulation was created. A second configuration of the system specifically designed to measure cycle count in the execution of a C program in Microblaze

Through test it has been shown that the output is comparable with the C implementation run on a desktop PC.

It has been shown that when measuring power consumption on FPGAs, one has to take into account that the FPGA is biased, with power being consumed by other parameters then the unit we wish to measure power for. A solution to this has also been shown, unbiasing the results and thereby making them independent of the board and the FPGA chip size and type.

It has also been shown that in FPGAs the static power gives rise to some variation in the energy consumed by the tested systems. This is because of internal properties of FPGAs.

A system for measuring execution time or cycles on the Microblaze soft core

processor has been implemented and described. And shown to give consistent measurements.

Two different approaches to gaussian pseudo random generators, has been implemented and compared with respect to latency. Test has been done to show that choosing an effective random number generator has a very big impact on the execution time of the Monte Carlo simulation. Considering only the Microblaze a speed up of 3 was obtained by changing to another random generator.

A solution to the longer development times by using FPGAs, is to use libraries to create as many of the units as possible. Libraries such as FloPoCo can really speed up the implementation of a large datapath.

The problem with these standard solution can be that they are not as optimized as a unit implemented specifically for the current project. Another solution is to use IP cores, these on the other hand are often not as flexible as the libraries. Things such as input output length are often fixed. It may also be more expensive buying licenses for IP cores, than using these more generic units created by libraries or tools.

And with a library such as FLoPoCo the units can be tweaked after generation, because you have access to the source code. A downside of FLoPoCo is that it might not produce the prettiest and readable code.

The results with regard to power and timing has been summerized in table 6.10. The table clearly shows that using an FPGA based ASP approach to option pricing gives a very low energy consumption and a great speedup both in comparison to a soft core processor and compared to a PC CPU. Although power measurements has not been done for a PC CPU, we can safely assume that the power consumption is greater than the few hundred mW used by the FPGA implementation.

7.1 Improvements and Further Experiments

With more time it would have been interesting to investigate a way to measure the power dissipation in a CPU in a desktop. Exchanging FloPoCo with a commercial floating-point library and measure the difference in power consumption. Moving the design to a board with a bigger FPGA, to test even more parallelisation.

Expand the design to be part of a desktop computer.

As for the Microblaze experiments, it would be interesting to try a multiprocessor setup, with 4 or 8 Microblaze processors.

Reduce jitter in PC implementation, maybe by running a minimum Linux or BSD system(No GUI, network etc.).

Investigates other financial applications, like NASDQ OUCH or ITCH.

Bibliography

- [Alf96] Peter Alfke. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators. 1996.
- [Ash02] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2002.
- [EL04] Milos D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.
- [Loc12] John Lockwood. A low-latency library in fpga hardware for high-frequency trading. http://insidehpc.com/2012/08/23/video-a-low-latency-library-in-fpga-hardware-for-high-frequency-trading/?goback=.gde_2354499_member_156862104, 2012.
- [Xil12a] Xilinx. Chipscope pro software and cores user guide. 2012.
- [Xil12b] Xilinx. Edk concepts, tools, and techniques. 2012.
- [Xil12c] Xilinx. Embedded system tools reference manual. 2012.
- [Xil12d] Xilinx. Microblaze processor reference guide. 2012.

A.1 Testing with chipscope pro

After test bench testing it is often useful to test on the board. To test system output and internal signal a tool like Xilinx Chipscope Pro is very useful. This section describes the basics of getting started with Chipscope. The tool flow of testing with Chipscope could be like this:

- Prepare ise.
 - Set keep hierarchy to yes.
 - Set clock to jtag clock.
- Add chipscope module.
- Synthesize.
- configure chipscope module.
 - Add ILA(Integrated Logic Analyzer) unit.
 - Set number of signals to analyze and their data width.
 - Make connections between ILA and the desired signals.

- Open analyze With chipscope.
 - Download bitstream.
 - Start capturing signals.

A.1.1 Example

This Example uses the Floating Point multiplier project FP_Multiplier_genesys, this consists of a control unit that loads two floating point vector and the result of multiplying these together(the expected result), from a ROM. The two vectors are transmitted to a floating point multiplier. The output of the multiplication is then sent to the control unit, which compares it to the expected result and sets an led high if they match.

A.1.1.1 Preparing project

To make it easier to find the signals, that are to be captured, set the Keep Hierachy to yes, this preserves all modules as they are including their signals.

Right click Synthezise -XST → Process Properties → Set Keep Hierachy to yes.

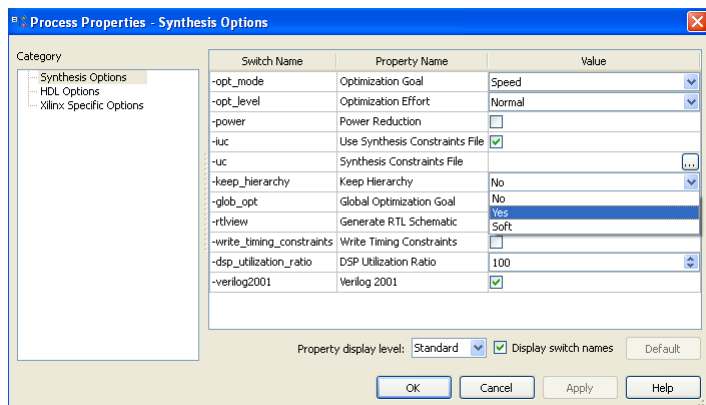


Figure A.1: Synthezise Process Properties

To set the clock.

Right click Generate Programming File → Process Properties → Startup Options → Set FPGA Startup Clock to JTAG clock.

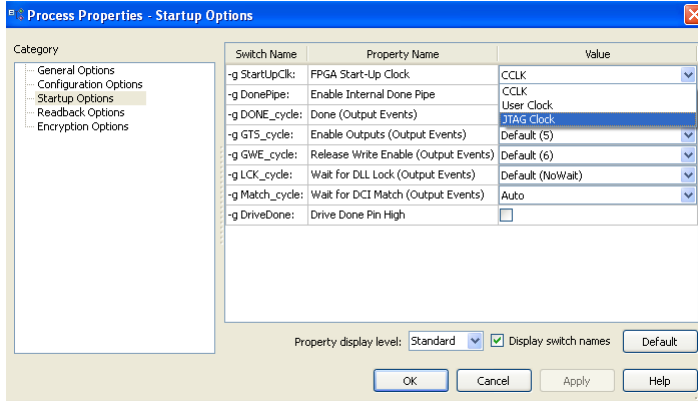


Figure A.2: Generate Programming File Process Properties

A.1.1.2 Adding Chipscop Module

A Chipscope module is needed to run the chipscope analysis, call the module `chipscope_probe`.

New Source → Choose ChipScope Definition and Connection File.

A.1.1.3 Configure Chipscope Module

To open the configuration click the `chipscope_probe.cdc` file in the Hierachy view, this opens the Chipscope Core Inserter.

In the opened window click next. To add the ILA click New ILA Unit and click next. Set the Number of input trigger ports to 3. Set the Trigger Width of TRIG0 and TRIG1 to 32 bits and TRIG2 to 1 bit and click Next.

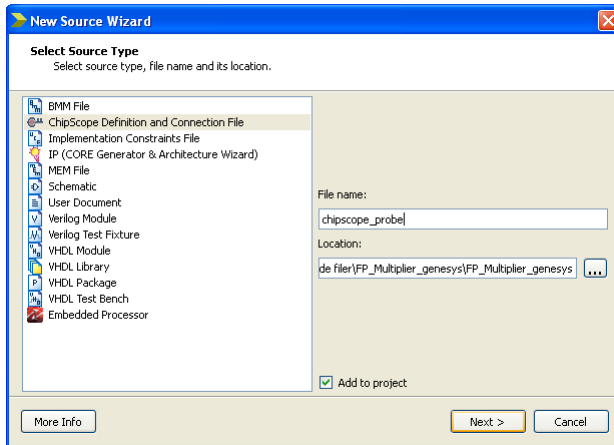


Figure A.3: Make Connections window

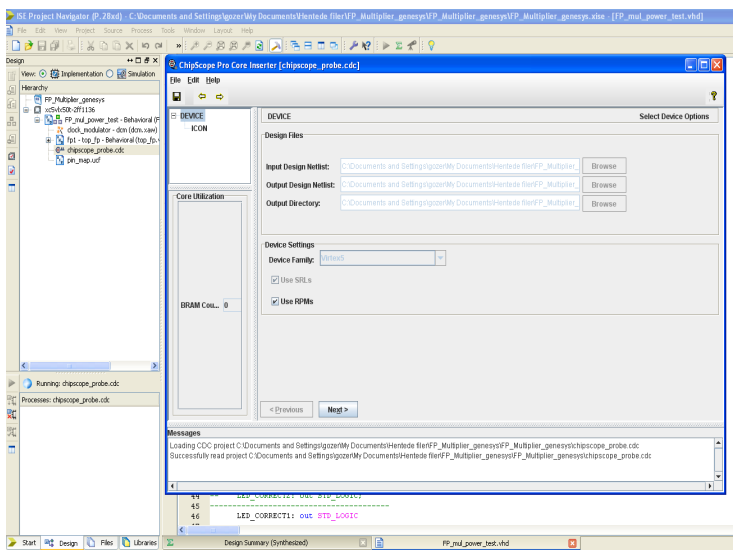


Figure A.4: Chipscope Core Inserter

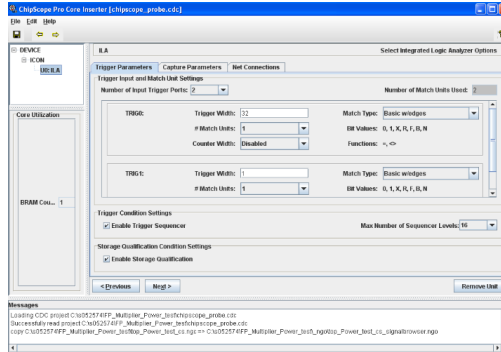


Figure A.5: Configure data triggers

In the Net Connections tab click Modify Connections.

Select the CLK signal and click Make Connections. In the Trigger/Data Signals Select Z (all 32 bits) in the module fp1 and click Make Connections. Do the same for TP1 with correct_led from the module CONTROLLER.

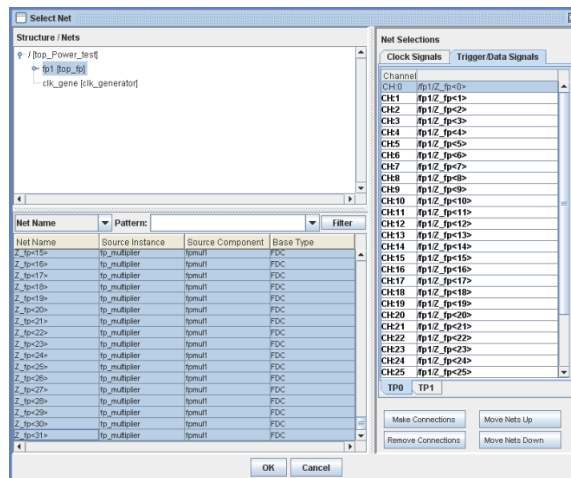


Figure A.6: Modify Connections window

To exit the configuration click Return to Project Navigator, click yes to save.

A.1.1.4 Analyzing with ChipScope

Select the top module of the design and click Analyze Design Using ChipScope to open ChipScope Pro.

In the JTAG Chain dropdown menu choose Xilinx Platform USB Cable, and click OK on both popups.

Right click device and choose configure click OK to configure the board with the bit stream.

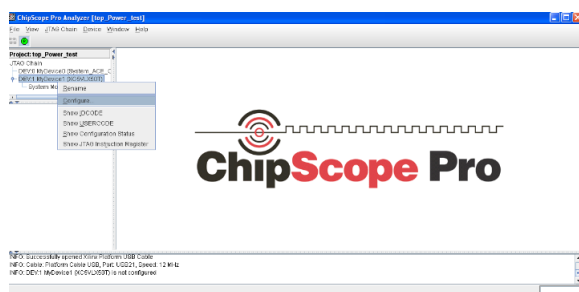


Figure A.7: Download the bit stream to the board

Double click Waveform and Trigger Setup.

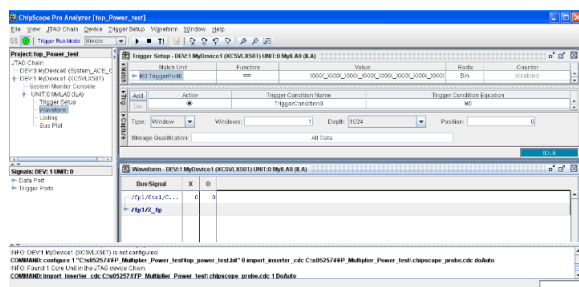


Figure A.8: Setting up the ChipScope window

To run the analysis click the play symbol. This will start the signal capture. As can be seen on figure A.9 chipScope captures the output of the multiplier Z and the led indicating that the multiplication was done correctly.

The box Trigger Condition Equation can be used to start the capturing of signal, when a specific condition is met, for example a signal switching to high. This is

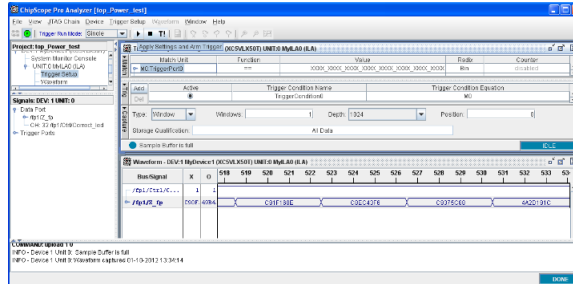


Figure A.9: Running ChipScope

useful to capture states at a specific time, i.e a done signal switching to indicate operations are completed. M0 is for triggerport TP0.

A.2 Working with XMD

The Xilinx Microprocessor Debugger is used to run and debug the a Microblaze system or application. This section will describe some of the useful commands needed when working with the debugger. The XMD is a command line interface in the Xilinx SDK, a screenshot of the interface is shown in figure A.10. An example of the flow of operations used in this project for testing:

- Connect to the Microblaze.
- Download executable and optionally data files to the Microblaze.
- Run executable.
- Stop executable.
- Read or write registers.

A.2.0.5 Connecting to Microblaze

In order to issue commands to the Microblaze, we need to connect to the Microblaze debug module. To connect to the debug module after the bit stream has been downloaded(i.e with Impact) to the board, open the XMD window. The following command establishes the connection:

```
mb mdm
```

A.2.1 Download Application Executable or Data File

To download the application that is to be run on the Microblaze.

```
dow <YourApplication.elf>
```

This will download the .elf YourApplication to the instruction memory of the Microblaze. To download a data use:

```
dow -data <YourData.dat> <address>
```

This will download the .dat file YourData, this file is a binary file in the data memory at the address specified(an editor like hexedit can be used to create the .dat files). The address is a 32 bit hex number. An example could be 0xCF400000.

A.2.1.1 Run Application

To start the execution of the .elf, use the command:

```
run
```

A.2.1.2 Stop Application Execution

To stop the execution use:

```
stop <processor id>
```

Where processor id indicates the number of the processor that is to stop, in case the system has more than one, the processor ids start at 0. One thing to be careful of is to issue the stop command right after the run command, before the XMD terminal outputs "Processor Started. Type "stop" to stop processor". If stop is issued before the connection between pc and board is often corrupted. An a full shut down of the board with redownloading of the bit stream necessary.

A.2.1.3 Reading and Writing Registers

Reading memory registers is done with the command:


```
mrd <address> <x>
```

Where address is the memory address and x is used to read subsequent registers, i.e. 4 will read out the register on address and the subsequent 3 registers.

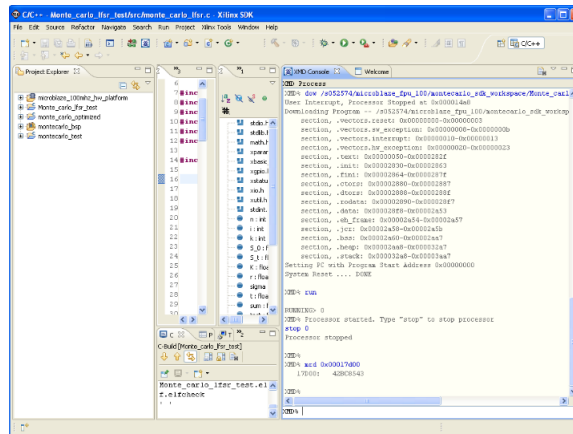


Figure A.10: XMD terminal in SDK

```
mwr <address> <value>
```

Where value is the 32 bit hex number to write in the specified address.

APPENDIX B

Source Code

B.1 Source Code for Monte Carlo Simulation

```
/*  
 * File: monte_carlo_main.c  
 * Author: s052574  
 *  
 * to compile run gcc -std=c99 monte_carlo_sim_opt.c -o monte -lm  
 *  
 * Created on May 16, 2012, 2:48 PM  
 */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
  
float gaussian_ra(void);  
int main(void) {  
    int n = 100000;  
    float S_0=90.0;  
    float S_t=0.;  
    float K=100.0  
  
    float r=0.1;  
    float sigma=0.25;
```

```

float t=2.0;

float K1=100.0/S_0;
float vsqrdt = sigma*sqrtf(t);
float drift = (r-(sigma*sigma/2))*t;
float expRT = expf(r*t);
float sum = 0.;
float final = (S_0*expRT)/n;

srand(2);
for(int i=0; i<n; i++){
    S_t = expf(drift+vsqrdt*gaussian_ra());

    if (S_t-K1 > 0){
        sum+=(S_t-K1);
    }

}

sum=sum*final;

printf("Average_sum_is_%f\n", sum);

return (EXIT_SUCCESS);
}

//gaussian_ra generates floating numbers with gaussian distribution
#define PI 3.141592654
float gaussian_ra()
{
    static float U, V;
    static int phase =0;
    float Z;

    if (phase == 0){//generate two floating point numbers
        U = (rand()+1.)/(RAND_MAX+2.0);
        V=rand()/(RAND_MAX+1.);
        //Use Box-Muller transforms to change the
        //distribution to gaussian
        Z=sqrtf(-2.*logf(U))*sinf(2.*PI*V);
    }else
        Z=sqrtf(-2.*logf(U))*cosf(2.*PI*V);

    phase=1-phase;

return Z;
}

```