# ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
# ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ ΚΑΙ ΥΛΙΚΟΥ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

<<***Development of an Experimental Framework & Evaluation of Applications for Execution in Partial Reconfigurable FPGAs***>>

Γεώργιος Νούλας

Επιβλέπων: Καθηγητής Απόστολος Δόλλας

Εξεταστική Επιτροπή:

| | |
|---|---|
| Απόστολος Δόλλας | Καθ. Π.Κ |
| Διονύσης Πνευματικάτος | Καθ. Π.Κ |
| Ιωάννης Παπαευσταθίου | Επίκ. Καθ. Π.Κ |

# Ευχαριστίες

Καταρχήν θα ήθελα να ευχαριστήσω τον καθηγητή μου Απόστολο Δόλλα για την ευκαιρία που μου έδωσε να γνωρίσω το Hardware μέσα από τις δραστηριότητες του εργαστηρίου Μικροεπεξεργαστών και Υλικού. Τον ευχαριστώ επίσης για την απόφαση του να με επιβλέψει για την εκπόνηση της διπλωματικής μου εργασίας. Με τη συνεχή καθοδήγηση καθώς και το αμέριστο ενδιαφέρον του κατά τη διάρκεια εκπόνησης αυτής της εργασίας έκανε δυνατή την επίτευξη του στόχουμου.

Επίσης, θα ήθελα να ευχαριστήσω την επιτροπή της διπλωματικής μου εργασίας, Καθηγητή Δ. Πνευματικάτο και Επικουρο Καθηγητή Ι. Παπαευσταθίου για τη συμβολή τους στην εργασία αυτή.

Ευχαριστώ τον διδακτορικό φοιτητή Κυπριανό Παπαδημητρίου για τη συνεχή καθοδήγηση καθώς και την εντατική και πολύτιμη του βοήθεια σε όλα τα στάδια εκπόνησης της διπλωματικής εργασίας μου.

Επίσης θα ήθελα να ευχαριστήσω:

Τον μεταπτυχιακό φοιτητή, Παναγιώτη Δαγριτζίκο για την πολύτιμη βοήθειά του με τους κρυπτογραφικούς αλγορίθμους.

Όλους τους φίλους μου για το συνεχές ενδιαφέρον τους και την αμέριστη ηθική τους συμπαράσταση.

Την αδελφή μου για την συμπαράσταση, υπομονή και φροντίδα τα φοιτητικά αυτά χρόνια.

Τη Σεμίνα, που άντεξε εμένα και το Πολυτεχνείο ολο αυτόν τον καιρό και προσέφερε διαρκώς κουράγιο και υποστήριξη.

Τέλος, και περισσότερο απ' όλους, ευχαριστώ τους γονείς μου για την βοήθεια και υπομονή τους σε όλα τα χρόνια των σπουδών μου. Με το συνεχές ενδιαφέρον τους σε κάθε βήμα μου και με την ηθική και οικονομική συμπαράσταση τους έκαναν δυνατή την επίτευξη των στόχων μου. Θα ήταν αδύνατη η ολοκλήρωση της εργασίας αυτής χωρίς το κουράγιο και την υποστήριξη που διαρκώς μου έδιναν.

# Table of Contents

# Chapter 1

## Introduction

In this chapter we will be getting a first look at partial reconfiguration, what it is, where it is used, its advantages and disadvantages. Finally, information about this thesis will be given and the contributions of our work are pointed out.

### 1.1 Reconfiguration

One of the most interesting features in today's FPGAs is their ability to allow runtime-dynamic reconfiguration. This feature allows replacing a module on the device, while the reset remains intact and continues its operation. Run time reconfiguration is best used, when areas of a program are too complex or too numerous to be loaded simultaneously onto the available hardware provided by the FPGA. Run-time reconfiguration can be seen as virtual hardware. Like virtual memory, here, the physical hardware present is much smaller than the sum of the resources required by each of the configurations.

Reconfigurable computing is becoming an important part of research in computer architecture and software systems. Due to their flexibility to change over time and provide a method to map circuits into hardware, these systems have the potential to achieve far greater performance than software as well as possibly exploiting a greater degree of parallelism. Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software but also maintaining a higher level of flexibility than hardware. These systems use FPGAs or other programmable hardware to accelerate algorithm execution by mapping compute-intensive calculations to the reconfigurable substrate. A general microprocessor may also be used with the above hardware to control the reconfigurable logic and to execute any program code that cannot be efficiently accelerated.

FPGAs and reconfigurable computing are used today in several applications. Data encryption, for example is able to leverage both parallelism and fine grained data manipulation. Other applications that have been shown to exhibit significant speed-ups are automatic target recognition, string pattern matching, Golobm Ruler Derivation and more. Partial reconfiguration is the cornerstone for power-efficient and cost-effective software defined radios (SDR). SDRs are becoming reality for the defense industries as an effective and necessary tool for communications. In [14] partial reconfiguration was used in a video-based driver assistance system for cars.

## 1.2 Advantages of Dynamic Reconfiguration

The two biggest problems designs face are fitting more logic into an existing device and fitting a design into a smaller and cheaper devise. With the use of partial reconfiguration, a designer can overcome these problems.

Partial reconfiguration gives the ability for multiple design modules to share physical resources on an FPGA board. This allows for reduction of the hardware resources being used, which means that designs are smaller, therefore leading to smaller and cheaper FPAGs being used. This potential of using less space on the board, leads to consuming less power, which is a huge factor in today's embedded designs. Being able to change the hardware configuration allows for the implementation of highly specialized circuits. Partial reconfigurable systems can adapt to changes in their environment, their input data or their mission specifications. This capability makes the system more efficient as compared to a generic one, which cannot be optimal for a number of different situations.

The ability to change the hardware dynamically in a single FPGA also provides additional advantages. Partial reconfiguration:

- Provides real-time flexibility in the choice of algorithms or protocols available to an application at anytime.
- Enables the use of new techniques in design security.
- Improves FPGA fault tolerance
- Accelerates configurable computing
- Reduces bitstream storage requirements

*Figure 1* shows some of the advantages when using partial reconfiguration.

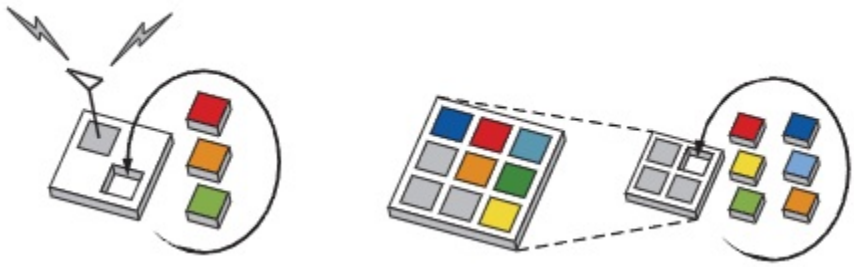

**Figure 1.** Modifying Functionality and Reducing Size using Partial Reconfiguration

Of course, with the advantages come some tradeoffs, which must not be neglected. Due to the time required to download the configuration data before the system is ready to execute can degrade the execution time. This delay is known as reconfiguration overhead. As stated above, the configuration bitstreams need to be stored somewhere in the system. The tradeoff here is the limited but fast on-chip memory, and the slow but large and inexpensive external memory. Of course, using the external memory means large reconfiguration overhead, due to the slow transfer rate. Due to the complex of the design the design cycle is larger, which could mean to a slower time to market.

Before deciding to use dynamic reconfiguration, a performance evaluation is needed to study the system behavior and point out the bottlenecks. Only after these results, can we decide whether dynamic reconfiguration suits the needs of an application.

## 1.3 Contributions of This Work

This work presents a detailed framework and evaluation of applications for execution using reconfigurable FPGAs. We were mostly interested in the reconfiguration time and the overhead added to our design.

For this work a Python script was created in order to gather all measurements taken during the experiments and import them to excel for further processing. A total of four systems were setup in order to test and evaluate the reconfiguration process. For evaluation of the setups the PRCC tool was used in order to provide theoretical results.

## 1.4 Structure

Chapter two provides references to related work to this thesis. Chapter three discusses the partial reconfiguration tools and the operating system used. It also provides an in depth description about the FPGA board that was used. Chapter four presents a series of measurements that were taken regarding different components on the Virtex-5 such as the serial port and BRAM memories. Chapter five presents the two algorithms used for evaluation and the benchmark results for the partial reconfiguration evaluation. Finally, this thesis will close with chapter six were we will conclude this thesis and provide information about future work.

# Chapter 2

## Related Work

When considering building a design using partial reconfiguration several issues need to be studied. This chapter goes through the existing technology in order to discuss the problems that ... to be studied. It starts with the recent trends in PR design flow which demonstrate that although a significant amount of work has been done, PR flow is still at its early stage, especially with regard to developing a painless way if designing applications with PR. Then it examines the overhead added by reconfiguration process and the effect of the different setups have on it. It continues with ways that have been proposed in order to hide the reconfiguration overhead. Finally, applications that have been deployed using PR technology are discussed.

### 2.1 Recent Trends in PR Design Flow

In [2], Lysaght *et al.* 2006 proposed a design methodology for partial reconfiguration using a Virtex-4 board. This paper introduces also the capabilities of the Virtex 4 family. The most important developments of this FPGA family are the following:

i) The hardwired tri-state buffers have been replaced with pre routed bus macros. These provide better communication between static and dynamic regions and much more flexibility.

ii) Reduction in the granularity of the unit of reconfiguration from a full device column to a smaller unit of 16 CLB's and is independent of device size or family. All the Virtex-4 configuration frames consist of forty-one 32-bit word resulting in a total of 1,312 bits per frame.

iii) The new ICAP port now has a 32 bit (old one was 8 bit wide) input and output and can communicate at a max speed of 100MHz. This, along with the 16 CLB's can increase the reconfiguration speed by as much as an order of magnitude for smaller modules.

The design process is enabled by two key enhancements to the mainstream design tools. First, the region being reconfigured can be of any rectangular size. The second major change permits signals in the static design to cross through partially reconfigurable regions without the use of a bus macro. The new design flow is illustrated in *Figure 2*.

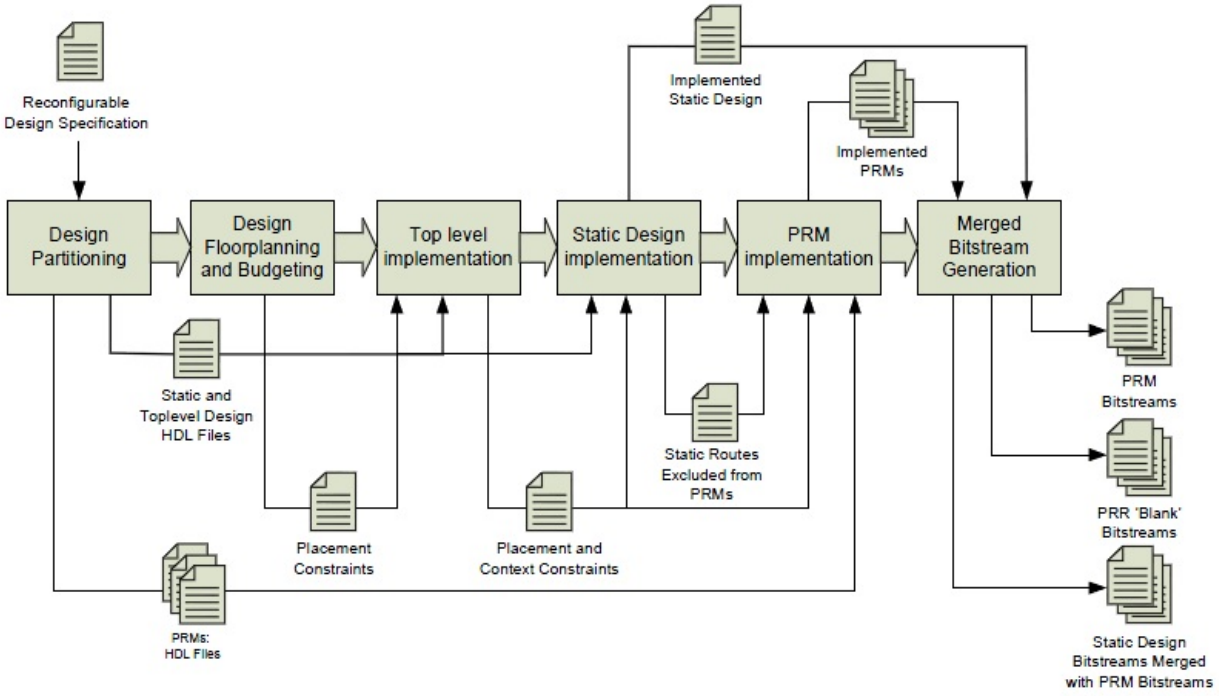


**Figure 2.** Partial Reconfiguration flow as suggested by Xilinx

In [9] McCaskill 2010, provided information about the evolution of partial reconfiguration and why it's easier today for teams to use this new technology considering the tools and support. McCaskill also suggested to follow six simple steps when using partial reconfiguration. In particular, a "bottom up" synthesis flow should be used. According to which each partition will form a separate synthesis project with its own netlist. The steps for a design using PR are

1. Set up the design structure, deciding on static versus reconfigurable logic. Synthesize all netlists, define RMs and create partitions. Then, assign RM netlists to the appropriate RPs.
2. Provide the proper constraints for each RP based on the assigned RMs.
3. Run the PR-specific design rule checks that are in PlanAhead.

4. Place and Route all design combinations. Create a 'golden reference' for the static logic and iterate to close all timing on all the combinations of static logic and RMs.
5. Create the bit files.
6. Test the design.

## 2.2 Reconfiguration Overhead

In [4] Papademetriou *et al.* 2010 describe a framework to evaluate dynamic partial reconfiguration of an application using a Virtex II PRO FPGA. The reconfiguration overhead was broken down into the following times to allow better understanding.

1) CF-PPC is the time to copy configuration data from the CF to the PPC memory with one transaction.
2) PPC-HWICAP is the time to write configuration data from the PPC memory to the HWICAP BRAM.
3) HWICAP BRAM-CM is the time to load the configuration data from one HWICAP BRAM to the FPGA CM.
4) Rec-HWICAP is the time elapsed between the PPC detection that a reconfiguration has been fired and the first launch of the configuration data from the HWICAP BRAM to the FPGA CM.
5) HWICAP-CM is the time for loading *all* configuration data from the HWICAP BRAM to the FPGA CM, including the pad frame.
6) RT is the time elapsed between the PPC detection that a reconfiguration has been fired and switching to the new execution, this is the total reconfiguration time.

The results showed that in most cases the delay, linearly increased with respect to the size of the bitstream with a fixed processor array. In general, depending on the size of the partial bitstreams, the selection of the system parameters might improve or degrade the performance.

The traditional way of accessing the ICAP was either through the OPB (OPBHWICAP) or PLB (IPIF) bus. The use of these busses though, takes up a large amount of resources; their use is complex and was designed to work with particular bus systems in each case, therefore not allowing them to be reused. In [7], Victor Lai and Oliver Diessel introduced a new interface for accessing the ICAP, the ICAP-I. The interface implements a wrapper that provides a new easy to use interface for accessing the ICAP

without the need of a bus, or a processor core but can still be used with other bus systems such as the OPB and PLB. The ICAP-I is a set of VHDL modules. The storage device provides configuration data for the ICAP and stores configuration data that is read from the ICAP while allowing other devices to use the storage device. The ICAP IF module accepts the read and write requests from the storage device and the application. The ICAP control arbiter controls the requests from the application and the BTU allows access to the ICAP. The storage device provides pre-generated bitstreams to the ICAP-I, where as the application provides on the fly bitstreams, generated during run time. Both bitstreams are loaded to the ICAP while the application is running. One of the main problems with the ICAP-I was its inefficient way of transferring data. Finally, a comparison between the performance of the ICAP-I , the OPB and the PLB based ICAP implementations is provided using a Virtex 4 FPGA board. The results showed that the ICAP-I uses a lot less resources, which is due to not using CPU cores. The ICAP-I also achieved a much higher throughput to the ICAP, 180 MB/s compared to the 95 MB/s achieved by the PLB. The only problem with the ICAP-I implementation is the storage device used to store bitstreams; this was the bottle neck of the setup. The device used, impacted the performance of the ICAP-I, the slower the device the less throughput achieved.

In [8] Shaoshan Liu *et al.* 2009 proposed two techniques to minimize the reconfiguration overhead. To improve the reconfiguration speed, a method using streaming DMA engines (direct memory access) to transfer the configuration data directly to the ICAP was proposed. A master DMA engine was added in the ICAP controller and communicates with the ICAP FSM and the slave DMA engine. The slave DMA engine was placed in the SRAM controller and communicates with the SRAM bridge and the master DMA engine. A FIFO was placed between the master and slave DMA engines to increase the throughput. The burst mode of the SRAM was also activated. The setup proposed can be seen in *Figure 3*. The FPGA board used was a Virtex 4. This method achieved an ICAP throughput of 395 Mbytes/s very close to the ideal 400 Mbytes/s. This design improved the simple use of DMAs proposed in [10], which achieved a speed of only 82 Mbytes/sec. The reason for not achieving maximum throughput was due to the master slave handshaking. This process took 12 cycles to complete, after the 12 cycles the FIFO is no longer empty. After the DMA operation is complete, it takes another 5 cycles to reset the two state machines. Therefore, there is a total overhead of 17 cycles. During the rest of the operation the throughput was 400 Mbytes/s.

The second way to reduce configuration time was by decreasing the configuration file size with compression techniques. To achieve this, they tried to find words that repeated themselves. The main advantage of this method was that it was simple and minimized the overhead of the decompression

circuit. When the ICAP receives a word it determines if decompression is necessary, if so, the ICAP controller performs decompression and then sends the configuration data to the ICAP port. The intelligent ICAP controller adds only a very little hardware overhead. Other methods to improve configuration performance include prefetching the configuration bitstream files and bitstream file relocation. After performing tests on the Intelligent ICAP controller the results showed that, by combining the DMA engines with the intelligent ICAP controller, they achieved an effective data transfer throughput of 1.2 GBytes/s in some cases, that well surpassed the upper bound of data transfer throughput of 400 Mbytes/s.

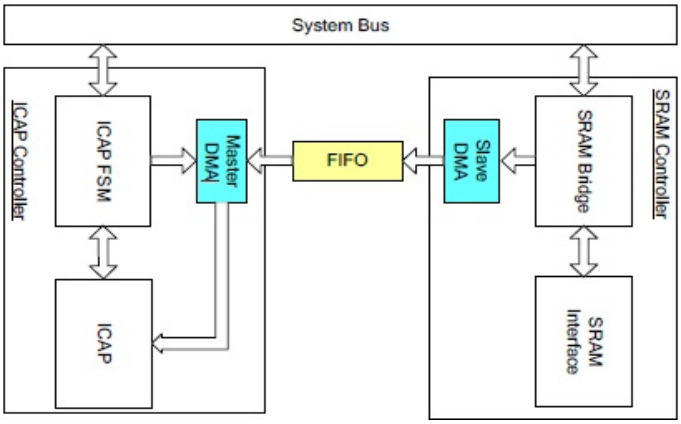


**Figure 3**. Structure of the master-slave DMA for PR

In [10] Ming Liu *et al.* evaluated nine different setups for their reconfiguration speeds on a Virtex-4 FPGA. Results showed that using a Master burst or a Direct Memory Access we can achieve the best times compared to the resources used (234.5 MB/s for MST and 82.1 MB/s for DMA). Use of BRAMS can approach the reconfiguration speed limit of the ICAP at the cost of large Block RAM utilization (332.1 MB/s). PLB and OBP setups were also studied but the reconfiguration times were much slower than the three setups above.

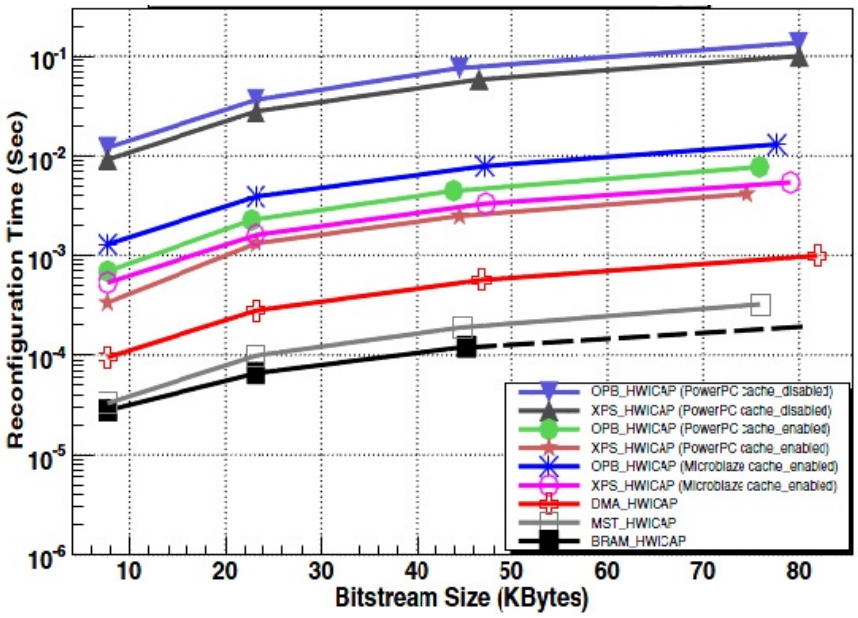


**Figure 4.** Reconfiguration times for different setups

In [5] Papadimitriou *et al. 2011* produced a cost model of partial reconfiguration and surveyed the performance of the factors that contribute to the reconfiguration speed. They concentrated on the three following factors of partially reconfigurable systems, the external storage, the configuration controller and last, the phases of the reconfiguration process and the formulas used to calculate the reconfiguration times. It is explained, how the external memory used to store the bitsteams, plays a major factor in the reconfiguration overhead, the slower the memory the bigger the overhead becomes. To overcome this problem, the bitstreams can be loaded to a high speed memory after the systems boots, to achieve faster reconfiguration. Concerning the reconfiguration controller, many solutions have been proposed, depending on the application. Customized controllers aim to speed up the process and allow the processor to do other tasks, in this case a DMA (direct memory access) controller is used. When the processor acts as the reconfiguration controller, the system suffers from long delays due to the time needed to access the memory, to call and execute the software instructions. The number of modules on the bus also plays a role on the delay, the more the devices the longer the delay. They then split the reconfiguration process into three different phases, i) Phase to pull the bitstream from the off chip memory to the memory of the processor, ii) Copy it from the from the processor memory to the ICAP cache, iii) Send it from the ICAP cache to the configuration memory of the FPGA. It has been shown that the largest overhead occurs from the first two phases since the third phase has a standard time of execution. A comparison between the bandwidth of the configuration port and the actual

reconfiguration throughput is presented, using different ports, controllers and external memories. In most cases the Actual Reconfiguration Throughput is much less than the bandwidth given for the configuration port. Results showed using a faster memory improves the ARTH dramatically even when an older OPB controller was used. In some cases the throughput was larger than that of a PLB controller.

## 2.3 Hiding Reconfiguration Overhead

In [11] Papademetriou and Dollas evaluated a preloading model to hide configuration overhead. The problem studied, was the delay to load a second configuration into the reconfiguration region when it did not fit in the region, simultaneously with the first. To solve this problem the bitstream that was most likely to be executed was preloaded and the second was transformed. It was split into two subtasks so that one portion fits on the remaining hardware. Therefore, if the second task was required to be executed, only the second subtask needs to be loaded. For the experiments a Virtex-II XC2V500 devise was used. Results showed that as the volume of the CLB columns that can be utilized for preloading the least likely to be executed operation increased, the execution length of the augmented model decreased compared to the original model. The largest improvement obtained was for seven available CLBs and was equal to 86.55%. The results showed the relationship between configuration latency and reconfiguration overhead and whether reconfiguration can be hidden by the processor's execution. The main advantage of the proposed model is the increase in the utilization of the available hardware achieved by splitting the least likely to be executed task.

In [13] Liu *et al.* introduced a new concept of Virtual Configurations for shortening the FPGA reconfiguration time by hiding it in the background. The solution they gave was to have two copies of configuration contexts; these represented a VCF and were located on a single Partial Reconfiguration Region. The active VCF could still keep working in the foreground when module switching was expected. When a new partial bitstream was needed, it was loaded into the second context. Once the reconfiguration was done, the newly loaded module started working by being swapped with the first context. This can be seen in *Figure 5* for switching the two contexts a multiplexer was used to only switch the control outputs; therefore the context swapping only took a very short time. To see the impact of the VCFs they set up a producer consumer design with run time reconfiguration capability. Results showed up to 29.9% throughput improvement of received packets by each consumer node. The

use VCFs is better suited for multi context FPGAs, since single context FPGAs, would require the reservation of two duplicate PRRs.

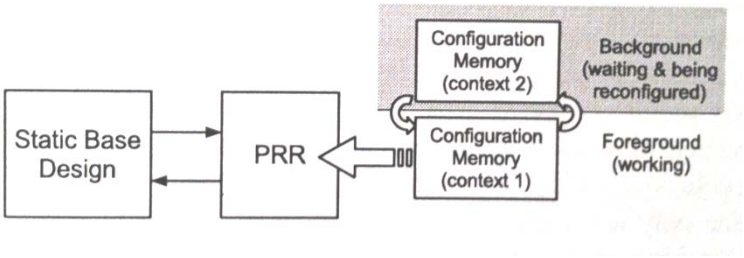


**Figure 5.** Virtual reconfigurations on multi-context FPGAs

## 2.4 Applications Designed in PR Systems

In [17] Nikoloudakis researched the advantages and disadvantages using reconfigurable logic with encryption applications such as the Advanced Encryption Standard (AES). He also created an embedded system which allowed different algorithms for cryptography, to be partially reconfigured on a FPGA board.

# Chapter 3

# Partial Reconfiguration Tools and FPGAs

In this chapter we will be introduced to the operating system, the Xilinx software and the FPGA board that was used throughout the process of the thesis.

## 3.1 Operating System

During this thesis it was decided to use a Linux based operating system. For better support and easier installation Xilinx, suggested the use of SUSE. Therefore, a 32-bit version of SUSE Enterprise Desktop Edition 11.1 was installed on a computer. Even though it was a 60 day trial version (did not receive updates after this period), this did not cause any problems. The software was downloaded from the Novell website, which can be found at www.novell.com/products/desktop/. The installation was straightforward and no problems occurred at this point.

## 3.2 Partial Reconfiguration Tools

The Xilinx design tools used were the ISE Design Suite 12.3, which was downloaded from the Xilinx website (www.xilinx.com/products/design-tools/ise-design-suite/system-edition.htm). The installation of the Xilinx tools consists of two steps. First, installing the software and second installing the platform cable driver for downloading the designs to the board.

The installation of the Xilinx software was quite simple and straightforward with the help provided in [14]. Unlike Windows, to run the Xilinx software, a script was written for this task, which can be found in the appendix.

On the other hand, the installation of the cable driver caused quite a few problems. Although Xilinx recommends SUSE, and insures a simple and straightforward installation, our first attempt to use the drivers provided by Xilinx, did not work. Therefore a different method needed to be used to install the software drivers. The solution came from a library found in [17]. The libsub library that was installed, allows the tools to access the JTAG cable without the need for a proprietary kernel module. For further information details on installing the drivers can be found in the appendix.

Therefore, due to the above difficulties there is no reason for the use of Suse, even though it is recommended by Xilinx. The same library can be used with the Ubuntu operating system and the supplied driver works with Windows.

Unlike Windows, Suse had no pre-installed terminal program (like HyperTerminal) to display the output from the serial cable. After some searching, it was decided to use CuteCom. CuteCom is a graphical serial terminal and is more user friendly compared to minicom, which is also a terminal program. Information about the installation can be found in the appendix.

## 3.3 Virtex 5 VLX-110t FPGA Board

The FPGA board used during this thesis was a Virtex 5 VLX-110t. It included all the necessary components for dynamic reconfiguration. The basic element is the configurable logic block (CLB), which contains look-up tables (LUT) as the basic function generators. The FPGA also contains several specialized circuits such as Block SelectRAM (BRAM) resources, multiplier blocks and Digital Clock Manager (DCM) modules. The FPGA has a total of 17,280 slices; each slice contains four LUTs and four flip-flops. Compared to older Virtex families the PowerPC and the OPB bus have been removed and are no longer used.

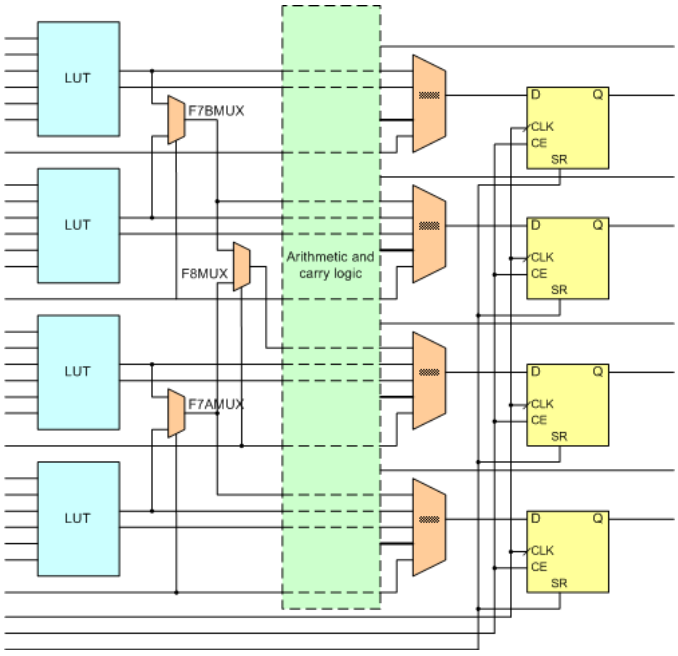


**Figure 6.** Virtex-5 slice

The serial JTAG, the SPI and the parallel SelectMAP allow for external configuration, where as the parallel Internal Configuration Access Port (ICAP) allows for internal, partial only, configuration. The ICAP on the Virtex-5 can support up to 32-bit transfers. The maximum operational frequency of the ICAP is 100 MHz. The block level diagram for the ICAP controller can be seen in *Figure 7*.

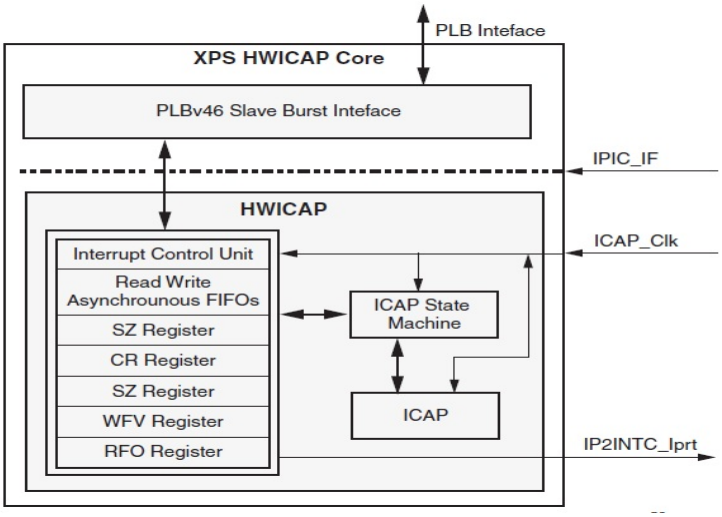


**Figure 7.** Top level block diagram for the XPS HWICAP Core

The XPS HWICAP controller provides the interface necessary to transfer bitstreams to and from the ICAP. The CPU bursts the required bitstream data directly from the main memory. Incoming data is stored within a Write FIFO, from where it can be fed to the ICAP. All the bitstreams must be stored in main memory before they can be used to reconfigure the FPGA.

The configuration data is stored in RAM memory called configuration memory. Configuration memory is arranged in frames that are tiled about the device. A frame is the smallest amount of configuration information that can be accessed and has a width of one bit. All operations must act upon whole configuration frames. On the newer Virtex 4 and 5 FPGA boards the height of the frames is 1,312 bits, compared to the older boards were the height varied depending on the devise. The frames stretch from the top edge of the device to the bottom edge. The devise has a total of 24,304 frames, from which 23,712 of those frames are configuration frames. *Figure 8* displays the configuration architecture of the Virtex 5 FPGA.

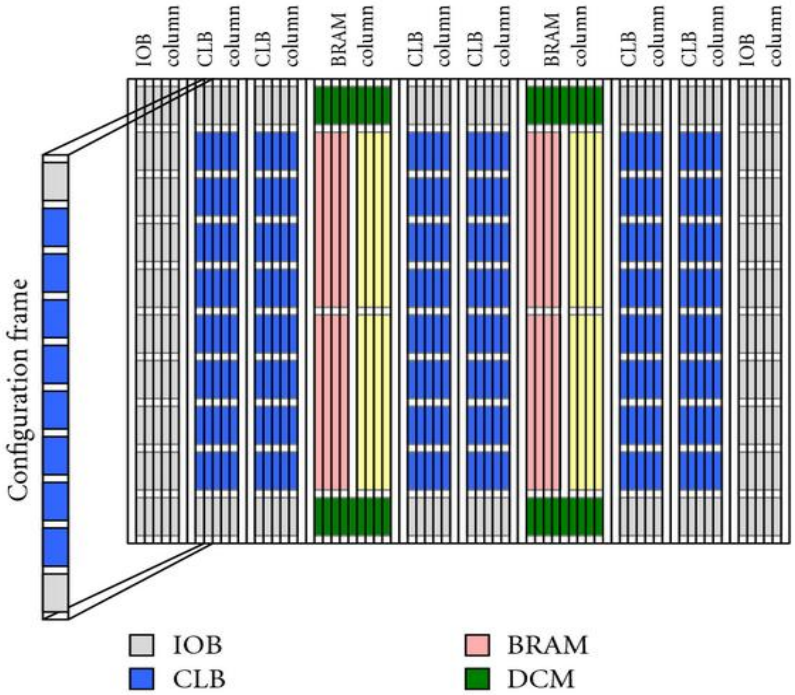


**Figure 8.** Xilinx Virtex configuration architecture

| CLBs | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Slices | Array (row x col) | Max Distributed Ram(kB) | Total CLBs | Slice/CLB | Total config. bits | Config. Frames | Frame size (bit) | DSP48E Slices | Block Ram (36kB) |
| 17,280 | 160x54 | 1,120 | 8,640 | 2 | 31,118,848 | 23,712 | 1,312 | 64 | 148 |

**Table 1.** Virtex 5 VLX110t Device Features

In *Table 1* we have listed the sizes of all the features included on the VLX110t. Each CLB contains two slices. The total memory used by the BRAMS is 5328 kB; the designer has the choice of splitting the 36 kB block ram into two smaller 18kB block rams, therefore having a total of 296 BRAMs. Each DSP48E slice contains a 25 x 18 multiplier, an adder, and an accumulator.

Finally, each block type (CLB, DSP48E, etc.), contains a different number of frames. The approximate number of configuration bits for each devise feature, along with the frames per slice can been seen in *Table 2*.

| Device Feature | Approximate Number of Configuration Bits | Frames per Slice |
|---|---|---|
| 1 Logic slice | 1,181 | 0.9 |
| 1 Block Ram (36kb) | 1,170 | 0.8917 |
| 1 Block Ram (18kb) | 585 | 0.446 |
| 1 I/O block | 2,657 | 2.025 |
| 1 DSP48E slice | 4,592 | 3.5 |

**Table 2.** Number of Configuration Bit for Different Block Types

## 3.4 Partial Reconfiguration

As seen earlier, partial reconfiguration allows designers to reconfigure selected areas of a FPGA board. The swapping of the reconfigurable modules can be seen in *Figure 8.* Both partial and static regions are pre-defined and cannot be changed during run-time. This means that the user must make sure that all the designs will fit into the partial reconfiguration region.

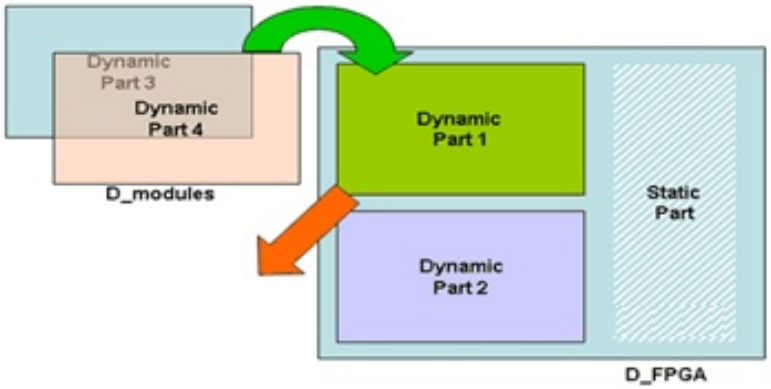


**Figure 9.** Swapping modules in and out of PRR

Partial reconfiguration can be divided into two separate categories, dynamic and static. Dynamic reconfiguration takes place while the device is running, while static reconfiguration is performed when the device is inactive. An extension of the dynamic concept is self-reconfiguration. It assumes that special circuits on the logic array are used to control the reconfiguration of other parts of the FPGA. When using dynamic reconfiguration, the part of the FPGA that is undergoing reconfiguration is also known as Partial Reconfiguration Region (PPR). This region is static and cannot be changed. The modules being swapped in and out of these regions are known as Partial Reconfigurable Modules (PRM) and are stored as partial bitstreams in a memory. This memory may be external such as a compact flash or on board the FGPA, for example BRAMs. The memory used for storing this data plays a major factor on the reconfiguration time.

Communication between the modules in the design is achieved by using special bus macros. These bus macros are fixed data paths for signals going between a reconfigurable module and another module. The bus macro can be wired so that signals can go in either direction (left-to-right or right-to-left) and is strongly recommended that once a direction is defined, it should not change for that particular FPGA design. During the reconfiguration process it is very important that communication between the modules is stopped due to the unknown state of the signals.

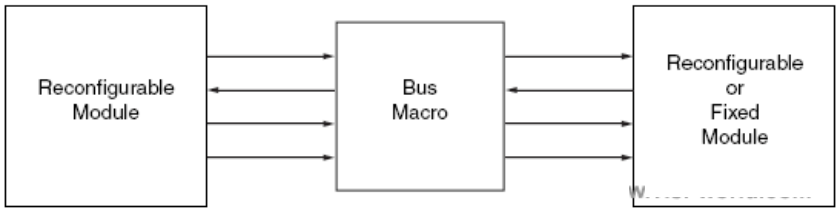


**Figure 10.** Use of bus macros for communication between PRR and static region

# Chapter 4

# Evaluation Framework

In this chapter we will see a series of tests that were performed on the Virtex-5 vlx110t board. The tests provided us with measurements concerning time requirements to perform different actions such as communication over the serial port and access of a BRAM.

## 4.1 Serial Port

The purpose of this test was to determine the total clock cycles required to send data from the FPGA to the computer over the RS 232 serial port. For the communication between the PC and the FPGA board, a simple design was setup using the EDK 12.3 software. The design parameters were the following, a microblaze processor, the system clock frequency was set to 125 MHz and the local memory had a size of 8 KB. The only peripheral attached to the design was a RS 232 port, with a baud rate of 9600.

In order to measure the clock cycles between actions, the XPS timer was used. This is provided by the EDK tools and sits on the PLB bus of the design. Using software we were able to start and stop the timer when desired.

For the testing, we sent different messages over the serial port. We started by sending each character, in the phase "Hello World" one at a time over the serial port. The results for the cycles needed to transfer each group of characters can be seen in *Table 1*. The first column shows the data that was sent over the RS232 port, the second column are the cycles needed to transfer the data and in the third column we have the increase in cycles compared to the pervious data that was sent. In *Figure 11, a* graph is presented which shows the increase of cycles needed compared to the characters being sent.

| Text sent in printf() command | Time elapsed (cycles) | Increase |
|---|---|---|
| H | 1583 | 0 |
| He | 260209 | 258626 |
| Hel | 390284 | 130075 |
| Hell | 520364 | 130080 |
| Hello | 650439 | 130075 |
| Hello | 780514 | 130075 |
| Hello W | 910609 | 130095 |
| Hello Wo | 1040697 | 130088 |
| Hello Wor | 1170772 | 130075 |
| Hello Worl | 1300847 | 130075 |
| Hello World | 1430942 | 130095 |
| Hello World! | 1561010 | 130068 |
| Hello World!! | 1691085 | 130075 |
| Hello World!!! | 1821180 | 130095 |

**Table 3.** Serial Port testing results

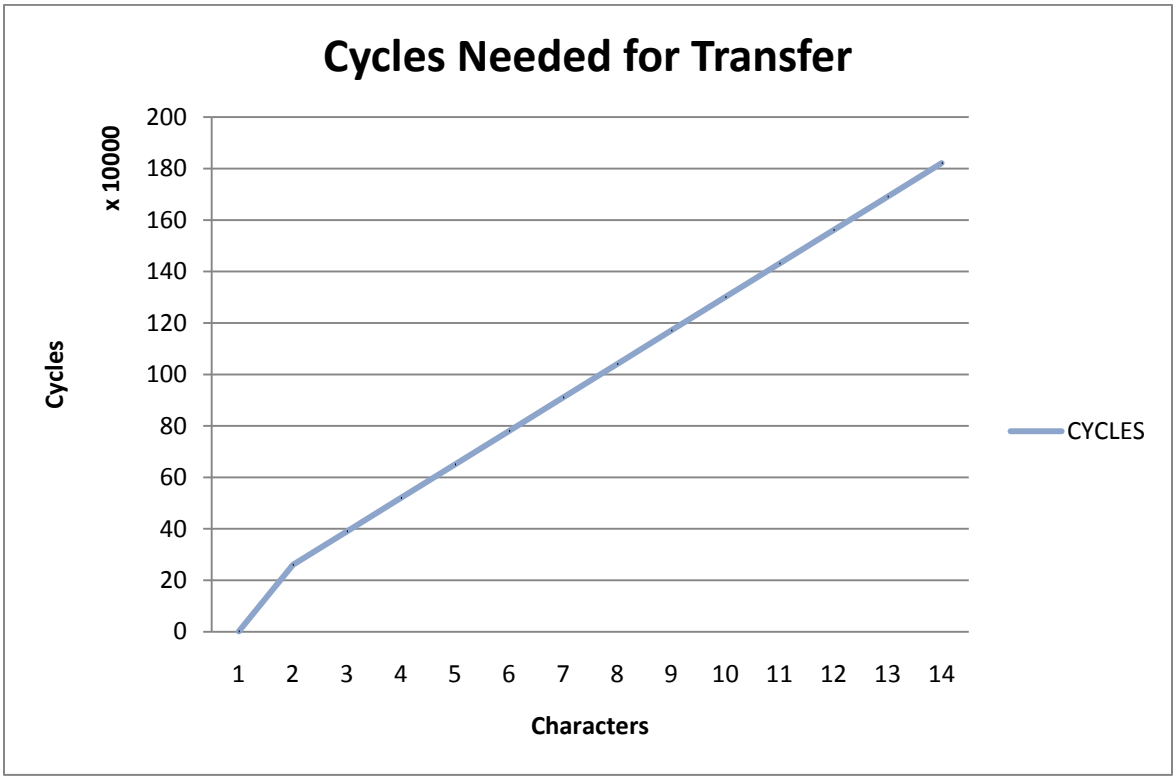


**Figure 11.** Cycles required to transfer data over the RS232 port

Results showed that the serial port needs a total of 1.583 cycles to send one character but when increasing the number of characters to two, there was a dramatic increase of 258626 cycles. After this

point and for every other character we saw a liner increase of required cycles. The slow communication speed achieved was also due to the low baud rate of 9600bps that was used. A possible increase in the rate would most lucky show us a decrease in the required cycles.

A part of the code used for the above testing can be seen below

```
XTmrCtr XPS_Timer; //name the timer
XTmrCtr_Initialize(&XPS_Timer, XPAR_XPS_TIMER_0_DEVICE_ID);
XTmrCtr_SetResetValue(&XPS_Timer, XPAR_XPS_TIMER_0_DEVICE_ID, 0);
XTmrCtr_Reset(&XPS_Timer, 0);
XTmrCtr_Start(&XPS_Timer, 0); //start timer
printf("Hello world!!!");
XTmrCtr_Stop(&XPS_Timer, 0); //stop timer
long int cycles;
cycles = XTmrCtr_GetValue(&XPS_Timer, 0); //get number of cycles
xil_printf("\ncycles: %d\r\n",cycles);
```

As can be seen in the above code, before the XPS_Timer can be used, it must be declared and initialized. These actions can be seen in line 1 and 2 of the above code.

## 4.2 BRAM access

The purpose of this test was to determine the total clock cycles required to read and write from a BRAM memory. Once again, a design was created in EDK using the same parameters used to test the serial port. In this case though, we added a BRAM memory and controller in order to test.

For the first series of tests, we wrote 1 Byte of data to an offset of the memory address and timed this function. Next, we read back the same data and also timed the function (Figure 12). For the second test, we sent 4 Bytes to an address of the BRAM and then read it back (Figure 13). The third test consisted of writing one Byte at a time to the BRAM and also reading back one byte at a time (Figure 14). Finally, we placed the read and write command in a for-loop and wrote and read 1 byte for each loop (FIGURE). The cycles measured during the experiments are shown in *Table 2*.

```
XTmrCtr XPS_Timer; //name the timer
XTmrCtr_Initialize(&XPS_Timer, XPAR_XPS_TIMER_0_DEVICE_ID);
XTmrCtr_SetResetValue(&XPS_Timer,XPAR_XPS_TIMER_0_DEVICE_ID, 0);
XTmrCtr_Reset(&XPS_Timer, 0); //reset timer
Xuint32 data;
XTmrCtr_Start(&XPS_Timer, 0); //start timer
XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR,
0x00000000000000000000000000001131);
XTmrCtr_Stop(&XPS_Timer, 0); //stop timer

long int cycles;
cycles = XTmrCtr_GetValue(&XPS_Timer, 0);
xil_printf("write cycles: %d\r\n",cycles);

XTmrCtr_Reset(&XPS_Timer, 0);
XTmrCtr_Start(&XPS_Timer, 0);
data = XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR); //read data
XTmrCtr_Stop(&XPS_Timer, 0);
xil_printf("Read from BRAM : 0x%X\r\n",data);
long int cycles2;
cycles2 = XTmrCtr_GetValue(&XPS_Timer, 0);
xil_printf("read cycles: %d\r\n",cycles2);
```

**Figure 12.** Code for testing BRAM with 32bit input

```
XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+0*32,0x00000031); //writes offset
XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+4*32,0x00000032);
XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+8*32,0x00000033);
XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+12*32,0x00000034);

data = XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+0*32); //reads offset
data2 = XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+4*32);
data3 = XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+8*32);
data4 = XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+12*32);
```

**Figure 13.** Code to write and read from different offsets

```
input=0x00000030;
int i;
XTmrCtr_Start(&XPS_Timer, 0);
for (i=0; i<13; i=i+4)
      {
            XIo_Out32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+i*32,input);
            input=input+1;
      }

int tmp[4];
for (i=0; i<13; i=i+4)
      {
            data=XIo_In32(XPAR_XPS_BRAM_IF_CNTLR_0_BASEADDR+i*32);
            tmp[i+1]=data;
      }
```

**Figure 14.** Code to write and read using for-loop

| Test | Cycles without I/D cache | | Cycles with I/D cache | |
|---|---|---|---|---|
| | **Write** | **Read** | **Write** | **Read** |
| 1 Byte | 66 | 70 | 61 | 63 |
| 4 Bytes sent with one command | 66 | 70 | 61 | 63 |
| 4 Bytes using four commands to access each offset | 93 | 122 | 82 | 87 |
| Using for-loop to access each offset | 198 | 222 | | |

**Table 4.** BRAM testing results

The results of the tests showed that, the time to write or read one byte or 4 bytes is the same if using one command. When using four commands, to either write or read each offset of the address, we observed an increase of 27 cycles to write and 52 cycles to read from the BRAM compared to writing all the bytes to the memory at once. This could be due to the number of commands used in the program which are now four compared to when using one to access the memory. When adding a "for loop" to read and write we observed an even larger increase in the cycles required. When writing 105 cycles more were needed and when reading 100 more cycles, compared to when using four separate commands to access our BRAM. The increase we saw was due to the for-loop, which requires a number of cycles to cheek the statement and decide whether or not to execute the loop. In both cases more cycles where required to read from a BRAM than to write.

Using the same design we enabled the I/D cache to see how much of an effect it would have on accessing the BRAM. Two different types of memories were used as cache, a DDR2 and a SRAM, each 4K. The results although, were the same for both memories. We executed the same test as above; the results can be seen in *Table 2*. Once again, when we used one command to write or read either 1 byte or 4 bytes, the same number of cycles were required. When using more than one command to access we also saw an increase in the cycles required.

Comparing the two designs (with and without cache), we saw that there was a small decrease when using one command, 5 cycles when writing and 7 when reading. When accessing four different offsets, there was a decrease of 11 cycles to write and 35 cycles to read, this is where the cache had the largest effect.

# Chapter 5

# Benchmarking

## 5.1 Algorithms for Cryptography

To evaluate partial reconfiguration on the Virtex-5, it was decided to use algorithms used for cryptography. The algorithms used were the following, the Advanced Encryption Standard (AES) and Blowfish. For both algorithms we only used the encryption mode to evaluate reconfiguration on the FPGA. Encryption is the process of converting information (which is called plaintext) into ciphertext, which is not understood to a third person. To convert the plain text to a ciphertext a key is used, which is known only to the communicants. Cryptography is mainly used to transfer data in a secure manner and ensure that it is not tampered with by intruders.

### 5.1.1 The Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric-key encryption standard. It was developed in order to substitute the older Data Encryption Standard. The standard comprises three block ciphers, AES-128, AES-192 and AES-256 and are known as the Rijndael algorithms. Each of these ciphers has a 128-bit block size, with key sizes of 128, 192 and 256 bits respectively. The algorithm used by us was the AES-128.

AES is based on a design principle known as a substitution permutation network; it is fast on both hardware and software. The AES cipher is specified as a number of repetitions of transformation rounds, that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. The number of rounds depends on the block and key size. In our case we had a total of nine rounds. AES operates on a 4x4 matrix of bytes which is known as the state. Each block contains 1 Byte and the table has a total of 16 Bytes (128 bits) of data.

The Algorithm consists of the following steps:

1. Key Expansion: round keys are derived from the cipher key using Rijndael's key schedule.
2. Initial Round
    2.1. Add Round Key: each byte of the state is combined with the round key using bitwise xor
3. Round
    3.1. Sub Bytes: a non-linear substitution step where each byte is replaced with another according to a look up table
    3.2. Shift Rows: a transposition step where each row of the state is shifted cyclically a certain number of steps.
    3.3. Mix Columns: a mixing operation which operates on the columns of the state, combining the four bytes in each column.
    3.4. Add Round Key
4. Final Round
    4.1. Sub Bytes
    4.2. Shift Rows
    4.3. Add Round Key

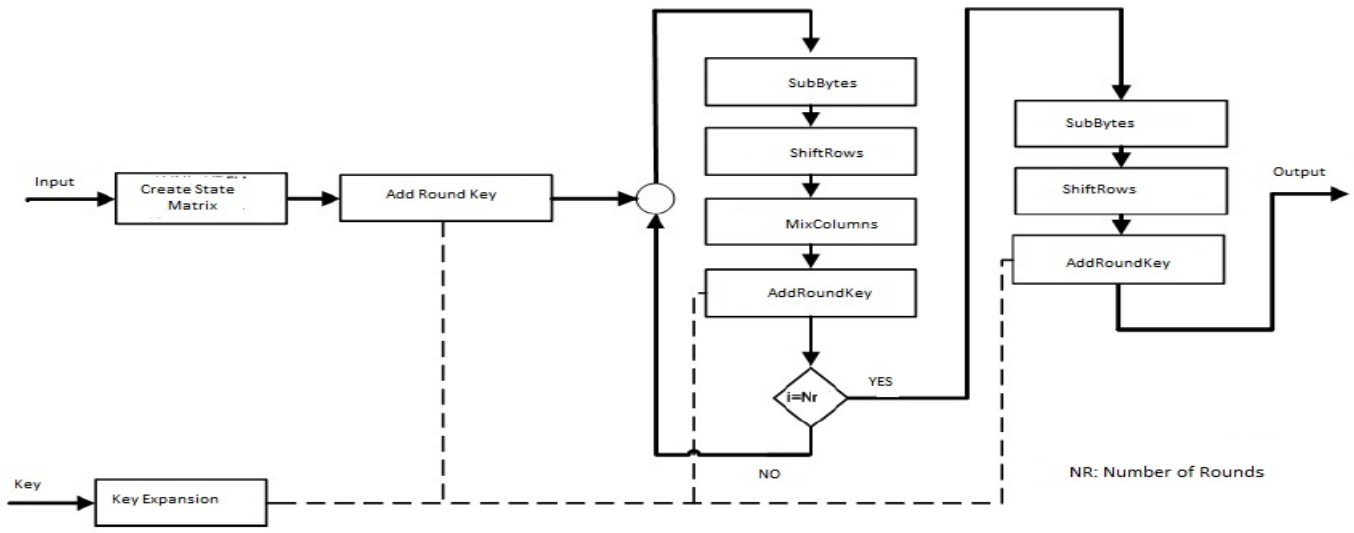


**Figure 15.** AES algorithm

In the SubBytes step, each byte in the matrix is updated using an 8-bit substitution box know as the Rijndael S-box. This step can be seen in *Figure 16.*

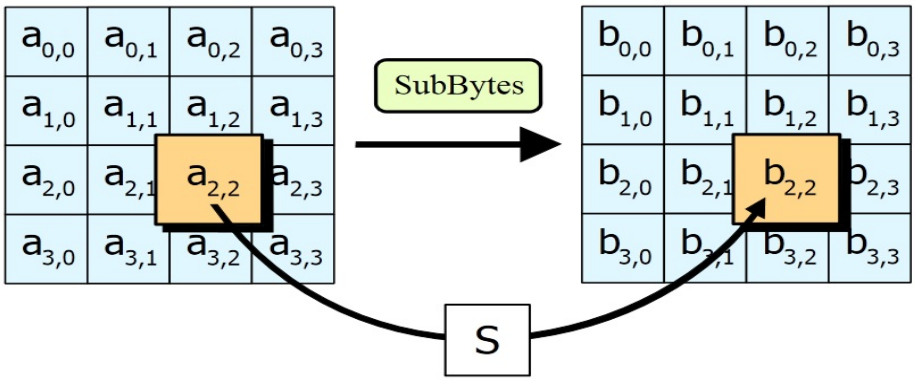


**Figure 16.** SubBytes

In the Shift Rows step, each row is shifted to the left by a certain offset as see in *Figure 17*. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively.

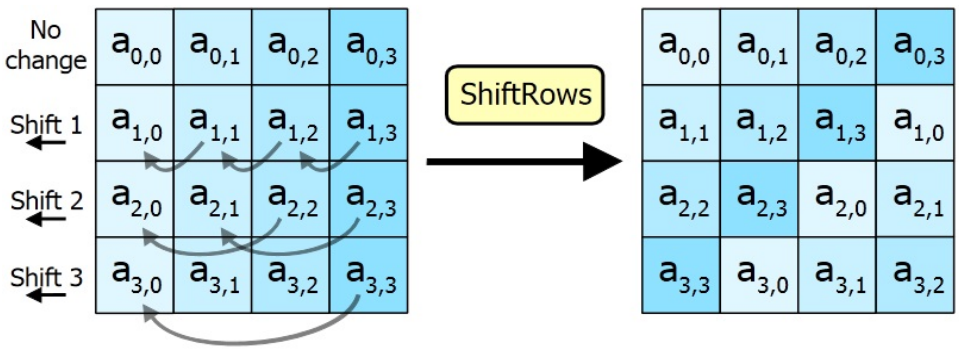


**Figure 17.** ShiftRows

In the Mix Columns step, each column of the state is multiplied with a fixed polynomial c(x) as seen in *Figure 18*.

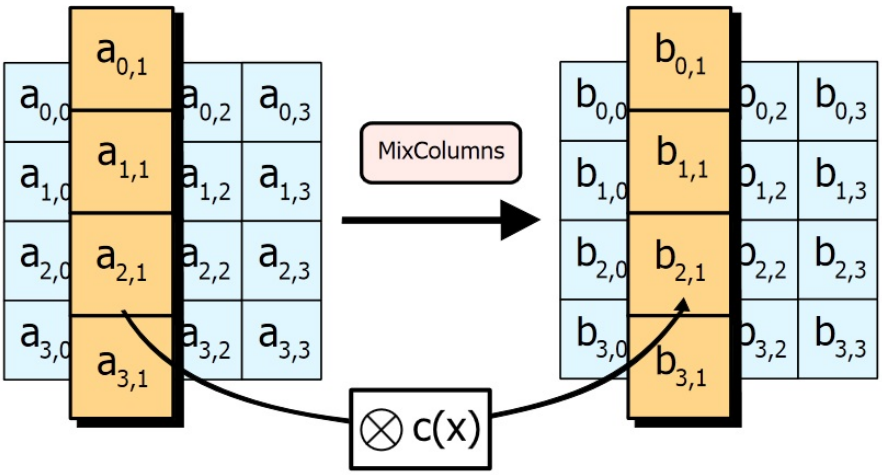


**Figure 18.** MixColumns

In the Add Round Key step, the sub key is combined with the state, as seen in *Figure 19*. For each round, a subkey is derived from the main key using Rijndael's key schedule. Each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

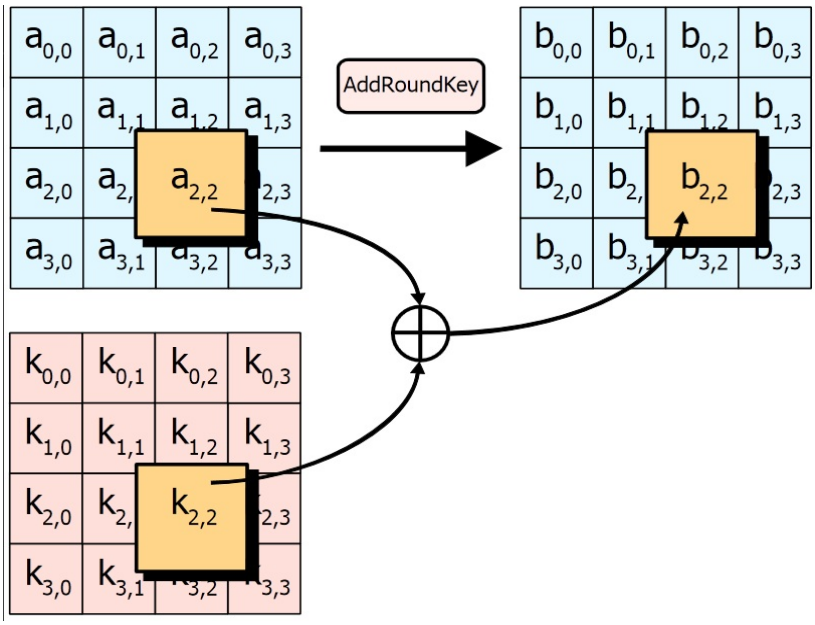


**Figure 19.** AddRoundKey

## 5.1.2 Blowfish

Blowfish is a keyed, symmetric block cipher. It has a block size of 64 bits and its key can be anywhere between 1 and 448 bits. In our study, the algorithm used had a 16-bit input and a key size of 32-bits. The algorithm consists of two parts: a key-expansion part and a data- encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes.

Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key - and data - dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round. The encryption of data is very efficient despite the fact that there is a complex initialization phase required before any encryption takes place.

Blowfish uses a large number of sub-keys. These keys must be precomputed before any data encryption or decryption.

- The P-array consists of 18 32-bit sub-keys:

  P1, P2, P3, P4...., P18.

- There are four 32-bit S-boxes (see *Figure 20*) with 256 entries each

S1,0, S1,1,..., S1,255;

S2,0, S2,1,..., S2,255;

S3,0, S3,1,..., S3,255;

S4,0, S4,1,..., S4,255.

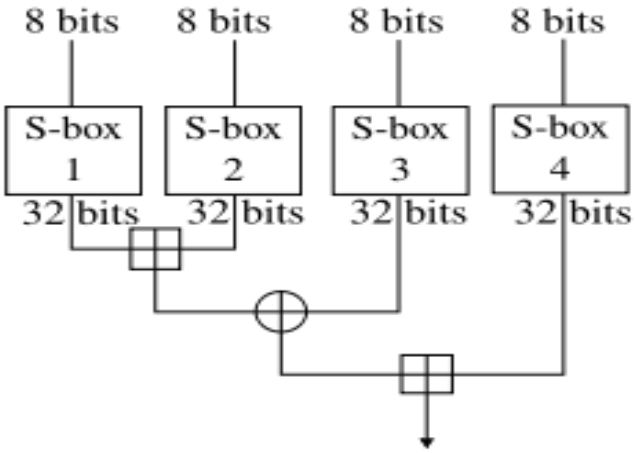


**Figure 20.** Blowfish S-boxes

Function F, the non-reversible function, gives Blowfish the best possible avalanche effect for a Feistel network: every text bit on the left half of the round affects every text bit on the right half. Additionally, since every sub-key bit is affected by every key bit, the function also has a perfect avalanche effect between the key and the right half of the text after every round. Hence, the algorithm exhibits a perfect avalanche effect after three rounds as well as every two rounds after that. The use of the F function and the complete data flow of the Blowfish cipher can be seen in *Figure 21*.

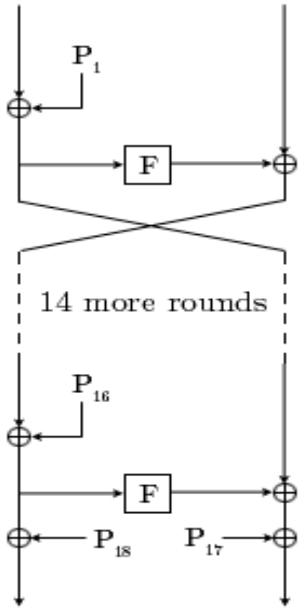


**Figure 21.** Data flow of Blowfish block cipher

## 5.2 Application Resource Analysis

In order to calculate the theoretical Partial Reconfiguration Time using the PRCC website, an analysis of the recourses used by each algorithm had to be done. By using EDK we were able to estimate these resources and calculate the partial bitstream size, depending on the frames used by our design. The flow of our experiments can be seen in *Figure 22*. By using the PRCC tool, we are given the opportunity in deciding whether the use of partial reconfiguration suits our design or not. Although, this tool has been made for the Power PC processor, it can provide reasonable results when using the Microblaze processor. Finally, we can test different setups for our design in order to choose the one that provides the fastest execution. All this can be done by changing only the setting on the PRCC and not creating a new project for each case.

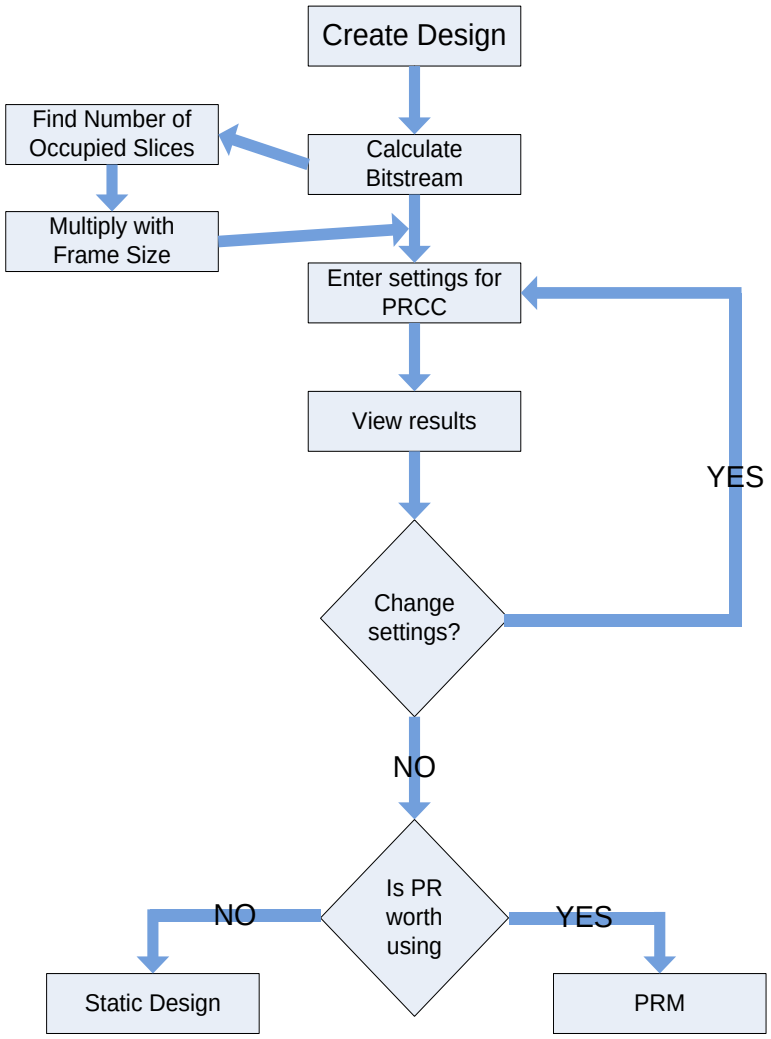


**Figure 22.** Experimental Flow

## 5.2.1 Advanced Encryption Standard 128 PLB Bus Analysis

The AES algorithm downloaded to our FPGA board works with the following steps:

1. Design downloads to board and awaits for data from PC
2. User enter a 128 bit key using a terminal program from the PC
3. User enters data with a width of 128 bits or a text file
4. Microblaze returns a 128bit result to PC and user can see encrypted data
5. User may enter more data with a width of 128bits, using the same key

The resources consumed by the AES algorithm can be seen in *Table 5.* The second column shows the total resources used by the complete system created in EDK, which contains the Microblaze processor, the AES peripheral and the RS232 port, used for communication with the PC. Therefore, these numbers were not accurate in order to calculate the bitstream size of the AES algorithm. The third column shows the resources used only by the AES algorithm, as calculated from the design in ISE 12.3. In the fourth column we can see the total number of configuration bits used by the algorithm. This was calculated by using the configuration bits used by each device feature, which can be found in *Table 2*.

| Device Feature | Total Resources Used in EDK 12.3 | Resources Used in ISE 12.3 | Configuration Bits Used |
|---|---|---|---|
| **Slice Registers** | 2748 out of 69120 (3%) | 676 out of 69120 (1%) | - |
| **Slice LUTs** | 3655 out of 69120 (5%) | 1424 out of 69120 (2%) | - |
| **Occupied Slices** | 1628 out of 17280 (9%) | 558 out of 17280 (3%) | 658998 |
| **Block RAM (36k)** | 8 out of 148 (5%) | 0 out of 148 (0%) | - |
| **External IOBs** | 6 out of 640 (1%) | 389 out of 640 (60%) | - |
| **DSP48E** | 3 out of 64 (4%) | 0 out of 64 (0%) | - |

**Table 5.** Resources used by the AES algorithm

In *Table 6* we have provided the resources used by the Microblaze processor.

| Device Feature | Miroblaze resources |
|---|---|
| **Slice Registers** | 1340 out of 69120 (1%) |
| **Slice LUTs** | 1660 out of 69120 (2%) |
| **Occupied Slices** | 793 out of 17280 (4%) |
| **Block RAM (36k)** | 8 out of 148 (5%) |
| **External IOBs** | 2 out of 640 (1%) |
| **DSP48E** | 3 out of 64 (4%) |

**Table 6.** Microblaze Resources

As we can see in *Table 5* column 2 and column 3 are very different despite the fact that we are using the same design. This is due to the size of the microblaze processor which is not calculated when using any ISE program. When adding the data in column 3 of *Table 3* and column 2 of *Table 6* the total is almost equal to the resources in column 2 of *Table 5.* The remaining resources used, are due to the PLB buses that also consume resources.

The total cycles required for encryption were also calculated at this point. The XPS_Timer was added to the design through the EDK software. A total of 669 cycles were needed to encrypt a 128 bit

block using a 128 bit key. These cycles do not include the time required by the user to enter the data and the time required to transfer the result to the PC. We next tested if these cycles would change with a large text file as an input instead of just 128 bits. When using a file, the time required to produce the output did not change, the algorithm produced all the outputs in a steady number of cycles.

In order to use the PRCC we needed to calculate the bitstream size in KBytes. A total of 80.44 Kbytes are needed for our AES partial bitstream. The reason for using only the occupied slices consumed by our design to calculate the configuration bits needed by the bitstream and not including the I/Os, is that the inputs and outputs are a parameter of the Partial Reconfiguration Region and do not affect the design placed in the region.

The first step for calculating the Reconfiguration Time and Throughput, was to choose our design parameters then select a FPGA and last enter our bitstream size. The data entered and the results can be found in *Table 7*. The reason we can choose any Virtex-5 FPGA and see no change in the reconfiguration times is that the frame size for the entire Virtex-5 family is 164 Bytes.

| *External Memory* | *Memory Bus Clock (MHz)* | *Memory Interface width (bits)* | *Processor Cache* | *Device* | *Reconfiguration Time (msec)* | *Reconfiguration Throughput (MB/sec)* |
|---|---|---|---|---|---|---|
| DDR | 100 | 8 | ON | Any Virtex 5 | 9.483 | 9.105 |
| DDR | 100 | 8 | OFF | Any Virtex 5 | 157.419 | 0.548 |
| DDR | 100 | 16 | ON | Any Virtex 5 | 7.009 | 12.319 |
| DDR | 100 | 16 | OFF | Any Virtex 5 | 116.35 | 0.742 |
| DDR | 100 | 32 | ON | Any Virtex 5 | 5.772 | 14.959 |
| DDR | 100 | 32 | OFF | Any Virtex 5 | 95.815 | 0.901 |
| DDR | 100 | 64 | ON | Any Virtex 5 | 5.772 | 14.959 |
| DDR | 100 | 64 | OFF | Any Virtex 5 | 95.815 | 0.901 |

**Table 7.** PRCC Settings and Results for AES

The results showed that the processer cache plays a major role in the reconfiguration time and throughput. With the cache turned on the reconfiguration time and throughput are about sixteen times smaller compared to the time needed when the cache is turned off. The reconfiguration time and throughput are also affected by the memory bus interface. We observed that, with increasing the width of the interface we can achieve slightly better performance of our design. Finally, there was no change in the reconfiguration time and throughput despite using a 32 or 64 bit memory interface width.

Comparing our theoretical results with the results seen in [10], when using a PLB bus and the cache enabled, we saw that our measurements do not differ by a large margin. We also took into consideration that those experiments were done with the Virtex-4.

*Figure 23* shows a layout of the AES design on the FPGA board. The Microblaze processor is connected to its memory through the dlmb and ilmb memory busses. Next, through the PLB bus the processor is connected the various IPs of the design. In our case, we had an AES 128, which is a custom IP designed by us and the RS232 interface used for communication with a terminal program on the computer. The PLB is 32bits wide, the system clock frequency is 125MHz and the Microblaze memory is 32Kb.

*Figure 24* illustrates the steps followed when running the AES algorithm on the FPGA board which is connected to the computer. Once the design is downloaded to the FPGA, the Microblaze processor waits for data from the RS port. The user first enters the key which is to be used for encryption; it must be 128 bits before the design can continue to function. Next, follows the data for encryption and the same rules apply as above for the size. Once all data has been received by the processor, it is forwarded to our AES peripheral. After the encryption is complete, the result is sent to the processor and from there, to the RS232 peripheral port, in 32 bit blocks, due to the size limitation of the PLB bus.

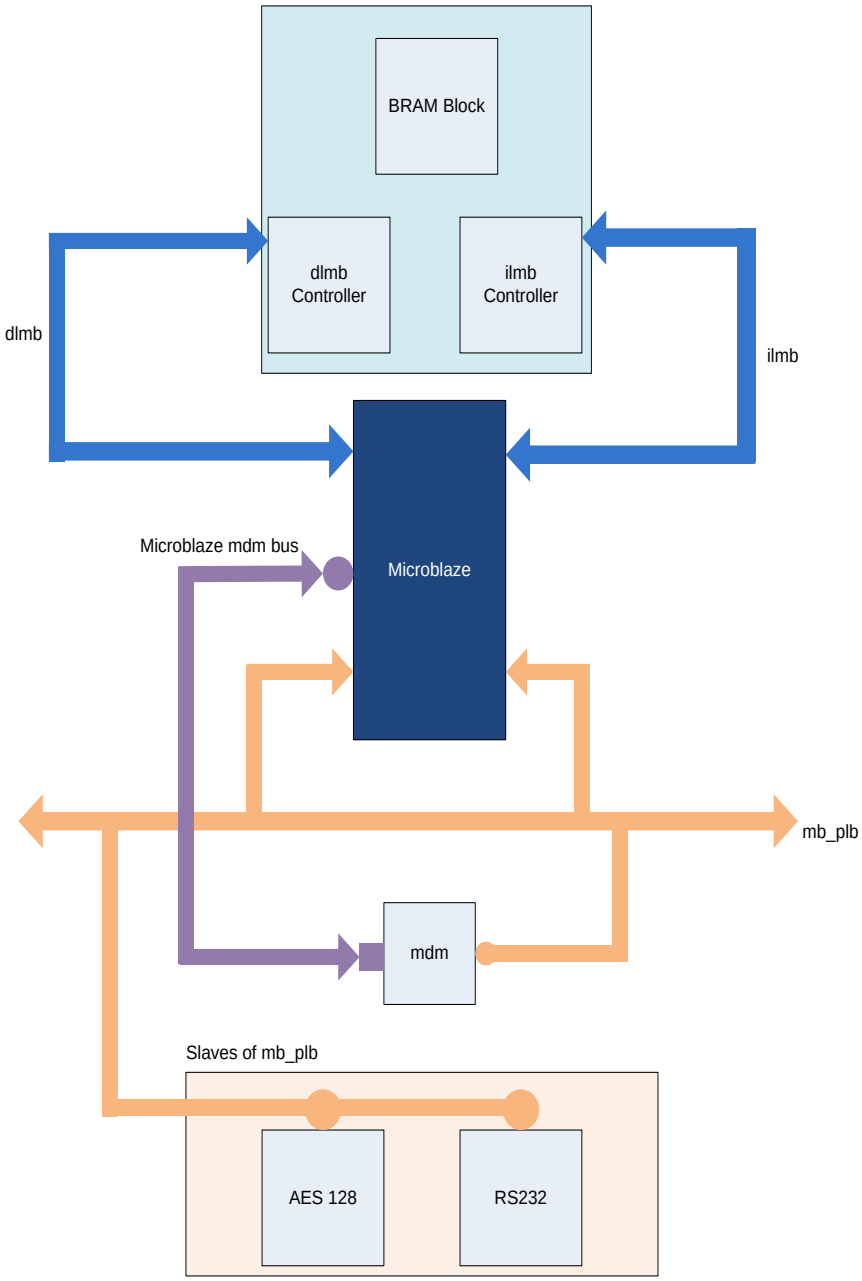


**Figure 23.** AES layout on FPGA

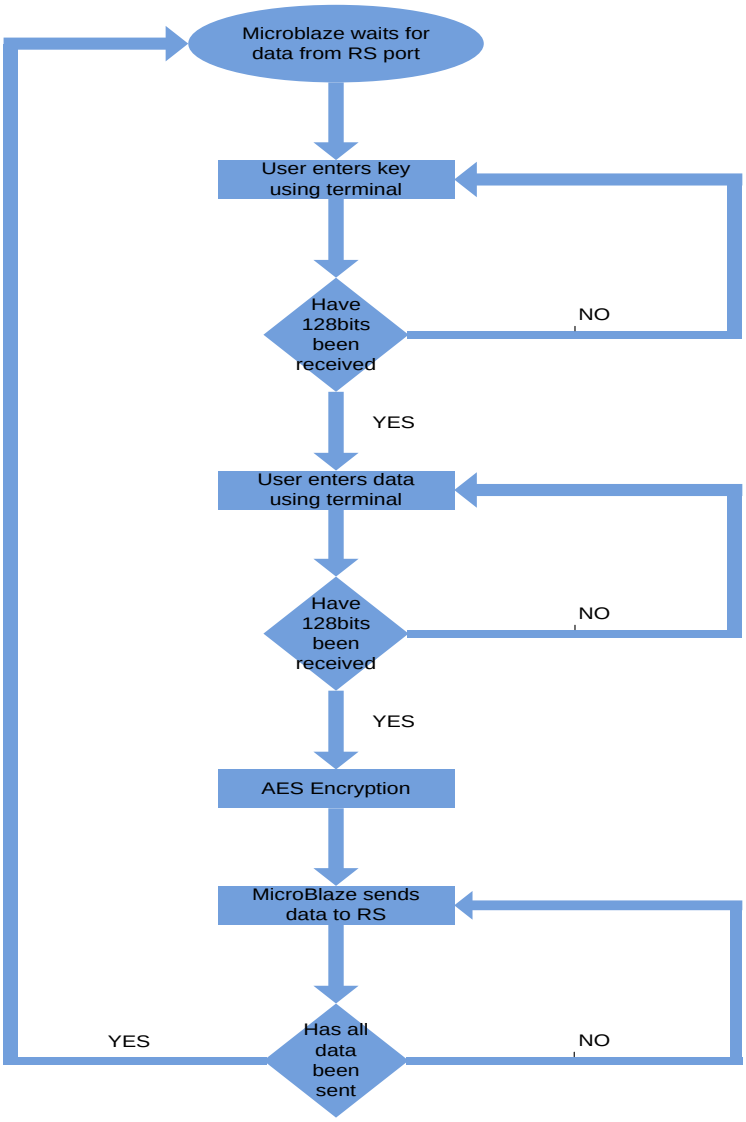


**Figure 24**. Flow chart for AES design

## 5.2.2 Advanced Encryption Standard FSL Analysis

In order to compare speed and resource consumption of our design on the PLB, the AES algorithm was placed on a Fast Simplex Link (FSL). In order to connect our peripheral to the FSL, a controller was written in VHDL, to control the reads and writes to the FIFO.

*Table 8* shows the resources consumed by our new design. Compared to the results in *Table 5* we observed a slight, in the data in column 2, such as slice registers and occupied slices. Although, there

was an increase in the slice LUTs needed. Column 2 displays the total resources our design consumed when placed on the FPGA board, this includes all peripherals, busses and the Microblaze processor.

The opposite occurs in column 3 of *Table 8*, here we observed an increase in the resources consumed by the AES peripheral alone. Due to the fact that our FSL design occupies more slices compared to the PLB design, more configuration bits are also needed.

| Device Feature | Total Resources Used in EDK 12.3 | Resources Used in ISE 12.3 | Configuration Bits Used |
|---|---|---|---|
| **Slice Registers** | 2557 out of 69120 (3%) | 981 out of 69120 (1%) | - |
| **Slice LUTs** | 3729 out of 69120 (5%) | 1771 out of 69120 (2%) | - |
| **Occupied Slices** | 1510 out of 17280 (8%) | 685 out of 17280 (3%) | 808985 |
| **Block RAM (36k)** | 8 out of 148 (5%) | 0 out of 148 (0%) | - |
| **External IOBs** | 11 out of 640 (1%) | 76 out of 640 (11%) | - |
| **DSP48E** | 3 out of 64 (4%) | 0 out of 64 (0%) | - |

**Table 8.** Resources used by AES on FSL

*Figure 26* shows the block diagram of our design using the FSL. The only changes made in comparison to *Figure 23*, is the fact that the AES peripheral is now connected to microblaze processor through a master and slave FSL. The processor places data in the master FSL and reads data out of the slave FSL, the AES peripheral operates in the same manner.

Finally, the cycles required to encrypt the input were calculated. Using the same method with the PLB design, a total of 740 cycles were needed. This number is 71 cycles larger than the cycles required by the PLB design. This is mainly due to the software code used to run the algorithm, which consists of for-loops. We have previously shown in chapter 3, that these loops consume a large number of cycles. Although the FSL is faster than the PLB bus, a total of 5 cycles are required to write to the FSL when the PLB needs 10, the controller used to run our algorithm using the FSL is slower than that used to control the peripheral on the PLB. Unfortunately, we were forced to use different controllers due to the fact that the PLB works with registers and the FSL with FIFOs. Attempts were made to speed up the FSL controller.

Tests were also run using larger inputs such as a file. In order to gather and evaluate the results our python script was used to export the results from CuteCom to Excel. The file used had a total of 23 inputs, each 128bits long and the same key was used for each run. In order for CuteCom to send all data to the Microblaze processor correctly, a char delay of 10ms was required. This setting can be found in

CuteCom. The results showed that the time required to encrypt the data was not steady but had two different values 740 and 725 cycles. The results can be seen in *Figure 25*.

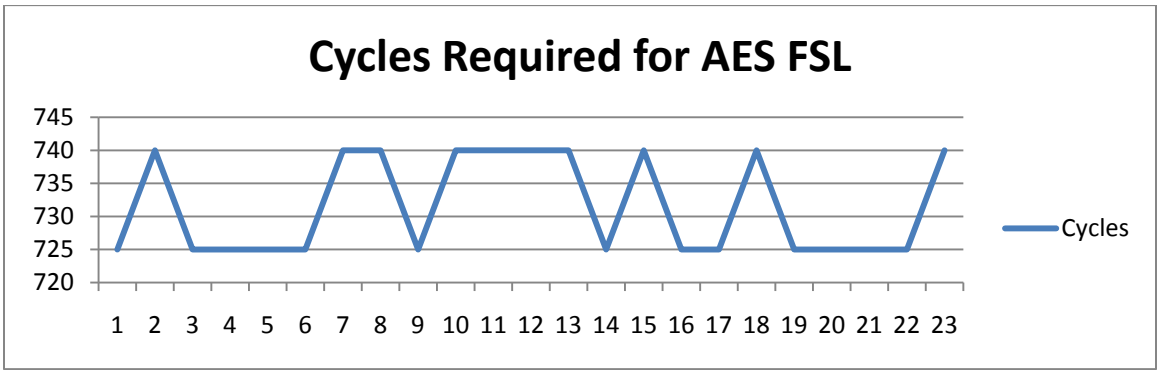


**Figure 25.** Cycles Required for AES Encryption on FSL

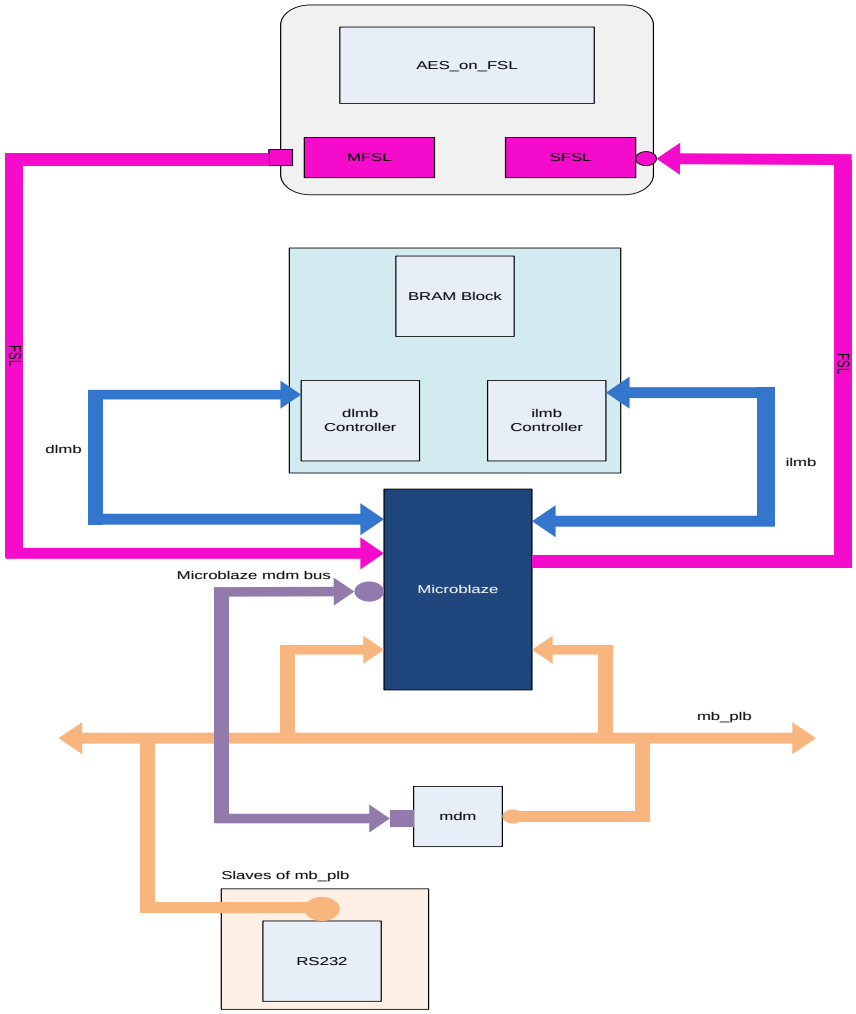


**Figure 26.** AES FSL Block Diagram

*Table 9* places the two complete designs side by side in order to allow comparison. As we can see, the FSL design required less resources in every feature except for the number of slice LUTS. It needed 188 fewer than the PLB design.

*Table 10* illustrates the resources consumed only by the two peripherals and not the complete design. Once again, the FSL peripheral required more resources than the PLB peripheral.

| Device Feature | Total PLB Resources | Total FSL Resources |
|---|---|---|
| **Slice Registers** | 2748 out of 69120 (3%) | 2557 out of 69120 (3%) |
| **Slice LUTs** | 3655 out of 69120 (5%) | 3729 out of 69120 (5%) |
| **Occupied Slices** | 1628 out of 17280 (9%) | 1510 out of 17280 (8%) |
| **Block RAM (36k)** | 8 out of 148 (5%) | 8 out of 148 (5%) |
| **External IOBs** | 6 out of 640 (1%) | 11 out of 640 (1%) |
| **DSP48E** | 3 out of 64 (4%) | 3 out of 64 (4%) |

**Table 9.** AES PLB design vs. FSL design

| Device Feature | PLB Peripheral Resources | FSL Peripheral Resources |
|---|---|---|
| **Slice Registers** | 676 out of 69120 (1%) | 981 out of 69120 (1%) |
| **Slice LUTs** | 1424 out of 69120 (2%) | 1771 out of 69120 (2%) |
| **Occupied Slices** | 558 out of 17280 (3%) | 685 out of 17280 (3%) |
| **Block RAM (36k)** | 0 out of 148 (0%) | 0 out of 148 (0%) |
| **External IOBs** | 389 out of 640 (60%) | 76 out of 640 (11%) |
| **DSP48E** | 0 out of 64 (0%) | 0 out of 64 (0%) |

**Table 10.** AES PLB Peripheral vs. AES FSL Peripheral

## 5.2.3 Blowfish PLB Analysis

Using the same method with the AES algorithm we estimated the resources used by the Blowfish encryption algorithm, these can be found in *Table 9*. Once again, in the second column we see the total resources needed by the design, which contains a Microblaze processor, our custom Blowfish peripheral, the RS232 interface and the LED_display interface. The third column shows the resources used only by our Blowfish encryption algorithm which were estimated using the ISE 12.3 software. Due

to the fact that the Blowfish algorithm uses a small amount of resources, it could be better to have it as a static region on the FPGA instead of swapping it in and out of our design.

| Device Feature | Total Resources Used in EDK 12.3 | Resources Used in ISE 12.3 | Configuration Bits Used |
|---|---|---|---|
| **Slice Registers** | 2075 out of 69120 (3%) | 70 out of 69120 (1%) | - |
| **Slice LUTs** | 2256 out of 69120 (3%) | 134 out of 69120 (1%) | - |
| **Occupied Slices** | 1193 out of 17280 (6%) | 67 out of 17280 (1%) | 79127 |
| **Block RAM (36k)** | 8 out of 148 (5%) | 0 out of 148 (0%) | - |
| **External IOBs** | 10 out of 640 (1%) | 180 out of 640 (28%) | - |
| **DSP48E** | 3 out of 64 (4%) | 0 out of 64 (0%) | - |

**Table 11.** Resources used by the Blowfish algorithm

Using the results from *Table 11*, our bitstream size for the Blowfish algorithm had a total size of 9.66 KBytes. Once again, using the PRCC website we measured the theoretical reconfiguration time and throughput for the encryption algorithm.

| *External Memory* | *Memory Bus Clock (MHz)* | *Memory Interface width (bits)* | *Processor Cache* | *Device* | *Reconfiguration Time (msec)* | *Reconfiguration Throughput (MB/sec)* |
|---|---|---|---|---|---|---|
| DDR | 100 | 8 | ON | Any Virtex 5 | 1.051 | 8.973 |
| DDR | 100 | 8 | OFF | Any Virtex 5 | 17.453 | 0.543 |
| DDR | 100 | 16 | ON | Any Virtex 5 | 0.777 | 12.14 |
| DDR | 100 | 16 | OFF | Any Virtex 5 | 12.9 | 0.731 |
| DDR | 100 | 32 | ON | Any Virtex 5 | 0.64 | 14.741 |
| DDR | 100 | 32 | OFF | Any Virtex 5 | 10.623 | 0.888 |
| DDR | 100 | 64 | ON | Any Virtex 5 | 0.64 | 14.741 |
| DDR | 100 | 64 | OFF | Any Virtex 5 | 10.623 | 0.888 |

**Table 12.** PRCC Settings and Results for Blowfish

Results from *Table 12* once again, showed that the processor cache plays a major factor in the reconfiguration time and throughput. By enabling the cache we can achieve results about sixteen times better than with the cache disabled.

## 5.2.3 Blowfish FSL Analysis

Changes were made to our Blowfish peripheral in order to connect it to the Microblaze processor using the Fast Simplex Links (FSL). After the peripheral was successfully added to the design measurements were taken to determine the size and speed of the new peripheral. *Table 13* in column two, shows the resources used by our design, including all of the peripheral implemented into the design. Column three on the other hand, shows the resources used only by the blowfish peripheral.

| Device Feature | Total Resources Used in EDK 12.3 | Resources Used in ISE 12.3 | Configuration Bits Used |
|---|---|---|---|
| **Slice Registers** | 1787 out of 69120 (2%) | 222 out of 69120 (1%) | - |
| **Slice LUTs** | 2142 out of 69120 (3%) | 185 out of 69120 (1%) | - |
| **Occupied Slices** | 1035 out of 17280 (5%) | 126 out of 17280 (1%) | 148806 |
| **Block RAM (36k)** | 8 out of 148 (5%) | 0 out of 148 (0%) | - |
| **External IOBs** | 10 out of 640 (1%) | 73 out of 640 (11%) | - |
| **DSP48E** | 3 out of 64 (4%) | 0 out of 64 (0%) | - |

**Table 13.** Blowfish FSL Peripheral Resources

Tests showed the total number if cycles required for our peripheral to encrypt the data was 54 cycles. This time only includes the cycles required to receive the data from the processor for encryption and calculate the output. It does not include the time to send the data on the PC. We next entered more data using a text using the same method as with the FSL AES design. The results showed that the time required to encrypt the data was not steady but had two values, 54 and 39 cycles. We used a total of 42 16 bit inputs for this test. The results can be seen in *Figure 27*.

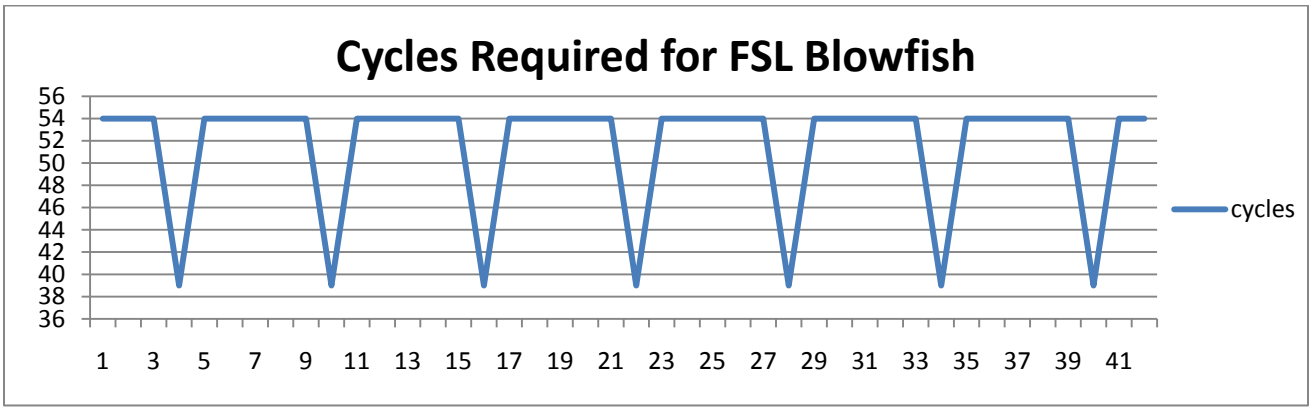


**Figure 27.** Cycles Required for FSL Blowfish

| Device Feature | Total PLB Resources | Total FSL Resources |
|---|---|---|
| **Slice Registers** | 2075 out of 69120 (3%) | 1787 out of 69120 (2%) |
| **Slice LUTs** | 2256 out of 69120 (3%) | 2142 out of 69120 (3%) |
| **Occupied Slices** | 1193 out of 17280 (6%) | 1035 out of 17280 (5%) |
| **Block RAM (36k)** | 8 out of 148 (5%) | 8 out of 148 (5%) |
| **External IOBs** | 10 out of 640 (1%) | 8 out of 640 (1%) |
| **DSP48E** | 3 out of 64 (4%) | 3 out of 64 (4%) |

**Table 14.** Blowfish PLB Design vs. FSL Design

*Table 14* shows a comparison between the PLB design and the FSL design. As we can see the FSL design is smaller than the PLB design in all the categories.

| Device Feature | PLB Peripheral Resources | FSL Peripheral Resources |
|---|---|---|
| **Slice Registers** | 70 out of 69120 (1%) | 222 out of 69120 (1%) |
| **Slice LUTs** | 134 out of 69120 (1%) | 185 out of 69120 (1%) |
| **Occupied Slices** | 67 out of 17280 (1%) | 126 out of 17280 (1%) |
| **Block RAM (36k)** | 0 out of 148 (0%) | 0 out of 148 (0%) |
| **External IOBs** | 180 out of 640 (28%) | 73 out of 640 (11%) |
| **DSP48E** | 0 out of 64 (0%) | 0 out of 64 (0%) |

**Table 15.** PLB Blowfish Peripheral vs. FSL Peripheral

*Table 15* compares the resources consumed only by the two Blowfish peripheral. As we can see the FSL peripheral uses more compared to the PLB. Due to the fact that the FSL peripheral occupies more slices, the bitstream will also be larger. In fact, the FSL bitstream is 69679 bits larger, almost twice the size of our PLB design.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

This work introduces a complete system that uses the AES and Blowfish algorithms for data encryption running on a Virtex-5 VLX-110t FPGA board. Both algorithms were placed on the design as a PLB and FSL peripheral. The systems created during this thesis can be used for projects currently running in the lab, such as the S.M.A.R.T project. The purpose of this work was to analyze the reconfiguration times and consider whether this approach was suitable for our design.

In order to measure the reconfiguration time and throughput of our design, the Partial Reconfiguration Cost Calculator was used. This tool allowed us get an idea of the times needed for the reconfiguration process. Even though this tool was developed for the Power PC processor we saw that our results did not differ very much from those seen in [*10*].

In order to collect and process faster the results from our setups, a Python script was developed. This script creates a new file which puts our results in a table. This new file can then be used with Excel to develop graphs and charts to further analyze the results of the experiments. One of the main advantages of this script is that it can be used with any application, allowing users to collect their results.

After analyzing the results from the PRCC we came to the conclusion that using Partial Reconfiguration for the AES and Blowfish algorithm has no advantages. This is mainly due to the large difference in their bitstream size. For example, when placing the Blowfish algorithm in the Partial Reconfiguration Region, more than half of it will be empty due to its small size. In order to benefit from partial reconfiguration we need to use bitstreams that do not have a large difference in size.

Must of the time spent in this thesis was to first get the tools to work with the operating system and second to make changes to the AES and Blowfish algorithm in order to automate the design and also to place both modules on the FSL. The most complex part of the design process was working with the Xilinx EDK software.

Finally, due to difficulties such as obtaining licenses for our software, we were not able to perform partial reconfiguration on our design and had to rely on our theoretical measurements using the PRCC. Partial reconfiguration will be attempted in future work.

## 6.2 Future Work

Due to Xilinx presenting new software tools with such  a fast rate and fixing bus from previous editions, it is necessary for our design to be upgraded to the newer tools. As we have seen there were several bugs in the EDK 12.3 edition. Probably, with the newer versions the bugs will be fixed and better cores will be provided.

A real application using partial reconfiguration should be implemented and evaluated in order to compare our theoretical results with real experimental results. Due to the large difference in the size of the two bitstreams it may also be a good idea to use a different algorithm than the Blowfish. This algorithm should have a bitstream size closer to that of the AES algorithm.

# References


1. Katherine Compton - Scott Hauck, *"Reconfigurable Computing: A Survey of Systems and Software"*, in ACM Computing Surveys, Vol. 34, June 2002, pp. 171-210.
2. Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young and Brendan Bridgford, *"Invited Paper : Enhanced Architecture, design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAS".*
3. Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan and Prasanna Sundarrajan, *"A Self-reconfiguring Platform"* , in FPL 2003, LNCS 2778, pp. 565-574, 2003.
4. Kyprianos Papadimitriou, Antonis Anyfantis and Apostolos Dollas, *"An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems"*, in IEEE transactions on instrumentation and measurement, Vol. 59, No. 6, June 2010.
5. Kyprianos Papadimitriou, Apostolos Dollas and Scott Hauck *"Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model"*, in ACM Transactions on Reconfigurable Technology and Systems.
6. "*Two Flows for Partial Reconfiguration: Module Based or Difference Based"*, Xilinx Application note (v1.2) September 9, 2004.
7. Victor Lai and Oliver Diessel *"ICAP-I: A Reusable Interface for the Internal Reconfiguration of Xilinx FPGAs".*
8. Shaoshan Liu, Richard Neil Pittman, Alessandro Forin *Microsoft Research* "*Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller*", Technical Report MSR-TR-2009-150 September 2009.
9. John McCaskill and David Lautzenheiser "*FPGA Partial Reconfiguration Goes Mainstream*", Xcell Journal Fourth Quarter 2010.
10. Ming Liu, Wolfgang Kuehn, Zhonghao Lu, Axel Janstsch "*Run-Time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration".*
11. Kyprianos Papademetriou, Apostolos Dollas *"Performance evaluation of a Preloading Model In Dynamically Reconfigurable Processors".*
12. Valery Sklyarov, Iouliia Skliarona, Arnaldo Oliveira, Antonio B. Ferrari *"A Dynamically Reconfigurable Accelerator for Operations over Boolean and Ternary Vectors",* Proceedings of the Euromicro Symposium on Digital System Design 2003 IEEE.

13. Ming Liu, Zhonghai Lu, Wolfgang Kuehn, Axel Jantsch *"Reducing FPGA Reconfiguration Time Overhead using Virtual Configurations".*
14. Christopher Claus, Rehan Ahmed, Florian Altenried, Walter Stechele *"Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance System"*, ARC 2010, LNCS, pp. 55-67.
15. *"ISE Design Suite 12: Installation, Licensing and Release Notes"*, UG 631 ver.12.3 September 21, 2010.
16. *"Virtex-5 Family Overview"*, DS100 (v 5.0) February 6, 2009.
17. Nikoloudakis Georgios *"Encryption Applications Using Reconfigurable Logic"*, Diploma Thesis, University Of Crete, 2009.
18. www.rmdir.de/~michael/xilinx/.
19. www.fpgadeveloper.com.
20. *"ML505/ML506/ML507 Evaluation Platform"* Xilinx User Guide, UG347 (ver. 3.1.1) October 7, 2009.
21. *http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.*
22. *http://en.wikipedia.org/wiki/Blowfish_%28cipher%29.*
23. *"Virtex-5 FPGA Configuration User Guide"* Xilinx User Guide, UG191 (v3.5) October 29, 2008.
24. "*SEU Strategies for Virtex-5 Devices*", Xilinx Application Note, XAPP864 (v2.0) April 1, 2010.
25. *http://users.isc.tuc.gr/~kpapadimitriou/prcc.html.*
26. Papadopoulos Konstantinos "*Implementation of Security Algorithms for Wireless Sensor Networks Using Reconfigurable Devices*", Master Thesis, Technical University of Crete, October 2009.

# Appendix A

This appendix includes all the details for installing and running the required software used throughout this thesis.

## A.1 Installing Cable Driver

A series of steps were followed in order to install the drivers required for the Xilinx cable.

1. download usb-driver-HEAD.tar.gz
2. Run command "gunzip usb-driver-HEAD.tar.gz"
3. Run command "tar -xf usb-driver-HEAD.tar"
4. Run command "cd usb-driver"
5. Run command "make"
6. Run command "ls libusb-driver.so"
7. export LD PRELOAD=/path/to/libusb-driver.so
8. Reboot PC
9. The LED on the USB cable should light up yellow

In order to check if the driver was installed correctly, log in as root in a terminal and run command "lsusb | grep Xilinx" the output should be:

```
Bus 004 Device 004: ID 03fd:0008 Xilinx, Inc.
```

It is important that the number is marked in red. This states that the cable firmware was loaded correctly.

## A.2 Installing Terminal Program

1. Download cutecom from "cutecom.sourceforge.net/"
2. Instructions on how to install can be found in folder downloaded
3. After installation enter YAST
4. Add uucp to user group and logout

5. Change mode on serial port to using command "/sbin/chmod 666 /dev/ttyS0".

## A.3 Start Up Script for ISE

1. Open a new script
2. Write the following commands in the script:

```
#! /bin/sh
source /opt/Xilinx/12.3/ISE/settings32.sh
export LD_PRELOAD=/home/your_home_directory/usb-driver/libusb-driver.so
ise
#impact
exit
```

3. Save file
4. In a terminal window run command "chmod 775 fileName"

When running this script ISE should open.

## A.4 Start Up Script for EDK

1. Open a new script
2. Write the following commands in the script:

```
#!/bin/bash

# Please adapt these values to your system configuration.
XILINX_DIR=/opt/Xilinx/12.3/ISE_DS/
EDK_DIR=/opt/Xilinx/12.3/ISE_DS/EDK

# load settings
. ${XILINX_DIR}/settings32.sh
export XILINX_EDK=/opt/Xilinx/12.3/ISE_DS/EDK
export LD_LIBRARY_PATH=${XILINX_EDK}/bin/lin:${LD_LIBRARY_PATH}
export
PATH=${XILINX_EDK}/bin/lin:${XILINX_EDK}/gnu/microblaze/lin/bin:\
```

```
${XILINX_EDK}/gnu/powerpc-eabi/lin/bin:${PATH}

# start xilinx ise
${EDK_DIR}/bin/lin/xps
```

Save file

3. In a terminal window run command "chmod 775 fileName"

When running this script EDK should open.

# Appendix B

In this appendix we will review the bugs we encountered during the use of the Xilinx 12.3 software.

## B.1 When Creating a FSL Peripheral

The first bug we encountered during the use of EDK 12.3 was when creating a FSL peripheral using the wizard. Once the peripheral was made, we observed that in the peripheral_name.vhd file the FSL_S_Clk and FSL_M_Clk were set to outputs. In order for our design and any design to work correctly, these had to change to inputs each time we created a new FSL peripheral.

## B.2 When creating a PLB Peripheral

After creating a PLB peripheral and attempting to build the user applications, we received an error stating that XIo_Out32 and XIo_In32 were not declared. In order to fix this problem we had to make changes to the peripheral_name.h file. In this file, instead of using *XIo_Out32* and *XIo_In32*, the older commands *xil_IO_out32* and *xil_IO_in32* were used. After making the changes, the above error did not appear.

# Appendix C

In this appendix we provide the Python script used to gather the results from the experiments in order to import them to Excel for further analysis. This script can be used to locate any word in any file. In order for the script to locate the number of cycles, the output from the Cutecom log must have the following format, "Cycles: *number*". This can be done when writing the C-script that controls our system. This way when we ask the script to locate the word cycles it will also save the next word in the file, in a table, which in our case will be the cycles required to execute a series of commands.

```
import re
repeat=1
while (repeat==1):      #while will continue until y or n is given
      question=raw_input("Will you be using CuteCom's output (y or n)?: ")
      if question=='y':
            openFile = '/home/yorgon/cutecom.log' #Opens the cutecom log
            repeat=0    #set repeat to 0 to exit loop
      elif question=='n':
            openFile = raw_input("Please enter a file to open: ")
            repeat=0
      else:
            repeat=1
f= open(openFile, "r")
fileToWrite= open("/home/yorgon/Desktop/results", "w")  #file to write data
pattern = raw_input("Enter a word to locate in file: ")#Enter word to locate
regexp=re.compile('^'+pattern+'$',re.I)   #creates the pattern to match
lineNum= 0
wordNum= 0
match=0
fileToWrite.writelines('The word you are trying to match is: ' +pattern+'\n')
wordList=[]       #list for all the words in the input file
wordMatches=[]    #list for the spot of the match in the input file
with f as searchfile: #Search the whole file
    for line in searchfile:
      lineNum=lineNum + 1
        lineL = line.split()   #split line into words
      for word in lineL:
            word=re.sub(r'\W+', '', word)    #Ignore all '.', ',', '!' etc.
            wordList.append(word)            # put all words in a list
            result=regexp.match(word)
            if result:
                  match=match+1;
                  wordMatches.append(wordNum) #puts location of match in list
                  value=str(wordNum) #Make match a string to be able to write
                  fileToWrite.writelines('We have a match! It is word number:
' +value+'\n')
                  print 'Your word was found in line',lineNum, 'and is
word',wordNum
```

```
            wordNum=wordNum+1       #number of next word
print 'Total matches in text:', match
x1='TEST'
x2='CYCLES'
print '{0:2}  {1:10}'.format(x1, x2)
fileToWrite.writelines('{0:2}  {1:10}'.format(x1, x2)+'\n')
for index in range(len(wordMatches)):  #loop prints the next word after the
match
      nextWord=wordMatches[index] +1   #go to next word
      testNum=index
      print '{0:2}    {1:10}'.format(testNum, wordList[nextWord])
      fileToWrite.writelines('{0:2}    {1:10}'.format(testNum,
wordList[nextWord])+'\n')  #writes to file in two columns
f.close()
fileToWrite.close()
```