Project Code: BS02

Crypto Acceleration Using Asynchronous FPGAs

A Major Qualifying Project Report

Submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

Submitted By:

________________ ________________

Bryce Barcelo John Taylor

Sponsored by:

General Dynamics C4 Systems
77 A Street
Needham, MA 02494

Liaisons:

Brendon Chetwynd
Gerardo Orlando

Submitted To:

____

Prof. Berk Sunar

Abstract

The goal of this project, sponsored by General Dynamics C4 Systems, is to evaluate proprietary FPGA technology developed by Achronix Semiconductor Corporation and its effectiveness using a 128-bit, one clock cycle multiplier in a finite field, GF(2^{128}), as a test application. The testing will determine if there is a significant increase in speed that can be achieved by simple modifications of existing synchronous HDL designs using three metrics: number of LUTs, number of registers, and clock speed.

Acknowledgements

This project set out from the beginning with high hopes and expectations, none of which could have been accomplished without the continued support from our colleagues at General Dynamics C4 Systems and Worcester Polytechnic Institute, who provided unending support that allowed us to complete this Major Qualifying Project.

We would like to thank Brendon Chetwynd for the training he provided to us during the busy hours of his work day. We would also like to thank Gerardo Orlando for his explanations into the details that were required to synthesize and code this project correctly and Evan Custodio, a previous MQP student who participated at GDC4S, who provided us with contact information and helped us work through software difficulties. The Achronix Corporation was also kind enough to send Scott Erik Norrholm to the GDC4S site to provide us with a training session in use of the Achronix CAD Environment as well as explain some of the disclosable workings regarding the Achronix proprietary technology.

Finally, we would like to thank Professor Sunar who advised this project and provided us with the opportunity to perform this project at General Dynamics C4 Systems as part of an MQP program he has kept strong during his years at WPI. The guidance provided by these individuals was nothing less than necessary as we trained in and explored the tools of the trade used by professional engineers in the world today.

Table of Contents

Table of Figures

Table of Equations

1 Introduction

Field Programmable Gate Arrays (FPGAs) are used in commercial and industrial applications for both implementation and testing more so than they have ever been used before. A combination of cheaper technology coupled with faster and more reliable boards have made FPGAs one of the best tools for system development available to engineers. An FPGA allows the luxury of reworking and retooling a design on-the-fly without having to manufacture or build the design every time a new revision is decided upon. What FPGAs face, in terms of future development issues, is that they are not keeping up with the demand for speed and throughput that high-complexity applications, such as cryptography, are beginning to require. The limiting factor in most FPGA designs is the clock speed that must be carefully tuned for the system as to not disrupt the balance of the system clock and the synchronous components that require a clock. FPGAs achieve a much faster system throughput if they are designed asynchronously, but designing for an asynchronous system is time-consuming and extremely difficult.

In cooperation with General Dynamics C4 Systems and the Achronix Semiconductor Corporation, this MQP will explore experimental, proprietary technology developed by Achronix that will improve throughput speeds of a synchronous FPGA design to theoretical speeds of 1.6GHz+. This is accomplished by a proprietary blend of removing routing delays and allowing the system to run without a clock, asynchronously. This experiment will use a Galois Field multiplier, GF(2^{128}), that runs within one system clock cycle as a testing application. This multiplier serves a valid and important purpose in cryptographic systems such as Advanced Encryption Standard (AES) and Galois Counter Mode (GCM), thus the reason it was chosen. The system will be tested across two simulated FPGAs: the Achronix Speedster ACXSPD60 (Std. speed, FBGA1680 package) and the Altera Stratix III EP3SL150 (-2 speed, FC1152 package). The designs will be optimized using Synplicity's Synplify Pro and simulated for timing analysis within the Achronix CAD Environment and Altera Quartus II for the Speedster and the Stratix III, respectively. Of the results produced by these tools, the number of lookup tables (LUTs), logic registers, and the system clock speed will serve as benchmarks for the performance of the systems under test.

As with any MQP performed off-campus and in conjunction with an engineering firm such as GDC4S, time constraints and project goals must be kept in a delicate balance to support producing the best results in the seven week time frame. This project will accomplish the goals it has set out for itself, with the approval of its supervisors, as well as propose future research and experimentation that could be explored in future MQPs.

2 Background

This project is a culmination of two primary efforts: research and implementation. This section will explain the broad concepts necessary towards a complete understanding of the finished result and the design methodology.

2.1 Field Programmable Gate Arrays (FPGAs)

An FPGA is a semiconductor device in which a large array of logic blocks can be programmed to be connected to each other in different ways. This enables the designer to create a hardware device that is specifically designed for a particular task, without the need for the expensive process involved in creating an Application Specific Integrated Circuit (ASIC).

Since the first FPGAs created by Xilinx Inc. came to market in 1985, their use has been increasing steadily in a variety of disciplines. Not only are the costs for small batches of FPGAs significantly lower than those of comparable ASICs, but they are also easier to change once the initial design has been deployed. While FPGAs are generally slower and use more power than an ASIC designed for the same task, advancements in FPGA fabrication technology mean that they are rapidly reaching a speed and power utilization that enables them to be used in almost any application [13].

2.2 Hardware Encryption

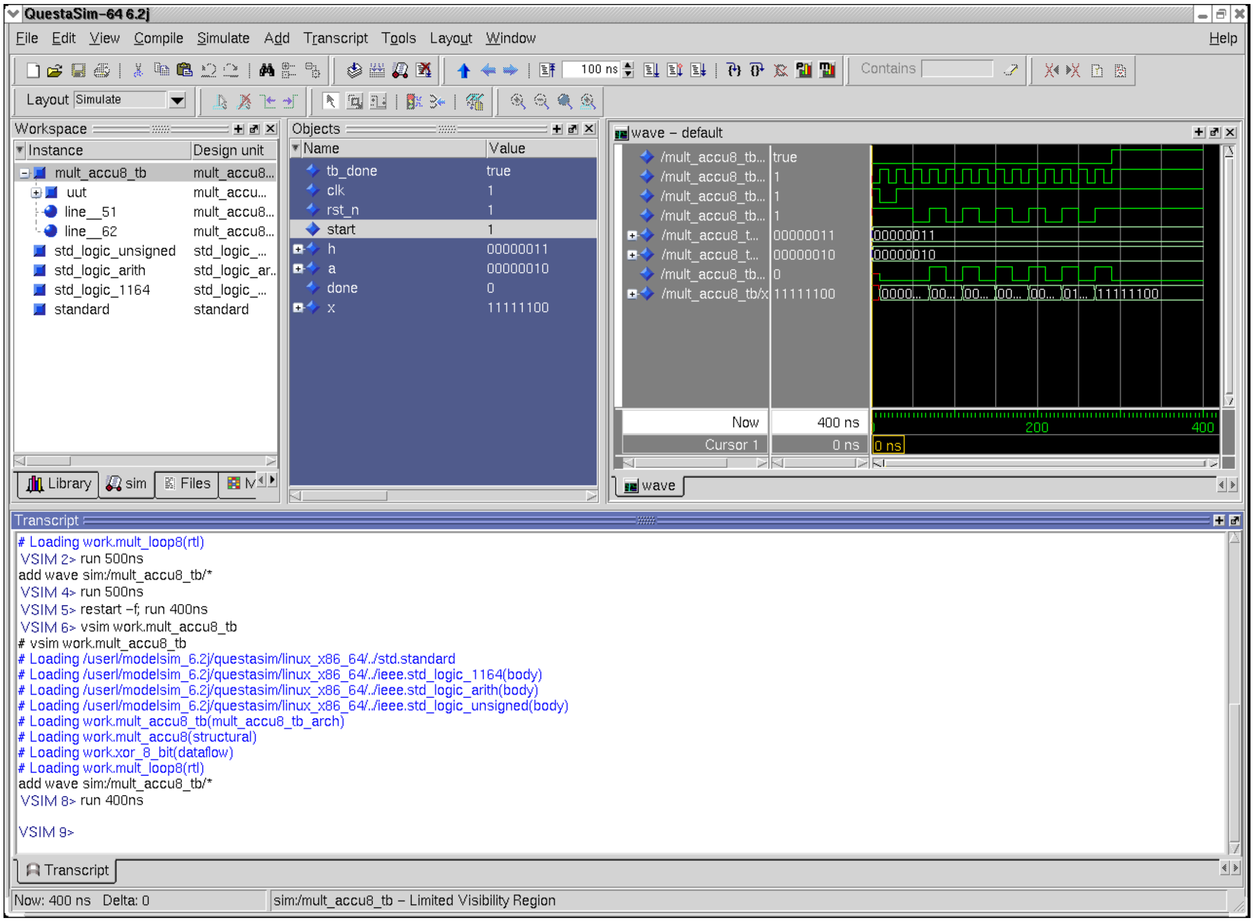
It is desirable to perform encryption in hardware for applications where large amounts of data need to be encrypted, transmitted, decrypted, and received in a time critical manner the performance gains provided by hardware encryption outweigh its increased cost. Specifically, algorithms like AES (elaborated upon in section 2.6 Advanced Encryption Standard (AES)) are particularly easy to implement in hardware due to the fact that most of the computations are based on bit manipulation, which run much more efficiently in hardware as opposed to software [14].

2.3 Available Tools

Throughout the course of this project, multiple software tools were used in order to facilitate simulation, version comparison and simplification processes to increase productivity. The most important of these tools are explained in further detail in this section.

2.3.1 Mentor Graphics QuestaSim

Questa is Mentor Graphics' Advanced Functional Verification (AFV) tool and is an integrated platform that includes QuestaSim. QuestaSim is capable of high efficiency advanced verification of large electronic systems, and includes built-in management and debugging utilities. QuestaSim, based upon Mentor Graphics' ModelSim, seen in Figure 1, is a standards-based digital simulator capable of receiving VHDL or a variety of other languages' code as input and simulating results based on test bench waveforms.



Instance	Design unit
mult_accu8_tb	mult_accu8...
uut	mult_accu...
line__51	mult_accu8...
line__62	mult_accu8...
std_logic_unsigned	std_logic_...
std_logic_arith	std_logic_ar...
std_logic_1164	std_logic_...
standard	standard

Name	Value
tb_done	true
clk	1
rst_n	1
start	1
h	00000011
a	00000010
done	0
x	11111100

/mult_accu8_tb...	true	
/mult_accu8_tb...	1	
/mult_accu8_tb...	1	
/mult_accu8_tb...	1	
/mult_accu8_t...	00000011	00000011
/mult_accu8_t...	00000010	00000010
/mult_accu8_tb...	0	
/mult_accu8_tb/x	11111100	0000... 00... 00... 00... 00... 01... 11111100
Now	400 ns	
Cursor 1	0 ns	0 ns

```
# Loading work.mult_loop8(rtl)
 VSIM 2> run 500ns
add wave sim:/mult_accu8_tb/*
 VSIM 4> run 500ns
 VSIM 5> restart –f; run 400ns
 VSIM 6> vsim work.mult_accu8_tb
# vsim work.mult_accu8_tb
# Loading /userl/modelsim_6.2j/questasim/linux_x86_64/../std.standard
# Loading /userl/modelsim_6.2j/questasim/linux_x86_64/../ieee.std_logic_1164(body)
# Loading /userl/modelsim_6.2j/questasim/linux_x86_64/../ieee.std_logic_arith(body)
# Loading /userl/modelsim_6.2j/questasim/linux_x86_64/../ieee.std_logic_unsigned(body)
# Loading work.mult_accu8_tb(mult_accu8_tb_arch)
# Loading work.mult_accu8(structural)
# Loading work.xor_8_bit(dataflow)
# Loading work.mult_loop8(rtl)
add wave sim:/mult_accu8_tb/*
 VSIM 8> run 400ns

VSIM 9>
```

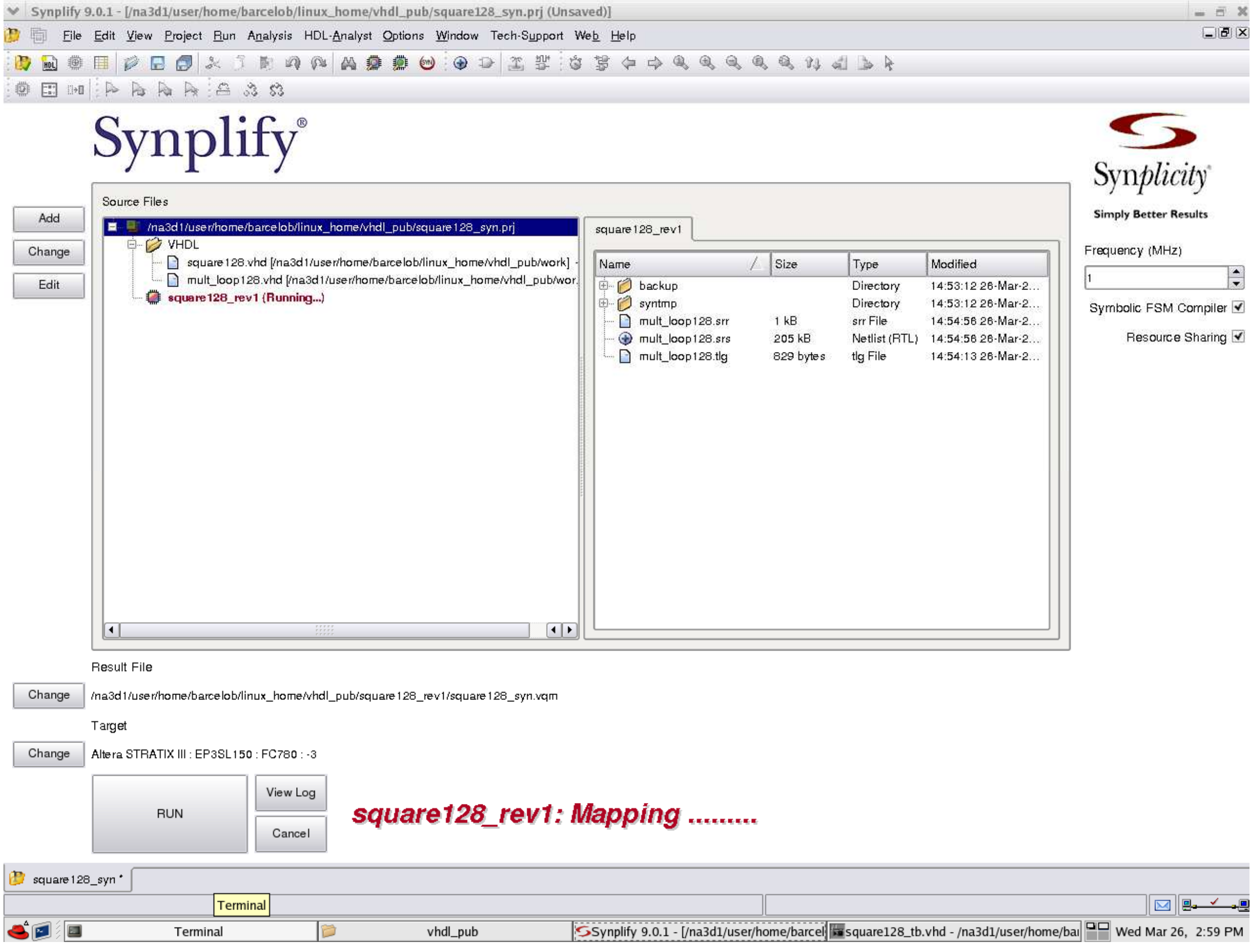
Now: 400 ns   Delta: 0   sim:/mult_accu8_tb – Limited Visibility Region

Figure 1: QuestaSim displaying simulation results for a period of 400ns

QuestaSim boasts a variety of features in addition to its primary functionality, such as low-power design verification and fast time-to-debug using assertions and a multi-abstraction debug environment [11].

2.3.2 Synplicity Synplify and Synplify Pro

Synplify is synthesis engine that is used to create FPGA designs. It takes in VHDL or Verilog code and outputs a netlist which can be optimized for a variety of FPGA vendors and packages. Synplify uses Behavior Extracting Synthesis Technology® (B.E.S.T. ™) to produce designs which are fast and highly efficient. Additionally, it is designed with a simple interface so that it is easy to use [16]. Below is a screenshot of the Synplify user interface during the mapping process of a VHDL module (Figure 2):



Synplify 9.0.1 - [/na3d1/user/home/barcelob/linux_home/vhdl_pub/square128_syn.prj (Unsaved)]

Source Files

- /na3d1/user/home/barcelob/linux_home/vhdl_pub/square128_syn.prj
  - VHDL
    - square128.vhd [/na3d1/user/home/barcelob/linux_home/vhdl_pub/work]
    - mult_loop128.vhd [/na3d1/user/home/barcelob/linux_home/vhdl_pub/wor
  - square128_rev1 (Running...)

square128_rev1

Name	Size	Type	Modified
backup		Directory	14:53:12 26-Mar-2...
syntmp		Directory	14:53:12 26-Mar-2...
mult_loop128.srr	1 kB	srr File	14:54:56 26-Mar-2...
mult_loop128.srs	205 kB	Netlist (RTL)	14:54:56 26-Mar-2...
mult_loop128.tlg	829 bytes	tlg File	14:54:13 26-Mar-2...

Frequency (MHz): 1

Symbolic FSM Compiler ☑

Resource Sharing ☑

Result File: /na3d1/user/home/barcelob/linux_home/vhdl_pub/square128_rev1/square128_syn.vqm

Target: Altera STRATIX III : EP3SL150 : FC780 : -3

square128_rev1: Mapping ....

square128_syn *

Synplify 9.0.1 - [/na3d1/user/home/barcel

square128_tb.vhd - /na3d1/user/home/ba

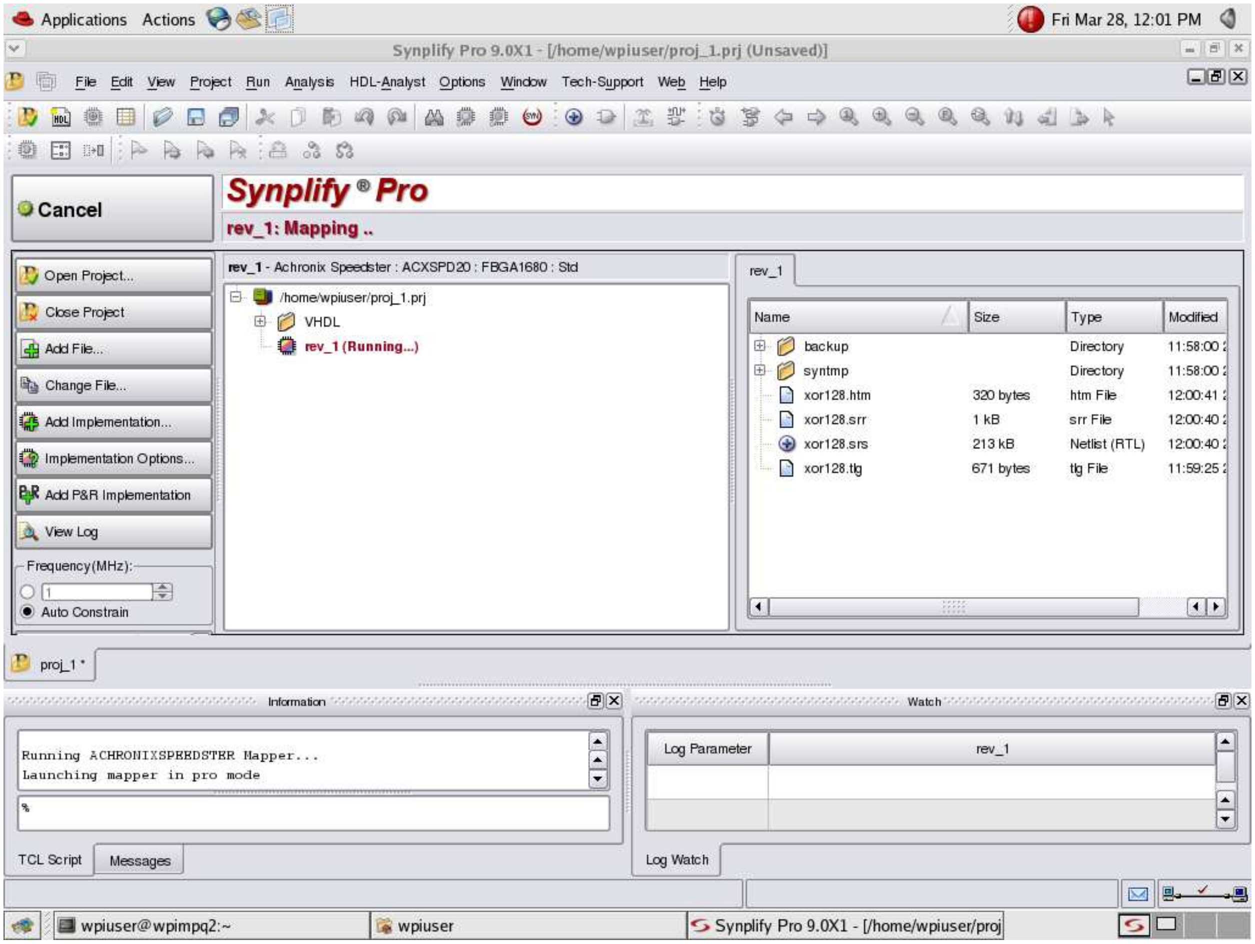
Wed Mar 26, 2:59 PM

Figure 2: Synplify mapping the *square128* VHDL module as part of the compilation process

Synplify Pro is similar in operation and use to that of Synplify, but offers better algorithms for compilation and mapping. In addition, it also improves the user interface (the Synplify Pro interface can be seen in Figure 3) and adds a great deal more options that may be used in design. This project uses both Synplify and Synplify Pro, the latter being used in situations concerning benchmarking due to the need of the auto constraining feature found within Synplify Pro.

2.3.3 Achronix CAD Environment (ACE)

The Achronix CAD Environment runs as a complementary tool to Synplicity's Synplify Pro software, seen in Figure 3, and allows for enhanced optimization techniques using Achronix's proprietary technology to decrease routing delays. This results in an overall throughput increase of the system and allows for FPGAs to run some applications at speeds greater than 1GHz. ACE, which can be seen in Figure 4, has been designed to be intentionally easy to use and while it functions on the premises of an asynchronous logic design, all input to the program is standard architecture, synchronous logic designs. This allows for current configurations to only require slight HDL modifications in order to benefit from the performance improvements ACE offers.



rev_1 - Achronix Speedster : ACXSPD20 : FBGA1680 : Std

- /home/wpiuser/proj_1.prj
  - VHDL
  - rev_1 (Running...)

Frequency(MHz):
○ 1
● Auto Constrain

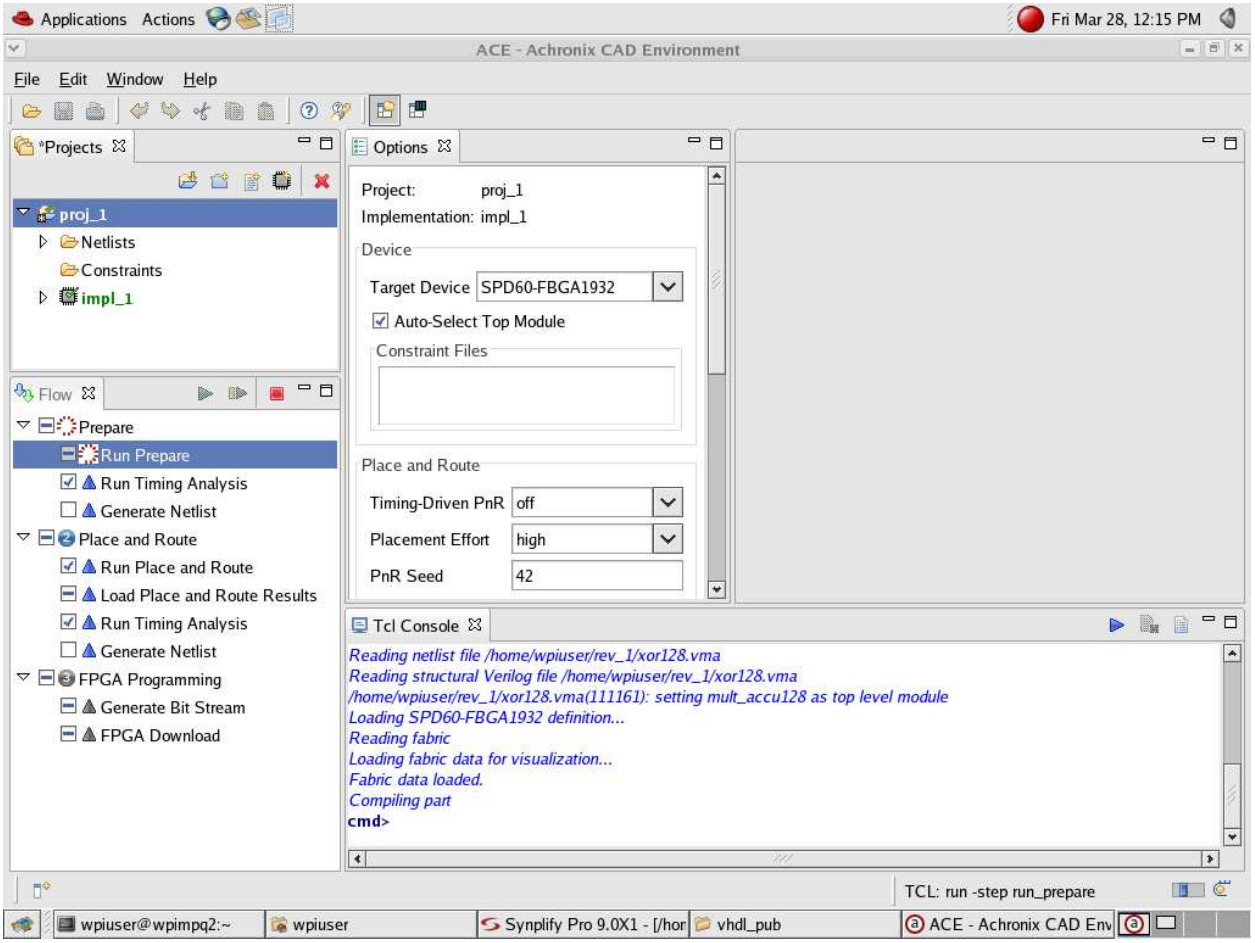
rev_1

Name	Size	Type	Modified
backup		Directory	11:58:00
syntmp		Directory	11:58:00
xor128.htm	320 bytes	htm File	12:00:41
xor128.srr	1 kB	srr File	12:00:40
xor128.srs	213 kB	Netlist (RTL)	12:00:40
xor128.tlg	671 bytes	tlg File	11:59:25

```
Running ACHRONIXSPEEDSTER Mapper...
Launching mapper in pro mode
%
```

Log Parameter	rev_1

Figure 3: Synplify Pro mapping *mult_accu128*



Project: proj_1
Implementation: impl_1

Target Device: SPD60-FBGA1932

☑ Auto-Select Top Module

Timing-Driven PnR: off

Placement Effort: high

PnR Seed: 42

```
Reading netlist file /home/wpiuser/rev_1/xor128.vma
Reading structural Verilog file /home/wpiuser/rev_1/xor128.vma
/home/wpiuser/rev_1/xor128.vma(111161): setting mult_accu128 as top level module
Loading SPD60-FBGA1932 definition...
Reading fabric
Loading fabric data for visualization...
Fabric data loaded.
Compiling part
cmd>
```

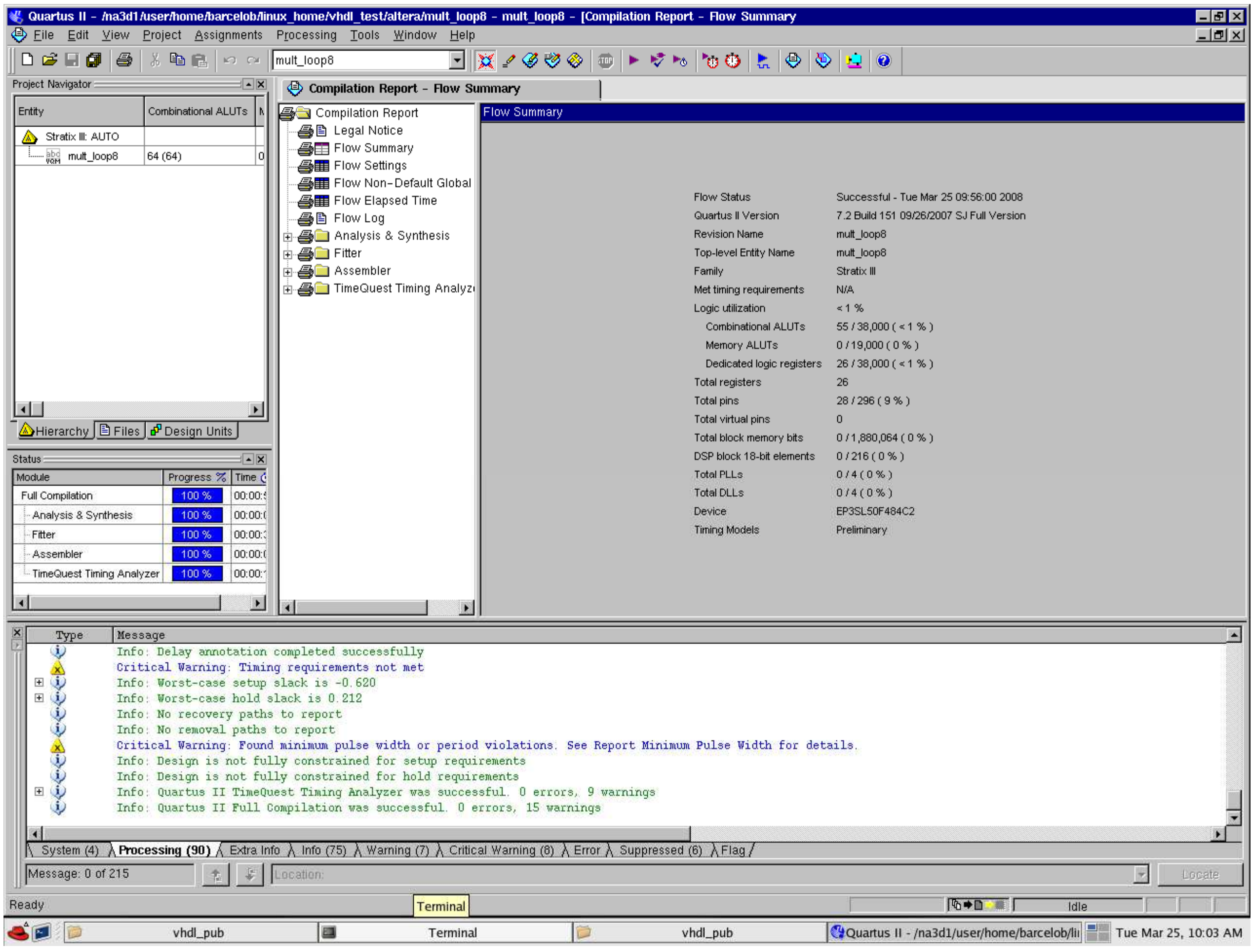
TCL: run -step run_prepare

Figure 4: Achronix CAD Environment compiling *mult_accu128*

At the time of this report's authoring, ACE is not commercially available but is scheduled to launch before the end of the year (2008) to major companies. As such, the version of ACE used in this project is only to be considered a pre-release, or beta, version of the software with some functionality not yet implemented by the Achronix software engineers.

2.3.4 Altera Quartus II

Altera's Quartus II software is a product of the Altera Corporation that provides a unified development design flow for FPGAs, structured ASICs, and CPLDs. Quartus II is capable of easily addressing problems relevant to designs such as post place-and-route design modifications. Compared to the Xilinx ISE, Quartus II provides higher benchmarks in performance with relevance to FPGA and CPLD designs. Quartus II also provides tools such as TimeQuest and PowerPlay that assist in timing analysis and power analysis, respectively, as well as a pin planner feature to be used in I/O pin assignment [15]. Quartus II's interface can be seen below in Figure 5:



Entity	Combinational ALUTs	M
Stratix III: AUTO		
mult_loop8	64 (64)	0

Module	Progress %	Time
Full Compilation	100 %	00:00:
Analysis & Synthesis	100 %	00:00:
Fitter	100 %	00:00:
Assembler	100 %	00:00:
TimeQuest Timing Analyzer	100 %	00:00:

Flow Summary

Flow Status	Successful - Tue Mar 25 09:56:00 2008
Quartus II Version	7.2 Build 151 09/26/2007 SJ Full Version
Revision Name	mult_loop8
Top-level Entity Name	mult_loop8
Family	Stratix III
Met timing requirements	N/A
Logic utilization	< 1 %
Combinational ALUTs	55 / 38,000 (< 1 %)
Memory ALUTs	0 / 19,000 (0 %)
Dedicated logic registers	26 / 38,000 (< 1 %)
Total registers	26
Total pins	28 / 296 (9 %)
Total virtual pins	0
Total block memory bits	0 / 1,880,064 (0 %)
DSP block 18-bit elements	0 / 216 (0 %)
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)
Device	EP3SL50F484C2
Timing Models	Preliminary

```
Type    Message
        Info: Delay annotation completed successfully
        Critical Warning: Timing requirements not met
        Info: Worst-case setup slack is -0.620
        Info: Worst-case hold slack is 0.212
        Info: No recovery paths to report
        Info: No removal paths to report
        Critical Warning: Found minimum pulse width or period violations. See Report Minimum Pulse Width for details.
        Info: Design is not fully constrained for setup requirements
        Info: Design is not fully constrained for hold requirements
        Info: Quartus II TimeQuest Timing Analyzer was successful. 0 errors, 9 warnings
        Info: Quartus II Full Compilation was successful. 0 errors, 15 warnings
```

System (4) | Processing (90) | Extra Info | Info (75) | Warning (7) | Critical Warning (8) | Error | Suppressed (6) | Flag

Message: 0 of 215

Location:

Figure 5: Example of Quartus II flow summary after compilation

2.4 General Dynamics C4 Systems (GDC4S)

"General Dynamics C4 Systems is a subsidiary of the General Dynamics Corporation located in Falls Church, Virginia. C4 Systems is part of the General Dynamics Information Systems and Technology group that consists of four business units: Advanced Information Systems, C4 Systems, United Kingdom Limited and Information Technology. General Dynamics C4 Systems is a leading provider of network-centric solutions. Their leadership credentials come from applying world-class capabilities to create high-value, low risk solutions for use on land, at or under the sea, in the air and in space. Based in Scottsdale, Arizona, General Dynamics C4 Systems employs approximately 11,000 people worldwide and is focused on the development, design, manufacturing and integration of secure communication, information and technology solutions." [5]

General Dynamics C4 Systems sponsored this MQP and its research by providing the project team with office space, software and hardware necessary to complete this project as well as some of its employees' time. While General Dynamics C4 Systems is a government contractor and primarily deals with classified materials, this work was unclassified and is subject to a non-disclosure agreement.

2.5 Achronix Semiconductor Corporation

"Achronix Semiconductor is a privately owned fabless corporation based in San Jose, CA. Achronix markets the world's fastest FPGAs capable of running at speeds of up to 2GHz, in throughput, using their unique, patented technology. Achronix FPGAs are targeted at a wide variety or markets ranging from medical to military and products are manufactured to different specifications, the highest of which requires their products be operable from -260°C to +130°C." [12]

Achronix provided this MQP with the necessary software to implement functional designs much faster than standard FPGA designs due to their software utilizing Achronix's unique technology. Achronix also provided training from an Achronix field applications engineer as well as documentation and tutorials not otherwise available.

2.6 Advanced Encryption Standard (AES)

Following the increasing number of simple exploits to the Data Encryption Standard (DES), the United States government needed a new encryption standard that could be trusted for general, unclassified materials. On May 26, 2002 [6], AES became that standard, replacing DES for all but legacy systems. Designed by Vincent Rijmen and Joan Daemen, the Rijndael algorithm was chosen by the National Institute of Standards and Technology (NIST) to be used for AES. AES is now widespread and is extremely common in both software and hardware applications, utilizing a 128-bit block structure with key sizes of 128, 192, and 256 bit forms [7]. This larger key size compares with the DES key sizes of 56 bits and allows for 10^{21} additional keys, rendering it extremely difficult to search for encryption keys using brute force methods [8].

AES systems remain, to date, unbroken. However, implementation-related attacks such as side-channel attacks may compromise insecure systems. Side-channel attacks do not rely on the encryption algorithm but rely on physical proximity and external monitoring of power, noise and other factors related to the system in order to formulate an attack [9]. Often based on timing information or transmission of leaked electromagnetic data, these attacks may require knowledge about the internal operation of the system under attack. Side-channel attacks may be averted on an AES by shielding the hardware components or implementing stricter security guidelines for physically accessing the computer system [10].

2.7 Finite Fields

Finite fields are useful mathematical structures found in a number of cryptographic primitives, most notably, the Advanced Encryption Standard (AES) and Elliptic Curve Cryptography (ECC). The subject of finite fields is an intensively studied field of mathematics with many constituents. Finite fields of order p^n, where n is a positive integer, are generally written as *GF*(p^n); GF stands for Galois field, named so after the famous mathematician who introduced finite fields of the order p^n; Galois fields are most useful in encryption when of the order *GF*(2^n). In *GF(2)*, addition is equivalent to logical XOR and multiplication is equivalent to logical AND; addition and subtraction are also equivalent to *mod 2*. Polynomial arithmetic in *GF*(2^n) is often used in encryption standards and is the basis for AES. AES uses the finite field *GF*(2^8) with the following irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

Equation 1: AES irreducible polynomial

2.8 Galois/Counter Mode of Operation (GCM)

By definition, the Galois/Counter Mode is a block cipher mode of operation that uses universal hashing over a binary Galois field whose purpose is to provide authenticated encryption. It is implementable in both hardware and software and is able to achieve high speeds at a low area overhead while maintaining low latency and, in software, perform significantly faster using table-driven finite field operations.

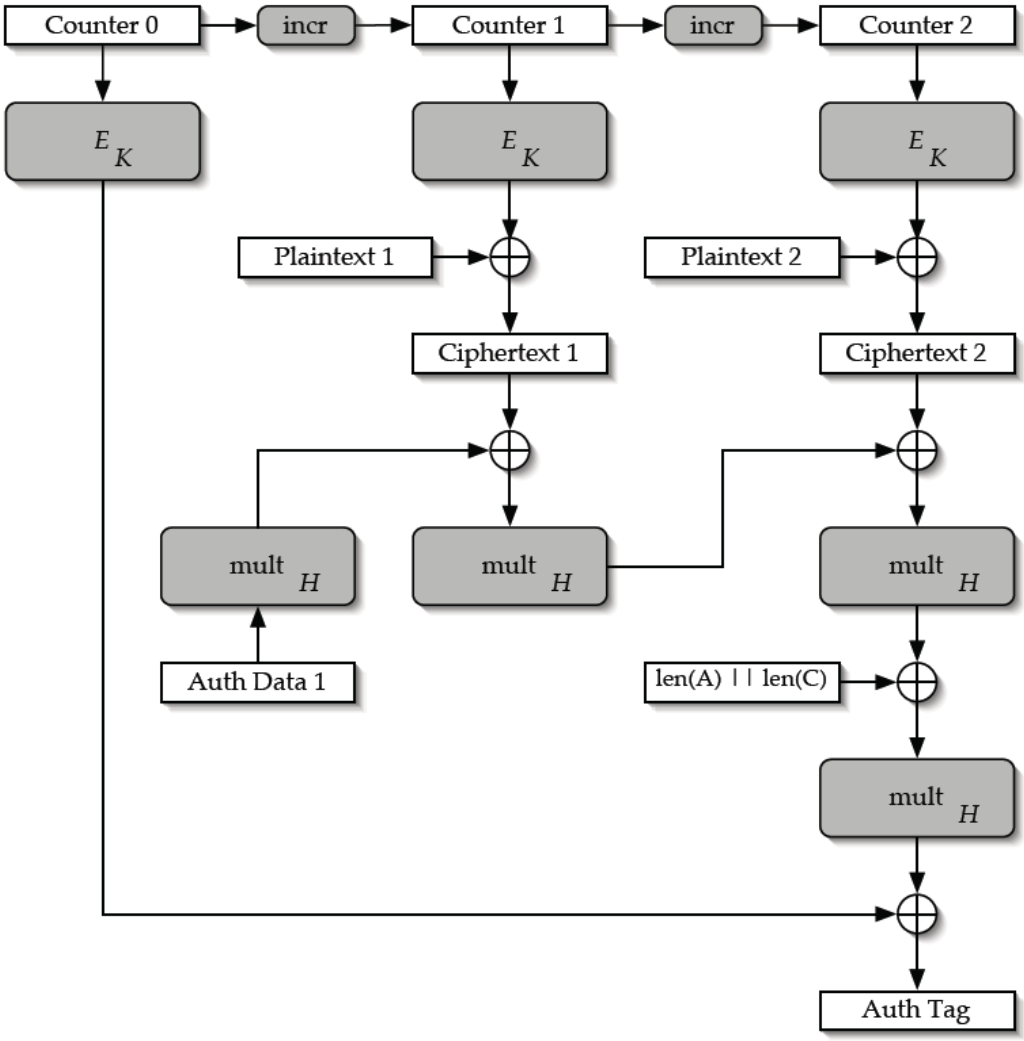


Figure 6: The GCM authenticated encryption operation using a simplified, single authenticated data block, two plaintext blocks model [3]

What makes GCM appealing is its ability to provide authenticated encryption at ten gigabits per second and higher (10Gbps+). GCM is also an open-license mode of operation and thus no royalties need be paid or licenses bought in order to implement and redistribute hardware or software operating with GCM. The ability to parallelize and pipeline systems using GCM without a loss of performance makes it more attractive to implement as the regular counter mode requires integer arithmetic and carry chains. What makes GCM ultimately unique is its functionality as a suitable standard message authentication algorithm. CBC-MAC, CCM, EAX, OMAC, OCB, CWC, and Counter Mode all suffer from inadequacies that make them less apposite choices than GCM. Of note, GCM is also capable of operating as a standalone Message Authentication Code (MAC) generator when there is no data to encrypt, as well as operating as an incremental MAC. The encryption techniques will be discussed in detail later, including the GHASH and authenticated encryption operation algorithms. [3]

3 Design Requirements

3.1 Initial Design

The initial designs for the multiplier used for GHASH were provided to this project by Mark Krumpoch, a GDC4S employee who wrote Galois field multipliers for unrelated projects and allowed us to utilize his expertly coded HDL designs as a basis. Three different versions of his multipliers were made available: the first which ran in one clock cycle, the second which required four clock cycles, and the third which required eight clock cycles. Given the software design goal that we implement the fastest GHASH function and, by association, multiplier into the tools that would be used for FPGA design, we used the one clock cycle multiplier as a basis.

The one clock cycle multiplier to be used in the GHASH implementation takes in two 128-bit inputs and performs a Galois field multiplication of the two polynomials, resulting in a single 128-bit output. This multiplier is designed synchronously and uses start, reset, and done ports which may be removed for implementation. The code can be safely modified for the reason that start is only used to initialize temporary variables used if the multiplier is not running; reset is also used similarly as top level code for the portion of code responsible for checking if the multiplier is running or not (our design is always running the multiplier, effectively); done is only used to pass a 1 or 0 to another component signifying the multiplication is finished, an unnecessary function in our design. While earlier sections of this project experimented with this multiplier on a smaller scale, it should be noted that modifying this multiplier to work with *n* bits is not only possible (given capable hardware) but also very simple.

To establish a control in this project, we followed the typical experimental process and followed our supervisor's instructions to benchmark an optimized GHASH multiplier versus the standard, synchronous design created by Krumpoch. As such, the *mult_loop128* and its variants (*mult_loop16* and *mult_loop32*) were tested in Synplify Pro and Quartus II/ACE without any modifications to the original code, but could not be included in this document as they are property of General Dynamics C4 Systems. It is important to note that the code header includes our modifications regarding the method in which the AES polynomial is declared, this is common to all the designs and variants requiring this and is simply an improved method for readability over bitwise declaration for 128 bits.

3.2 Transition from Synchronous to Asynchronous Design

While asynchronous designs are generally more difficult to create than synchronous designs, they have several advantages over synchronous designs in speed and power critical applications. Asynchronous designs can run in average time, as opposed to worst case time. Different regions can also run at different effective clock rates, so that slower elements don't slow down the entire design. Additionally, asynchronous designs also tend to use less power, as idle unclocked components consume very little power compared to idle components in a clocked, synchronous design. [18]

In a training seminar with Achronix field applications engineer Scott E. Norrholm, Mr. Norrholm explained that to switch a synchronous design to an asynchronous model, ACE uses a technique known as pico-piping to increase performance. The ACE software creates data tokens and form pipelines along the data flow paths in the synchronous FPGA design. By placing pico-pipes at roughly evenly spaced intervals it enables the design to run at a much higher speed because the routing delays are shorter. Additionally, since components are not clocked, idle components consume almost no power, which greatly reduces power consumption for sleep modes. This implementation, however, has some important limitations due to this pipelined design. Any design which includes loops or feedback will be limited by the time it takes a data token to travel around the loop, as no more data can flow into the loop until the first data token has completed its circuit. Additionally, it is important to make sure that there are roughly the same number of pipeline stages when two or more data tokens re-converge, since whichever token arrives first will have to wait for its companion tokens before going through that logical component, causing a backup of the tokens behind it.

4 Implementation

4.1 Design Intent

This project began with the ambition of optimizing and implementing the GHASH algorithm on an FPGA board capable of running it at speeds of over 1GHz. The methodology behind accomplishing this was to use Achronix's proprietary routing technology the increase system throughput to that greater than 1GHz. This speed would render regular synchronous design nearly impossible, as well as challenging other components that need to be clocked carefully. As a result, the FPGA design was to be fully asynchronous. Being asynchronous requires that no component of the design use a clock to synchronize the system as it passes signals and data to other locations in the system.

Accomplishing the initial objective quickly became a matter of time and not practice, for which the project's goals changed to focus on the slowest part of the GHASH implementation – the multiplier. The multiplier component became the focus of this project and its optimization to run at speeds beyond 1GHz became the goal. Below is the GHASH algorithm in generalized form, which excludes special cases found in the GCM specification that do not apply to this project:

$$X_i = \begin{cases} 0 & \text{for i=0} \\ (X_{i-1} \oplus A_i) \cdot H & \text{for i=1,...m} \\ (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for i=m+1,...m+n} \\ (X_{m+n} \oplus (length(A)||length(C))) \cdot H & \text{for i=m+n+1} \end{cases}$$

Equation 2: GHASH message stream compression algorithm [2]

To accomplish this goal, we attempted to implement a fully asynchronous multiplier that is capable of being synthesized and simulated at speeds greater than 1GHz using squaring methods explained in their entirety in [17]. The basis of the new squaring algorithm is that one can transform a square operation, in GF(2^m), into a multiplication by a constant and a sum. This not only reduces registers, but also reliance on multiple clock cycles. This, in sequence with the Achronix proprietary optimization methods, should yield a sufficiently fast multiplier for use in GHASH.

4.2 Experimental Coding Exercises

Prior to the design of the fully asynchronous 128-bit multiplier and squarer, several smaller steps were taken to validate the algorithms we researched and employ the multiply module we received from GDC4S in a way in which we could test them by hand. Since 128-bit multiplication in a finite field is very difficult to do by hand, we restructured the multiply module we received and built supplemental modules to test with 8 bit components, instead of 128-bits. For these tests, we made an 8 bit XOR, 8 bit register, and an 8 bit GF(2^8) multiply based on the multiply module we received. First, we verified the output of the multiplier by comparing its outputs to hand calculations of the expected values that a multiplier in a GF(2^8) field would give. Second, we built an 8-bit XOR and an 8-bit register and, with the multiply, constructed a multiply accumulate circuit. This design, shown below in Figure 7, multiplies incoming values with the value already stored in its output register.

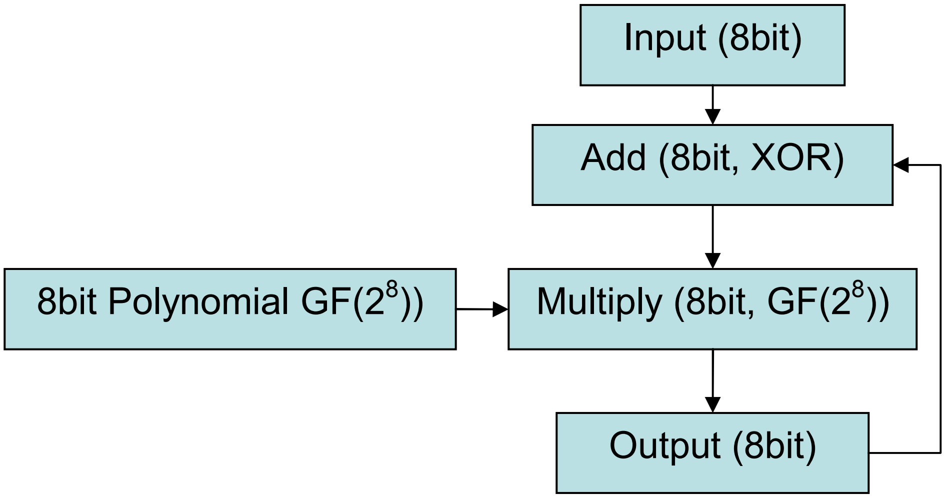


Figure 7: ***mult_loop8*** **operational block diagram**

This design allowed us to test the multiplier in a mode of operation that would be similar to how it would function in a traditional GHASH implementation. Following the validation of these modules and their operation in QuestaSim, we then tested our environment by putting this small design through Synplify and Quartus to verify that those tools were also working properly.

4.3 VHDL Design – Top and Module Levels

The processes for the creation of our VHDL design began with the expansion of our 8-bit modules covered in section 4.2 Experimental Coding Exercises, to 128-bits. In addition to increasing the size of these modules, the 128-bit multiplier was streamlined and trimmed of some excess control logic that was not necessary for our implementation. After constructing the 128-bit versions of our modules in VHDL, we tested them in QuestaSim with test vectors to verify their output. We then generated netlists with Synplify and applied the netlists to a Stratix III EP3SL150F1152C2ES (an Altera FPGA) to ensure that the modules were of an appropriate size with respect to LUT and register usage.

4.4 VHDL Design – Component Level

4.4.1 128-bit XOR Logic

The most basic module we created as a part of this project was the 128-bit XOR. It takes in two, 128-bit inputs, performs a bitwise XOR, and then sends the result to a 128-bit output.

4.4.2 128-bit Register

Another basic module we created for this project was a 128-bit register. It is composed of 128 D flip-flops, allowing it to take in, store, or output a 128-bit value. It has one 128-bit input and one 128-bit output. It also has two control signals, load enable and reset, in addition to a clock signal. When load enable is toggled, values are read into the register on the input side. When reset is toggled, the value stored in the register is reset to 0. There is no output enable/disable, this module always outputs the value stored in it.

4.4.3 128-bit Multiply

The simplest 128-bit multiply is the once clock cycle version, mult_loop128. This module takes in two 128-bit values and outputs there product in the finite field GF(2^{128}). It performs this computation in one clock cycle by computing all of the shifts simultaneously. This means, however, that it is quite large in terms of LUT usage, and the register to register delay is also large, limiting the maximum frequency with which it can be run. On a Stratix III, it took up 8672 LUTs and ran at a maximum speed of 60MHz, which results in a minimum compute time of 16.7ns.

Mult_loop32 performs the same operation as mult_loop128 except that it computes it over the course of four clock cycles, using the same set of components four times. This indicates that it is significantly slower in terms of total time to compute, but also takes up much less space in terms of LUTs and can be run at a higher clock rate. On a Stratix III, the multiplier occupied 2375 LUTs and ran at a maximum speed of 164MHz, which results in a minimum compute time of 24.4ns.

Mult_loop16 is an extension of mult_loop32, in that it runs in eight clock cycles instead of four. It is the smallest of the three multipliers, but it is also the slowest in terms of time to compute. It can, however, be run at the highest clock rate. On a Stratix III, it took up 1308 LUTs and ran at a maximum speed of 276MHz, which results in a minimum compute time of 29.0ns.

4.4.4 128-bit Squarer

The 128-bit multiply is large and, in addition to a multiply for 128-bit, we also needed a squarer. If one just uses the basic multiplier and ties the inputs together, a moderate reduction in size can be observed. However, we knew from the algorithm proposed in [17] that the squaring function could be done smaller and more efficiently. From an arithmetical discussion with G. Orlando, the equations used to describe the optimized squaring algorithm are derived as follows in Equation 3:

$$A = \sum_{i=0}^{m-1} a_i x^i$$

$$A^2 = C''\cdot C' + B$$

$$= (x^7 + x^2 + x + 1)(\sum a_{j+\frac{m}{2}} x^{2j}) + \sum_{i=0}^{i=\frac{m}{2}-1} a_i x^{2i}$$

Equation 3: Mathematical representation of optimized squaring algorithm for AES GF(2^{128})

This module takes the input, spreads the input out by placing an extra zero in between each bit, multiplies the top half by the GF(2^{128}) polynomial, then adds the bottom half to the result of that multiplication. [17] Because the GF(2^{128}) polynomial contains almost all zeros except for four bits, most of the multiplier is simplified away in the synthesis process performed by Synplify. By creating a module specifically for squaring, we reduced the size of the 128-bit squarer from over 5000 look-up tables (LUTs) down to just 192 LUTs. An added benefit of this module, in addition to its reduction in size, is that it can run much faster than a normal multiplier. On a Stratix III, it took up 192 LUTs and ran at a maximum speed of 480MHz, which results in a minimum latency of 2.1ns.

4.4.5 128-bit Bit Spreader

The spread128 component is a subcomponent of square128 that spreads out a 128-bit input into two 128-bit outputs by putting a zero between each input bit to get to 256 bits and then dividing the upper half and lower half to achieve two 128-bit outputs.

4.5 Optimizing VHDL Design for Highest Throughput Performance

4.5.1 Efficient Squaring Algorithm

There are more efficient ways to square numbers in GF(2^{128}) than to simply run the input to both ports of a GF(2^{128}) multiplier. Below is an example of one of these techniques, originally outlined by Orlando in [17].

The first step in the algorithm, shown in Equation 4, is to rewrite the square into a sum squared, moded by the finite field polynomial.

$$A^2 \equiv (\sum_{i=0}^{m-1} a_i \alpha^i)^2 \bmod F(\alpha)$$

Equation 4: Transform of a square operation into a sum, pt. I [17]

The next step, as presented in Equation 5: Transform of a square operation into a sum, pt. II [17], is to divide the terms into two halves, one for the lower set of bits and one for the upper set of bits. In addition, the size of each half is doubled by inserting zeros in between each input bit.

$$\equiv \sum_{i=0}^{m-1} a_i \alpha^{2i} \bmod F(\alpha)$$

$$\equiv \sum_{i=\lceil m/2 \rceil}^{m-1} a_i \alpha^{2i} \bmod F(\alpha) + \sum_{i=0}^{\lceil m/2 \rceil - 1} a_i \alpha^{2i}$$

$$A^2 \equiv A' B' \bmod F(\alpha) + C'$$

Equation 5: Transform of a square operation into a sum, pt. II [17]

The final result, as shown in Equation 6, Equation 7 and Equation 8, is that the square is the sum of the top half, B’, multiplied by the polynomial for the finite field one is operating in, and that of the bottom half, C’.

$$A' = \sum_{i=0}^{\lfloor m/2 \rfloor - 1} a_i +_{\lceil m/2 \rceil} \alpha^{2i}$$

Equation 6: Transform of a square operation into a sum, pt. III, Definition of A’ [17]

$$B' = \alpha^{2\lceil m/2 \rceil} \bmod F(\alpha)$$

Equation 7: Transform of a square operation into a sum, pt. IV, Definition of B’ [17]

$$C' = \sum_{i=0}^{\lceil m/2 \rceil - 1} a_i \alpha^{2i}$$

Equation 8: Transform of a square operation into a sum, pt. V, Definition of C’ [17]

This process shows that we are multiplying by a constant value, the finite field polynomial, and that value contains mostly zeros, which allows a large simplification of the logic. This not only greatly decreases the size of the logic required to square a number, but it also increases the speed, as several stages of the multiplier get synthesized away.

4.5.2 Making the Single Clock Cycle Multiplier Asynchronous

To enable the multiplier to be pipelined more effectively by ACE, it was important to convert it to be fully asynchronous. To do this, all signals other than the two inputs and the one output were removed. Since there is no start or reset control, the multiplier outputs garbage data for the first few clock cycles, but this data will be ignored by the upper level module that uses this multiplier. To drive the multiplier, the inputs x and y replaced the clock in the sensitivity list of the process that encompasses the GF(2^{128}) multiplier. This way, the multiply calculation is computed whenever the inputs change, as opposed to on the transition of a clock, making the multiplier fully asynchronous. Now that the multiplier was implemented asynchronously, ACE was able to optimize it to a much higher level. ACE was able to use six extra pipelining stages, as described in 2.3.3 Achronix CAD Environment (ACE), which resulted in over a 300% performance gain in speed, in comparison to the version that we implemented for a Stratix III using Quartus II.

4.5.3 Making the Efficient Squarer Asynchronous

In addition to making the multiplier asynchronous, the efficient squarer, described in section 4.5.1 Efficient Squaring Algorithm, was also made asynchronous. This was done by using the same techniques implemented on the asynchronous multiplier. The clock was removed in both the XOR and the spreader modules and the sensitivity lists were changed to reflect the inputs rather than the clocks. This enabled the Achronix chip to clock up extremely high, almost reaching 1GHz, which is more than twice as fast as the maximum speed achievable on a Stratix III.

5 Synthesis, Testing and Results

5.1 Synthesis and Testing Procedure

By utilizing the various aforementioned programs, it is possible to obtain results that accurately represent performance for a given module or VHDL design. The metric used in this project is the clock setup time which is then converted to megahertz or gigahertz simply as

$$\frac{1}{t_{ns}} = Clk_{MHz}$$

Equation 9: Formula used to determine maximum clock speed of a system or module.

Note that t is the "clock setup time" in nanoseconds

Due to the nature of programs such as Synplify and Quartus, this process can be repeated by the tester numerous times unless a procedure is defined that will stop the tester from continuing to tweak inputs to the programs to give the highest clock speed. When running Synplify, the user must enter a clock speed which determines Synplify's design optimization, directly affecting how Synplify maps the design. This information makes it difficult to establish a benchmark for a variety of units under testing (UUTs) and it was decided that Synplify Pro would prove a more useful tool for comparison testing. Synplify Pro allows the user to instruct the program to auto constrain the design so that the program decides what clock speed is best to optimize for. Synplify Pro also utilizes better mapping algorithms and provides for a better comparison when compared to the Achronix testing results. The outputted .vma file (virtual memory area descriptor) is then passed to Quartus II, which can synthesize the design for a given FPGA. The FPGA chosen for Synplify and Quartus II designs is the Altera Stratix III EP3SL150, package FC1152 with speed rating equivalent to -2 (EP3SL150F1152C2ES). Before Quartus II runs it synthesis process, a system clock speed must be set. This clock speed, though it can be any value, is not capable of being auto constrained and the user must decide what system clock speed is optimal for system performance. For this project, a system clock speed of 1GHz was used within Quartus II. While this clock speed was far too high for a system to operate at, Quartus II outputs the "actual time" or, more appropriately, the maximum clock speed the system could function at.

For the Achronix CAD Environment tests, the procedure is simplified. Input the appropriate HDL files into Synplify Pro with the selected FPGA being the Achronix Speedster ACXSPD60, package FBGA1680 and speed equivalent to "Std." After this step, open ACE and pass the appropriate .vma file and ensure that the target device is set to SPD60-FBGA1932 with "Timing-Driven PnR" off, "Placement Effort" on high, and "PnR Seed" set to 42. Then run "Prepare" and it will generate timing analysis results as well as place and route and output to a plaintext log file which can be viewed to find the max system clock speed after Achronix proprietary optimization.

5.2 Results

The primary goal of this project was to evaluate the Achronix technology and its feasibility in a real-world market and application. Compiling and synthesizing different designs, we have produced results that offer a good comparison between standard FPGA technology, represented by the Altera Stratix III, and Achronix technology, represented by the Achronix Speedster FPGA. The three metrics to be discussed in this section were all generated simultaneously by either Quartus II or ACE and are the number of LUTs, number of registers and the system clock speed. While graphs will be used to assist in explanation and comparison, the tables in Figure 8 and Figure 9 contain the experimental data we obtained that corroborates with the proceeding figures.

Component	Target	6-Input ALUTs	Registers	Speed (MHz)
mult_loop128_reg	Stratix III:EP3SL150F1152C2ES	8672	1254	60
mult_loop32_reg	Stratix III:EP3SL150F1152C2ES	2375	1050	164
mult_loop16_reg	Stratix III:EP3SL150F1152C2ES	1308	909	276
square128_n_reg	Stratix III:EP3SL150F1152C2ES	192	448	480
mult_loop128_f_reg	Stratix III:EP3SL150F1152C2ES	8676	1576	58
mult_32b_f_reg	Stratix III:EP3SL150F1152C2ES	609	165	284
square128_reg	Stratix III:EP3SL150F1152C2ES	5638	840	57
square128_nf_reg	Stratix III:EP3SL150F1152C2ES	128	261	480

Figure 8: Table of Stratix III experimental results obtained from Quartus II testing

Component	Target	4-Input LUTs	Registers	Speed (MHz)
mult_loop128_reg	Achronix: SPD60-FBGA1932	Fail	Fail	Fail
mult_loop32_reg	Achronix: SPD60-FBGA1932	4261	912	153
mult_loop16_reg	Achronix: SPD60-FBGA1932	2640	915	161
square128_n_reg	Achronix: SPD60-FBGA1932	448	448	595
mult_loop128_f_reg	Achronix: SPD60-FBGA1932	13688	1101	225
mult_32b_f_reg	Achronix: SPD60-FBGA1932	927	117	295
square128_reg	Achronix: SPD60-FBGA1932	Not Tested	Not Tested	Not Tested
square128_nf_reg	Achronix: SPD60-FBGA1932	257	256	926

Figure 9: Table of Speedster experimental results obtained from ACE testing

Below, in Figure 10, is a graph of the number of LUTs, separated by design scenario. Before analyzing this particular metric, it is important to note that the Stratix III board uses 6-input LUTs that are a combination of two 4-input LUTs that are registered and function as a 6-input LUT. This is in comparison to the Achronix Speedster FPGA that uses more typical 4-input LUTs in its design. With this stated, the graph clearly represents that the Achronix Speedster requires more LUTs for all of the designs. LUTs serve as an important metric as FPGAs are only capable of supporting so many LUTs before the design exceeds the board’s physical capacity. With more complex designs, LUTs become a scarce resource that need to be balanced properly with the size of the FPGA.

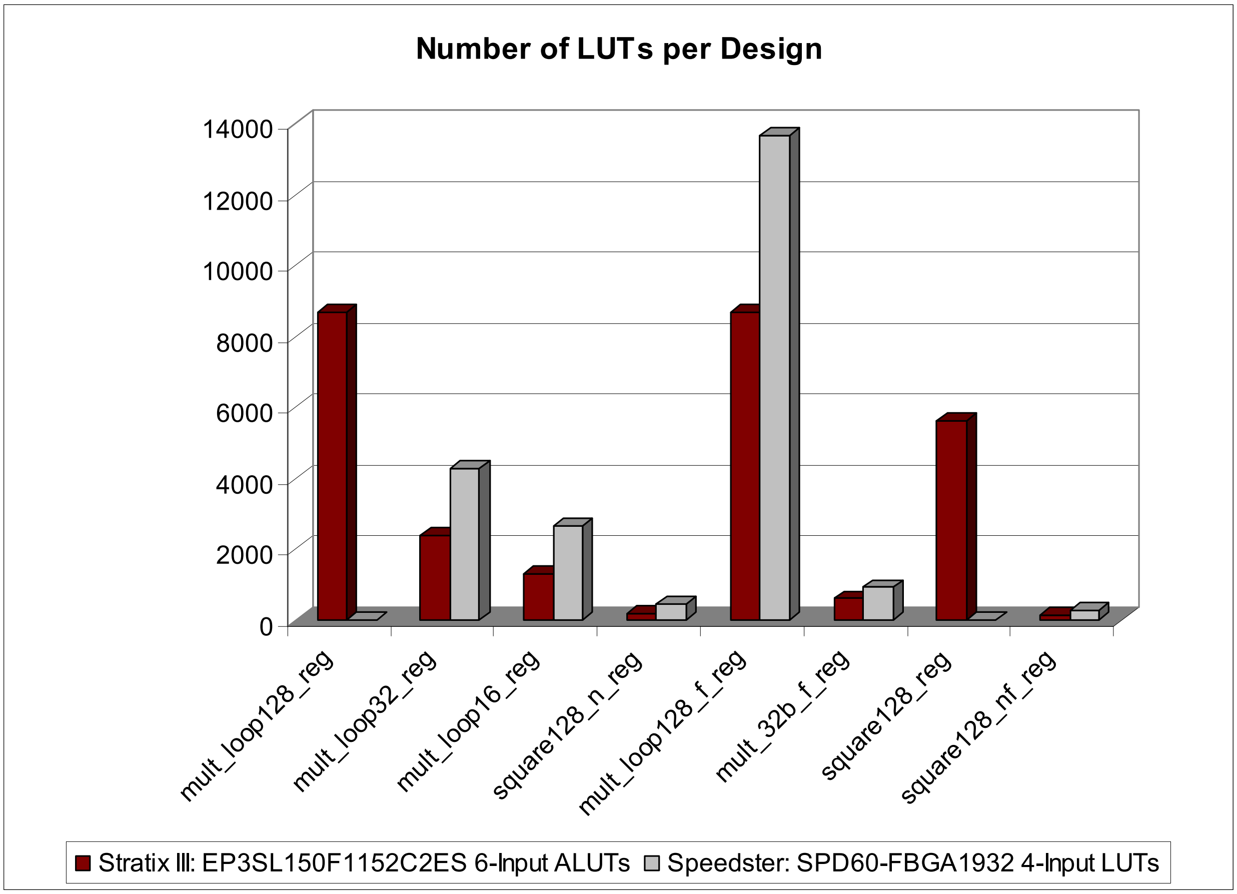


Figure 10: Graphical representation comparing number of LUTs between the two FPGAs

The next metric to be discussed is the number of registers for each design which is depicted, again graphically, in Figure 11. Registers are much like LUTs in that they are both a physical limitation of the FPGA board. Therefore, designs that require too many registers, even if offering better speeds, will be discarded if the rest of the design will not fit onto the FPGA. Here, the results are mixed with two of the tests exhibiting more required registers for the Achronix FPGA, and the remaining tests requiring more registers with the Altera FPGA. As stated before, the ACE software works best with designs that do not possess feedback and a positive result for the ACE software is found in the *mult_loop128_f_reg* numbers which show that the 128-bit multiplier design, when registered and possessing no feedback, requires 1101 registers, fewer registers than the original 128-bit multiplier design on both the Altera (1254 registers) and Achronix (1748 registers) boards.

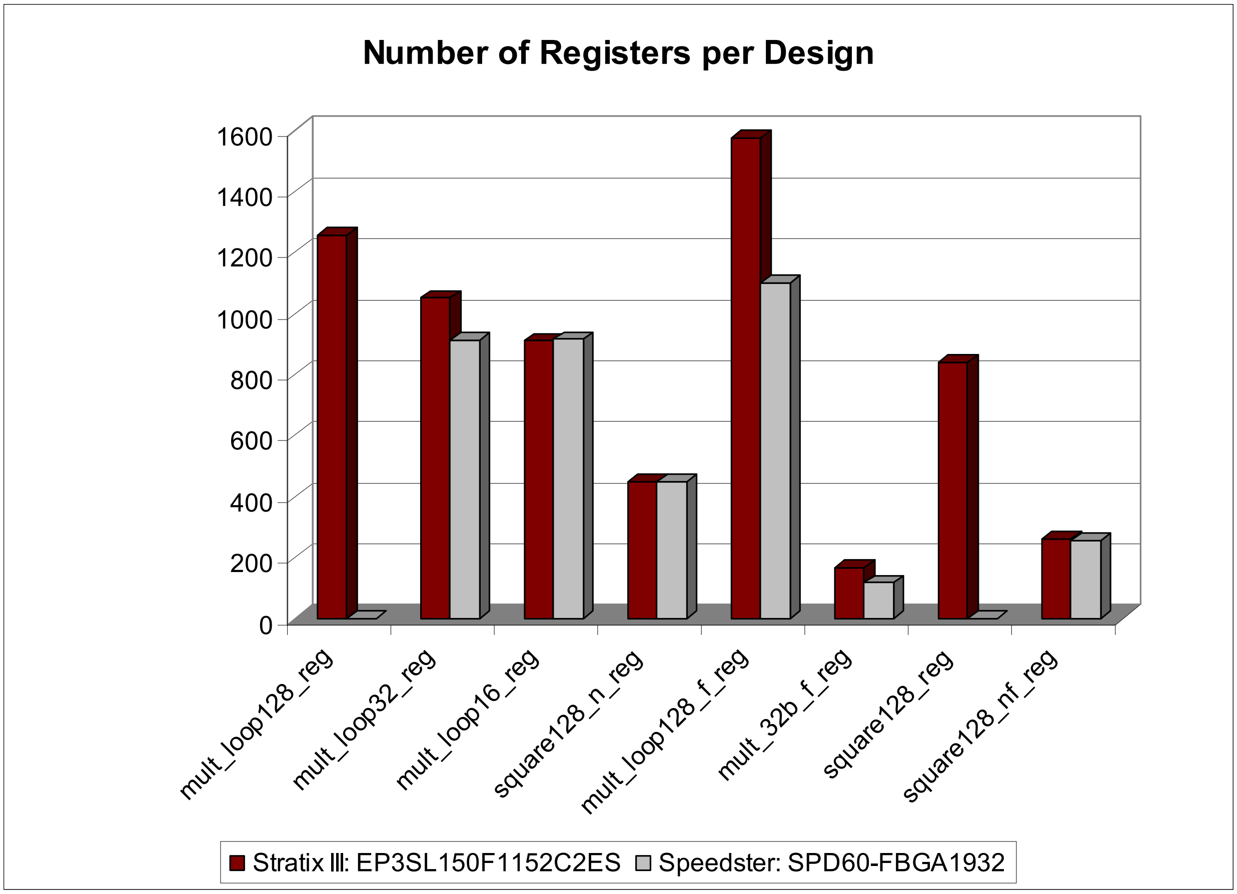


Figure 11: Graphical representation comparing number of registers between the two FPGAs

The final metric, depicted in Figure 12, is of great importance. The selling point of all Achronix's technology is that of speed for design throughput optimization. Figure 12 shows this specifically in system clock speed, measured in MHz, between the same designs on both the Altera and Achronix boards. However, though the results look as if ACE does not offer improvement for most designs, these designs are the most reliant on feedback and, as such, were not expected to perform better than the Altera designs. Achronix specifically noted this, and this project does not seek to point out any deficiencies, rather, the tests with feedback were performed so that some control points could be established. These control points represent unmodified, existing synchronous designs.

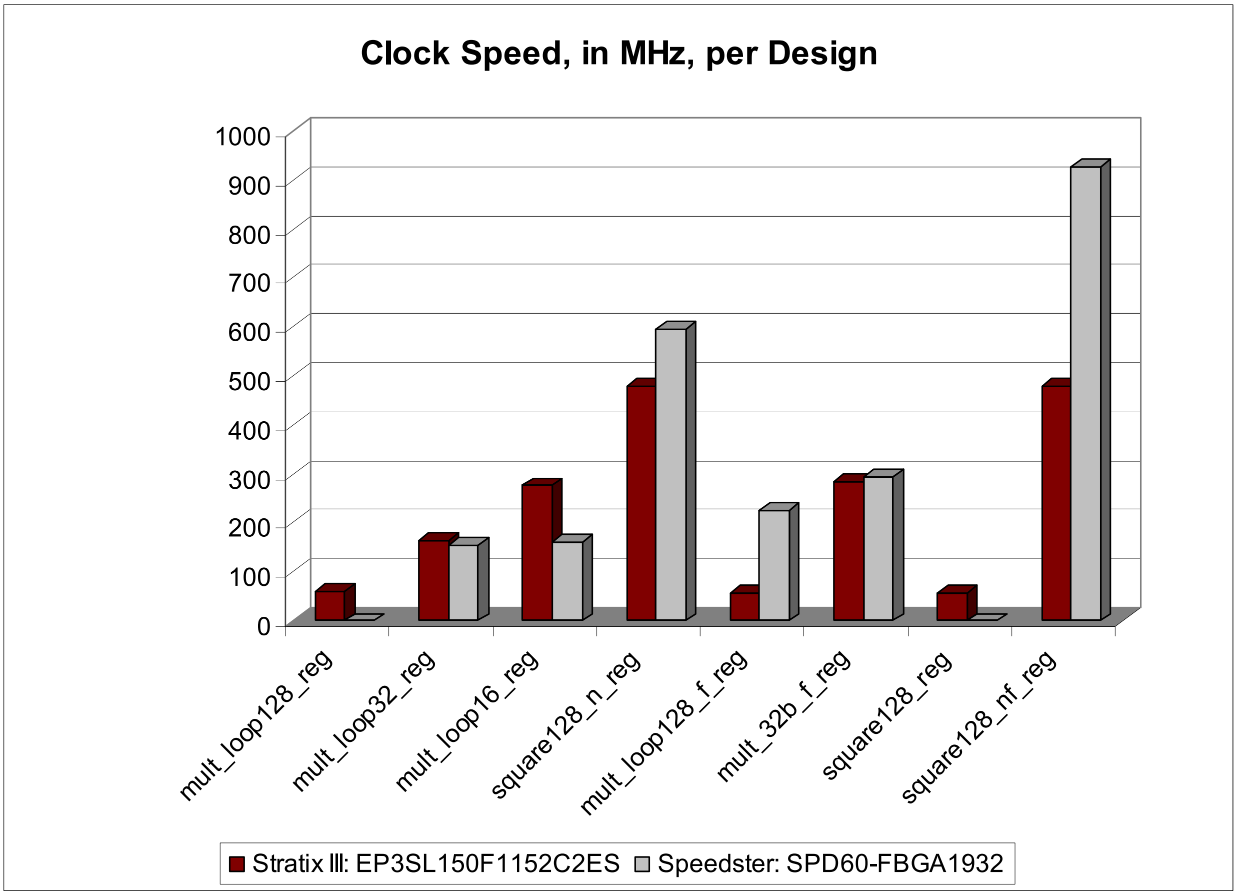


Figure 12: Graphical representation comparing clock speeds between the two FPGAs

The most significant results obtained are those of the performance measured in *square128_nf_reg* and *mult_loop128_f_reg*. These two components were modified only a small amount to work better in tandem with how the Achronix optimization algorithms are performed; removing feedback as well as removing resets, clocks, and other synchronous components while maintaining design functionality proves to be a trivial task for these designs and Achronix produces results that are commendable. The clock speed for the Altera and Achronix boards for the *square128_nf_reg* design were 480MHz and 926MHz, respectively. This is a roughly 93% increase in clock speed which, given the simplicity of the design modifications, is an excellent increase in clock speed. This increase in clock speed is also beneficial due to most FPGAs having a clock speed ceiling speed of around 400-450MHz that can only be increased a small amount and puts a greater stress on the synchronized components in the design. For *mult_loop128_f_reg*, the results are impressive too. The Achronix Speedster, in conjunction with ACE, was able to run at a speed of 225MHz compared to that of the Altera Stratix III board that was only capable of running the design at 58MHz. This result indicates a 388% increase in speed which is a huge increase in system throughput. These results are also of importance given that they are required, functional parts of the GHASH equations that describe GCM. By increasing the speed of the multiplier and squarer, the GHASH speed directly benefits by the increase as the other algorithms defining GHASH are not the limiting factors – the multiplier is. With the multiplier running at almost four times normal speeds, GCM is capable of encrypting at a similar increased speed multiple, theoretically.

The significance and magnitude of these results are important for another key goal of this project: to evaluate the feasibility of introducing ACE to a commercial or industrial designer. This, in combination with other factors experienced when working with ACE, can be found in 6.2 Achronix CAD Environment Effectiveness.

6 Conclusions

6.1 Conclusion

This project placed us into a real-world engineering firm with a meaningful project relevant to more than just our academic interests. General Dynamics C4 Systems and the Achronix Semiconductor Corporation relied on us to complete this project to the standards of engineering professionals. This helped provide the project with a real sense of meaning as we knew that the results we obtained would be taken into account for possible purchase of future software and hardware for GDC4S.

What quickly became a challenge for us was time management. While we, and our supervisors, had goals we wanted to accomplish and set out with those original goals, the real world of engineering and time constraints hastily reminded us that we only had seven weeks to accomplish whatever goals we put forth. It was with this time constraint in mind that we had to adjust our goals and reach for goals that we knew were within the time period we had, leaving the other goals open still for future researchers or project teams.

We ran into several issues during our project with regards to not only technical and software problems, but also problems in communication. We knew our supervisors were going to be very busy and thus had to take even the hardest of problems onto our own plates and determine a way to solve them. This was not necessarily the quickest way of solving these problems, but it was necessary that we solve them ourselves and so we did. Software problems arose that ranged anywhere from licensing issues with limited software to trying to understand complex issues such as pipelining with programs like Synplify Pro. We managed to avoid wasting time, however, by thoroughly testing our designs and following good coding practices as well as implementing a solid procedure for synthesizing and compiling our code.

This project culminated with the success of achieving near gigahertz speeds on the *square128_nf_reg* design, which represents not only the success of the ACE software, but the success of goals that, for a moment, looked unreachable in our time frame. Reaching the maximum speed on a board such as the Stratix III and then watching the FPGA design in ACE more than break that speed by a factor of two was not only satisfying, but represented a design goal being accomplished during the final week. This goal looked unreachable and the project was out of time, almost choosing to settle on the progress it had made, but with the speed obtained in the *square128_nf_reg* design, the real purpose and goal of the project was able to be realized.

6.2 Achronix CAD Environment Effectiveness

The Achronix CAD Environment offers promising results despite the developmental nature of the tool in its current state. While this project did not delve deeply into all the features of the program and thus we can only evaluate the features we used throughout this project, from which our decision about ACE effectiveness will be drawn. As most companies' engineering force is built of synchronous designers, it was an intelligent choice for Achronix to force their software to accept synchronous designs. Typical, unmodified designs tested as controls, such as *mult_loop16*, *mult_loop32* and *mult_loop128* did not benefit from the ACE tools. These designs were input in their original synchronous form with registers on the inputs and output (for the purpose of measuring speeds) and performed better on the Altera Stratix III board overall than on the Achronix Speedster. With regards to unmodified designs, we found the ACE tools to be not as practical as Quartus II and slower as well.

The main purpose of the ACE tools was that one could provide simply modified designs and produce significant results. We tested this and, as can be seen in *mult_loop128_f_reg*, the results were laudable. A good increase in speed was also achieved for the *square128_nf_reg* module and illustrates that ACE has no trouble taking designs that are at their limit on other FPGAs and bumping them up to near gigahertz speeds without too much extra work. Clock speed comparisons aside, ACE produced results that indicate some mixed success in other design aspects of an FPGA system. While *square128_nf_reg and mult_loop128_f_reg* may have increased in speed, so it did too in number of LUTs needed. This, however, is mitigated by the fact that Altera uses 6-input ALUTs, which are composed of two 4-input LUTs. Taking this into account it is clear that designs implemented on an Achronix Speedster with the ACE tools do not significantly increase or decrease the number of LUTs used.

Where Achronix does not succeed with minimizing LUTs, however, it does well with register optimization in certain situations. Referring specifically to Figure 11, nearly all designs optimized through ACE show a reduced number of registers. The main reason for the reduction of additional registers needed in ACE is that Synplify dose not need to duplicate the registers as many times on the Achronix chip in order to achieve optimal speed. *Mult_loop128_f_reg* had only 1101 registers in ACE compared to 1576 in Quartus II. This is almost a 44% decrease in registers. This information makes it an easier decision when evaluating a design coming from ACE as both clock speed and number of registers, two deciding factors, are better with ACE.

The user interface in ACE is straight forward and intuitive to anyone who may have used Xilinx ISE or Synplify Pro and is easily learned with relatively no guidance. A console window may be viewed at any time that details the programs processes specifically. In addition, all commands may be run through this console window as the programs graphical user interface is simply linked to these console commands. This makes advanced use and batch scripting easier for complex projects. The use of HTML-formatted results that are displayed in one of the windows using the system’s integrated browser provides for easily read results that are also available in plaintext format. On Linux systems, at this point in the beta of the software, however, there appears to be some issue recognizing the default browser for the operating system. The console window, while useful, displays cryptic results as well, at times. During the program’s timing analysis, the program notifies you of the step of the timing analysis it is currently on, though it does not tell you how many steps there will be. This can be problematic as time between steps may be hours and there is no way of estimating the time or number of steps the program will take before completion. This can range uncontrollably from 100 steps to 8000 or more steps in this project’s experience.

There does appear to be an issue with designs that begin to fill the Achronix chip more completely. While running the un-optimized version of the multiplier, mult_loop128, through ACE we encountered not only a prohibitively high run time of over 11 hours, but also a design that failed to successfully place and rout due to a Design Rule Check (DRC) failure. Additionally, we were unable to fully diagnose the cause of this problem due to the limited documentation we were provided with. Hopefully these problems will be addressed in a future release of the ACE tool.

Overall, this project recommends the Achronix CAD Environment as a promising option for designs for which feedback can be minimized or eliminated. While specifically designing for it is not required, projects that start their design with ACE and the Speedster FPGA in mind will most likely achieve higher initial results without additional modifications. For pre-existing designs, ACE is also viable as a tool for engineers that are looking to get every last ounce of performance from their FPGA, as is so often the case. While we suggest that the final version of ACE fix or modify some of the user interface issues we encountered, the program itself is stable and has never crashed. Designs have failed to compile in ACE successfully, but given that the designs were unmodified and not expected to be used specifically with ACE, this is not a reason to discount the program's otherwise praiseworthy optimization routines.

6.3 Recommendations for Future Research

The scope of this project was initially much larger and the project had to be re-focused in order to accomplish worthwhile goals in the short amount of time allocated. As such, future research opportunities that can build off the results here are pursuable. Included in this section are a couple of the research and development ideas that this project was not able to further investigate.

6.3.1 Manual Pipelining

While pipelining is done easily on Achronix chips using the ACE tool, it can also be done manually on traditional FPGAs by inserting registers into the middle of the design. This holds some potential for improving the performance of large, slow modules like the 128-bit, 1 clock cycle multiplier. Additionally, there is also an automated pipelining feature provided in Synplify Pro. Currently, it appears that the automated pipelining option in Synplify Pro will only pipeline very specific kinds of designs. Synplify Pro is, however, an evolving tool, and it is conceivable that they will expand this feature in a future release. It is worth noting that traditional silicon is limited to around 450MHz. The Stratix III chip used for this report would not exceed a clock speed of 480MHz, so designs like the efficient squarer are already at their limit in terms of performance. The Achronix speedster, however, can achieve speeds well over 1GHz. Manual pipelining cannot improve designs that already operate at the absolute limit of traditional FPGAs.

6.3.2 Multiplier Implementation into GCM

While Achronix had already built and tested an AES module for fast hardware encryption on their FPGA, General Dynamics C4 Systems wanted us to increase the existing multiplier's speed using ACE so that it may be used in a Galois/Counter Mode system. As mentioned previously, the multiplier was the limiting factor in the GHASH algorithm that describes message encryption and decryption for GCM. While we were able to greatly increase the speed of the multiplier that is to be used in the GHASH algorithm, it proved to be only days before the completion of this project. This put GHASH aside and, thus, it was not implemented. Future MQPs would do well to obtain an actual Achronix Speedster FPGA and attempt to implement the GHASH algorithm on it, testing its results in a physical system and assuring the system could operate without error. In addition to implementing the GHASH algorithm, the subcomponents of the algorithm could be tested to ensure they are capable of operating with the same performance of the now faster multiplier.

References

[1] W. Stallings, *Cryptography and Network Security : Principles and Practices*. Upper Saddle River, N.J. Pearson/Prentice Hall, 2006.

[2] B. Yang, et al., “High Speed Architecture for Galois/Counter Mode of Operation (GCM),” Cryptology ePrint Archive: Report 2005/146, Jun. 2005. http://eprint.iacr.org/2005/146.pdf

[3] D. McGrew, et al., “The Galois/Counter Mode of Operation (GCM),” May 2005. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-revised-spec.pdf

[4] Satoh, Akashi. “High-Speed Parallel Hardware Architecture for Galois Counter Mode.” Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium On (2007): 1863-1866.

[5] General Dynamics C4 Systems (2008, March 18) General Details - About Us - About Us. [Online]. Available: http://www.gdc4s.com/about/

[6] National Institute of Standards and Technology and Department of Commerce (2008, March 18) Announcing Approval of Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES). [Online]. Available: http://csrc.nist.gov/archive/aes/frn-fips197.pdf

[7] "Federal Register," *Federal Information Processing Standards*, Announcing the Advanced Encryption Standard (AES) ed., vol. 66, no. 40, February, 2001.

[8] National Institute of Standards and Technology (2008, March 18) Advanced Encryption Standard (AES) Questions and Answers. [Online]. Available: http://www.nist.gov/public_affairs/releases/aesq&a.htm

[9] H. Bar-El, "Introduction to Side Channel Attacks", Discretix, Israel, 2006.

[10] M. Kristan, B. Loveland, and R. Sazanowicz, "Dynamic Partial Reconfiguration of a Field Programmable Gate Array," MQP, Worcester Polytechnic Institute, Worcester, 2006.

[11] (2008, March 20) Questa AFV (Advanced Functional Verification). [Online]. Available: http://www.mentor.com/products/fv/abv/questa_afv/index.cfm

[12] (2008, March 20) Our Company. [Online]. Available: http://www.achronix.com/company.html

[13] E. Custodio and B. Marsland, "Self-Healing Partial Reconfiguration of an FPGA," MQP, Worcester Polytechnic Institute, Worcester, 2007.

[14] J. Fernando, D. Dalessandro, A. Devulapalli, and K. Wohlever, "Accelerated FPGA Based Encryption," *Cray User Group Proceedings*, 2005.

[15] Altera Corporation (2008, March 25) Quartus II Software General Questions and Answers. [Online]. Available: http://www.altera.com/products/software/products/quartus2/q_and_a/general/qts-q_and_a_general.html

[16] Synplicity, Inc. (2008, March 26) Synplicity: Products: Synplify Pro. [Online]. Available: http://www.synplicity.com/products/synplifypro/

[17] G. Orlando, "Efficient Elliptic Curve Processor Architectures for Field Programmable Logic," Ph.D Dissertation, Worcester Polytechnic Institute, Worcester, 2002.

[18] J. Teifel and R. Manohar, "An Asynchronous Dataflow FPGA Architecture," *IEEE Transactions on Computers*, vol. 53, iss. 11, 2004.

Glossary

AES – Advanced Encryption Standard; adopted by the National Institute of Standards and Technology (NIST) in 2002 as the primary encryption standard in the United States. Replaced Data Encryption Standard as a greatly more secure standard is still used widely as of 2008.

ASIC – Application Specific Integrated Circuit; lacks field-programming capabilities and suffers from being poorly adaptive but is higher speed, cheaper, and draws less power than the majority of FPGAs.

Black Box – Blank modules used as placeholders in design process. Representative placeholder of some module to be introduced later; technique commonly used in modular design.

CPLD – Complex Programmable Logic Device; a programmable logic device whose complexity is greater than of a Programmable Array Logic (PAL) but less than that of an FPGA that uses building blocks known as macro cells which contain logic implementing disjunctive normal form expressions and other specialized operations.

DRC – Design Rule Check; a check performed by FPGA tools to ensure that the design dose not violate any minimum or maximum physical constraints imposed by the silicon.

Exceed – Commercial X server that runs under Microsoft Windows allowing users to connect to Unix/Linux desktops and work similarly to that of Microsoft's Remote Desktop service.

Finite Fields – also known as Galois fields; fields with a finite number of elements. The number of elements in a finite field is sometimes referred to as the order of the field.

FPGA – Field Programmable Gate Array; semiconductor device containing programmable logic blocks offering the capabilities to be repeatedly re-programmed after initial board production. Ideal choice for designs that must be updated often before finalization.

GCM – Galois/Counter Mode; a block cipher mode of operation that uses universal hashing over a binary Galois field to provide authenticated encryption. It can be implemented in hardware at low cost and with low latency while maintaining high speeds.

GUI – Graphical User Interface; considered to be more user friendly, a GUI is used in place of a terminal to help facilitate software functions.

I/O – Input/Output.

Modular Design – Design model used in complex designs created by several team members. Design flow is created and followed by the team allowing for concurrent element design and ability to troubleshoot independent of entire design. Changes or modifications do not ripple through on each design modification, further increasing design success and implementation.

PPC – PowerPC microprocessor; originally developed by the Apple-IBM-Motorola alliance and intended for use in personal computers. Operates on the Reduced Instruction Set Computer (RISC) architecture and are now widely used in embedded designs.

UUT – Unit Under Testing; a term used by engineers and engineering programs that describes the unit currently being tested upon.

VHDL – VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language; design-entry language used in FPGAs and ASICSs originally developed by the Department of Defense (DOD) and is now an IEEE standardized language used by hardware designers to describe the design of circuitry.

Appendix A Hardware Description Language Code

Hardware Description Language Code derived from Mark Krumpoch’s 128-bit multiplier has been removed at the request of General Dynamics C4 Systems. This includes mult_loop128, mult_loop32, and mult_loop16 and their corresponding asynchronous versions.

A.1 128-bit XOR Logic

```
------------------------------------------------------------------
--    file: xor128.vhd        modified: Mar26,2008 09:42AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;

ENTITY xor128 IS
      PORT(
            clk         : IN  std_logic;
            a,b         : IN  std_logic_vector(127 DOWNTO 0);
            sum         : OUT std_logic_vector(127 DOWNTO 0)
            );

END xor128;

ARCHITECTURE dataflow OF xor128 IS
BEGIN
      G1: FOR i in 0 TO 127 GENERATE
            sum(i) <= a(i) xor b(i);
      END GENERATE;
END dataflow;
```

A.2 128-bit Register

```
------------------------------------------------------------------
--    file: register128.vhd   modified: Mar26,2008 09:42AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
library IEEE;
```

```
use IEEE.std_logic_1164.all;

ENTITY register128 IS
      PORT  (
                  clk   : IN STD_LOGIC;
                  reset : IN STD_LOGIC;
                  le    : IN STD_LOGIC;
                  d_in  : IN STD_LOGIC_VECTOR (127 downto 0);
                  d_out : OUT STD_LOGIC_VECTOR (127 downto 0)
      );

END ENTITY register128;

ARCHITECTURE register128_arch OF register128 IS
BEGIN
      PROCESS(clk, reset)
      BEGIN
            IF reset = '0' THEN
                  d_out <= (others => '0');
            ELSE
                  IF clk'EVENT AND clk = '1' THEN
                        IF le = '1' THEN
                              d_out <= d_in;
                        END IF;
                  END IF;
            END IF;
      END PROCESS;

END ARCHITECTURE  register128_arch;
```

A.4 128-bit Registered Multiplier, 1 Clock Cycle

```
-------------------------------------------------------------------
--    file: mult_loop128_reg.vhd    modified: Mar28,2008 01:39PM--
-------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
-------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
-------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
-------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.STD_LOGIC_1164.ALL;
USE         ieee.STD_LOGIC_unsigned.ALL;
USE         ieee.STD_LOGIC_arith.ALL;
-------------------------------------------------------------------
ENTITY mult_loop128_reg IS
      PORT(
            clk         : IN  STD_LOGIC;
            rst_n       : IN  STD_LOGIC;
            reset       : IN  STD_LOGIC;
            le          : IN  STD_LOGIC;
            reg_x       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_y       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_out     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END mult_loop128_reg;
------------------------------------------------------------------
ARCHITECTURE structural OF mult_loop128_reg IS
      ---------------
      COMPONENT register128 IS
      PORT(
            clk   : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            le    : IN STD_LOGIC;
            d_in  : IN STD_LOGIC_VECTOR(127 downto 0);
            d_out : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT register128;
      ---------------
      COMPONENT mult_loop128 IS
            PORT(
            clk         : IN  STD_LOGIC;
            rst_n       : IN  STD_LOGIC;
            x           : IN  STD_LOGIC_VECTOR(0 to 127);
            y           : IN  STD_LOGIC_VECTOR(0 to 127);
            z           : OUT STD_LOGIC_VECTOR(0 to 127)
            );
      END COMPONENT;
      ---------------
      SIGNAL A_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL B_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL C_s: STD_LOGIC_VECTOR(127 downto 0);
------------------------------------------------------------------
BEGIN
      U1: register128   PORT MAP (clk, reset, le, reg_x, A_s);
      U2: register128   PORT MAP (clk, reset, le, reg_y, B_s);
      U3: mult_loop128  PORT MAP (clk, rst_n, A_s, B_s, C_s);
      U4: register128   PORT MAP (clk, reset, le, C_s, reg_out);
END structural;
```

A.5 128-bit Multiply Accumulate

```
------------------------------------------------------------------
--    file: mult_accu128.vhd  modified: Mar26,2008 09:39AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;

ENTITY mult_accu128 IS
```

```
      PORT(
            clk         : IN STD_LOGIC;
            rst_n       : IN STD_LOGIC;
            h           : IN STD_LOGIC_VECTOR(127 downto 0);
            a           : IN STD_LOGIC_VECTOR(127 downto 0);
            x           : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END mult_accu128;

ARCHITECTURE structural OF mult_accu128 IS
      ---------------
      COMPONENT xor128 IS
            PORT(
                  clk   : IN  STD_LOGIC;
                  a     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  b     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  sum   : OUT STD_LOGIC_VECTOR(127 downto 0)
                  );
      END COMPONENT;
      ---------------
      COMPONENT mult_loop128 IS
            PORT(
            clk         : IN  STD_LOGIC;
            rst_n       : IN  STD_LOGIC;
            x           : IN  STD_LOGIC_VECTOR(127 downto 0);
            y           : IN  STD_LOGIC_VECTOR(127 downto 0);
            z           : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT;
      ---------------
      SIGNAL C: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL L: STD_LOGIC_VECTOR(127 downto 0);
BEGIN
      U1: xor128 PORT MAP (clk, a, L, C);
      U2: mult_loop128 PORT MAP (clk, rst_n, h, C, L);
      x <= L;
END structural;
```

A.6 128-bit Squarer

```
------------------------------------------------------------------
--    file: square128.vhd     modified: Mar26,2008 09:39AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;
```

```
ENTITY square128 IS
      PORT(
            clk   : IN STD_LOGIC;
            rst_n : IN STD_LOGIC;
            a     : IN STD_LOGIC_VECTOR(127 downto 0);
            x     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END square128;

ARCHITECTURE structural OF square128 IS

      COMPONENT mult_loop128 IS
            PORT(
            clk   : IN  STD_LOGIC;
            rst_n : IN  STD_LOGIC;
            x     : IN  STD_LOGIC_VECTOR(127 downto 0);
            y     : IN  STD_LOGIC_VECTOR(127 downto 0);
            z     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT;

BEGIN
      U2: mult_loop128 PORT MAP (clk, rst_n, a, a, x);
END structural;
```

A.7 128-bit Efficient Squarer

```
------------------------------------------------------------------
--    file: square128_n.vhd   modified: Mar28,2008 01:01PM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;
------------------------------------------------------------------
ENTITY square128_n IS
      PORT(
            clk         : IN STD_LOGIC;
            reset       : IN STD_LOGIC;
            rst_n       : IN STD_LOGIC;
            a           : IN STD_LOGIC_VECTOR(127 downto 0);
            x           : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END square128_n;
------------------------------------------------------------------
ARCHITECTURE structural OF square128_n IS
      ---------------
      COMPONENT spread128 IS
```

```
            PORT(
                  clk   : IN  STD_LOGIC;
                  a     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  c     : OUT STD_LOGIC_VECTOR(127 downto 0);
                  g     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT;
      ---------------
      COMPONENT register128 IS
      PORT(
            clk   : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            le    : IN STD_LOGIC;
            d_in  : IN STD_LOGIC_VECTOR(127 downto 0);
            d_out : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT register128;
      ---------------
      COMPONENT mult_loop128 IS
            PORT(
                  clk   : IN  STD_LOGIC;
                  rst_n : IN  STD_LOGIC;
                  x     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  y     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  z     : OUT STD_LOGIC_VECTOR(127 downto 0)
                  );
      END COMPONENT;
      ---------------
      COMPONENT xor128 IS
            PORT(
                  clk   : IN  STD_LOGIC;
                  a     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  b     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  sum   : OUT STD_LOGIC_VECTOR(127 downto 0)
                  );
      END COMPONENT;
      ---------------
      SIGNAL H_s  : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL C_s  : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL G_s1 : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL G_s2 : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL F_s  : STD_LOGIC_VECTOR(127 downto 0);
BEGIN
      H_s <= (0 => '1', 1 => '1', 2 => '1', 7 => '1', OTHERS => '0');
      U1: spread128     PORT MAP (clk, a, C_s, G_s1);
      U2: register128   PORT MAP (clk, reset, '1', G_s1, G_s2);
      U3: mult_loop128  PORT MAP (clk, rst_n, H_s, C_s, F_s);
      U4: xor128        PORT MAP (clk, F_s, G_s2, x);
END structural;
```

A.8 128-bit Registered Efficient Squarer

```
------------------------------------------------------------------
--    file: square128_n_reg.vhd     modified: Apr02,2008 11:01AM--
```

```
------------------------------------------------------------------
--    Bryce Barcelo            John Taylor                      --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156               ext. 63158                       --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.STD_LOGIC_1164.ALL;
USE         ieee.STD_LOGIC_unsigned.ALL;
USE         ieee.STD_LOGIC_arith.ALL;
------------------------------------------------------------------
ENTITY square128_n_reg IS
      PORT(
            clk         : IN  STD_LOGIC;
            rst_n       : IN  STD_LOGIC;
            reset       : IN  STD_LOGIC;
            le          : IN  STD_LOGIC;
            reg_a       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_out     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END square128_n_reg;
------------------------------------------------------------------
ARCHITECTURE structural OF square128_n_reg IS
      ---------------
      COMPONENT register128 IS
      PORT(
            clk   : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            le    : IN STD_LOGIC;
            d_in  : IN STD_LOGIC_VECTOR(127 downto 0);
            d_out : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT register128;
      ---------------
      COMPONENT square128_n IS
            PORT(
            clk   : IN  STD_LOGIC;
            reset : IN  STD_LOGIC;
            rst_n : IN  STD_LOGIC;
            a     : IN  STD_LOGIC_VECTOR(0 to 127);
            x     : OUT STD_LOGIC_VECTOR(0 to 127)
            );
      END COMPONENT;
      ---------------
      SIGNAL A_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL B_s: STD_LOGIC_VECTOR(127 downto 0);
------------------------------------------------------------------
BEGIN
      U1: register128   PORT MAP (clk, reset, le, reg_a, A_s);
      U2: square128_n   PORT MAP (clk, reset, rst_n, A_s, B_s);
      U3: register128   PORT MAP (clk, reset, le, B_s, reg_out);
END structural;
```

A.9 128-bit Registered Asynchronous Multiplier

```
------------------------------------------------------------------
--    file: mult_loop128_f_reg.vhd  modified: Apr09,2008 4:04PM --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.STD_LOGIC_1164.ALL;
USE         ieee.STD_LOGIC_unsigned.ALL;
USE         ieee.STD_LOGIC_arith.ALL;
------------------------------------------------------------------
ENTITY mult_loop128_f_reg IS
      PORT(
            clk         : IN  STD_LOGIC;
            reg_x       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_y       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_out     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END mult_loop128_f_reg;
------------------------------------------------------------------
ARCHITECTURE structural OF mult_loop128_f_reg IS
      ---------------
      COMPONENT register128 IS
      PORT(
            clk   : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            le    : IN STD_LOGIC;
            d_in  : IN STD_LOGIC_VECTOR(127 downto 0);
            d_out : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT register128;
      ---------------
      COMPONENT mult_loop128_f IS
            PORT(
            x     : IN  STD_LOGIC_VECTOR(0 to 127);
            y     : IN  STD_LOGIC_VECTOR(0 to 127);
            z     : OUT STD_LOGIC_VECTOR(0 to 127)
            );
      END COMPONENT;
      ---------------
      SIGNAL A_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL B_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL C_s: STD_LOGIC_VECTOR(127 downto 0);
------------------------------------------------------------------
BEGIN
      U1: register128         PORT MAP (clk, '1', '1', reg_x, A_s);
      U2: register128         PORT MAP (clk, '1', '1', reg_y, B_s);
      U3: mult_loop128_f      PORT MAP (A_s, B_s, C_s);
      U4: register128         PORT MAP (clk, '1', '1', C_s, reg_out);
```

```
END structural;
```

A.10 128-bit Spreader

```
------------------------------------------------------------------
--    file: spread128.vhd     modified: Mar26,2008 09:41AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;

ENTITY spread128 IS
      PORT(
            clk   : IN  STD_LOGIC;
            a     : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
            c     : OUT STD_LOGIC_VECTOR(127 DOWNTO 0);
            g     : OUT STD_LOGIC_VECTOR(127 DOWNTO 0)
            );

END spread128;

ARCHITECTURE dataflow OF spread128 IS

BEGIN

PROCESS (clk)
BEGIN
      FOR i IN 0 TO 63 LOOP
            g(2*i)      <= a(i) ;
            g(2*i+1)    <= '0';
            c(2*i)      <= a(i+64) ;
            c(2*i+1)    <= '0';
      END LOOP;
END PROCESS;

END dataflow;
```

A.11 128-bit Fast XOR

```
------------------------------------------------------------------
--    file: xor128_f.vhd      modified: Apr16,2008 01:42PM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
```

```
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;

ENTITY xor128_f IS
      PORT(
            a,b   : IN  std_logic_vector(127 DOWNTO 0);
            sum   : OUT std_logic_vector(127 DOWNTO 0)
            );

END xor128_f;

ARCHITECTURE dataflow OF xor128_f IS
BEGIN
PROCESS (a,b)
BEGIN
      FOR i in 0 TO 127 LOOP
            sum(i) <= a(i) xor b(i);
      END LOOP;
END PROCESS;
END dataflow;
```

A.12 128-bit Asynchronous Squarer

```
------------------------------------------------------------------
--    file: square128_nf.vhd  modified: Apr16,2008 01:49PM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;
------------------------------------------------------------------
ENTITY square128_nf IS
      PORT(
            clk         : IN STD_LOGIC;
            a           : IN STD_LOGIC_VECTOR(127 downto 0);
            x           : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
END square128_nf;
------------------------------------------------------------------
ARCHITECTURE structural OF square128_nf IS
      ---------------
      COMPONENT spread128_f IS
            PORT(
                  a           : IN  STD_LOGIC_VECTOR(127 downto 0);
                  c           : OUT STD_LOGIC_VECTOR(127 downto 0);
                  g           : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT;
      ---------------
      COMPONENT mult_loop128_f IS
            PORT(
                  x           : IN  STD_LOGIC_VECTOR(127 downto 0);
                  y           : IN  STD_LOGIC_VECTOR(127 downto 0);
                  z           : OUT STD_LOGIC_VECTOR(127 downto 0)
                  );
      END COMPONENT;
      ---------------
      COMPONENT xor128_f IS
            PORT(
                  a     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  b     : IN  STD_LOGIC_VECTOR(127 downto 0);
                  sum   : OUT STD_LOGIC_VECTOR(127 downto 0)
                  );
      END COMPONENT;
      ---------------
      SIGNAL H_s  : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL C_s  : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL G_s  : STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL F_s  : STD_LOGIC_VECTOR(127 downto 0);
BEGIN
      H_s <= (0 => '1', 1 => '1', 2 => '1', 7 => '1', OTHERS => '0');
      U1: spread128_f         PORT MAP (a, C_s, G_s);
      U2: mult_loop128_f      PORT MAP (H_s, C_s, F_s);
      U3: xor128_f            PORT MAP (F_s, G_s, x);
END structural;
```

A.13 128-bit Asynchronous Registered Squarer

```
------------------------------------------------------------------
--    file: square128_n_reg.vhd     modified: Apr17,2008 10:01AM--
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.STD_LOGIC_1164.ALL;
USE         ieee.STD_LOGIC_unsigned.ALL;
USE         ieee.STD_LOGIC_arith.ALL;
------------------------------------------------------------------
ENTITY square128_nf_reg IS
      PORT(
            clk         : IN  STD_LOGIC;
            reset       : IN  STD_LOGIC;
            reg_a       : IN  STD_LOGIC_VECTOR(127 downto 0);
            reg_out     : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
```

```
END square128_nf_reg;
------------------------------------------------------------------
ARCHITECTURE structural OF square128_nf_reg IS
      ---------------
      COMPONENT register128 IS
      PORT(
            clk   : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            le    : IN STD_LOGIC;
            d_in  : IN STD_LOGIC_VECTOR(127 downto 0);
            d_out : OUT STD_LOGIC_VECTOR(127 downto 0)
            );
      END COMPONENT register128;
      ---------------
      COMPONENT square128_nf IS
            PORT(
            clk   : IN  STD_LOGIC;
            a     : IN  STD_LOGIC_VECTOR(0 to 127);
            x     : OUT STD_LOGIC_VECTOR(0 to 127)
            );
      END COMPONENT;
      ---------------
      SIGNAL A_s: STD_LOGIC_VECTOR(127 downto 0);
      SIGNAL B_s: STD_LOGIC_VECTOR(127 downto 0);
------------------------------------------------------------------
BEGIN
      U1: register128   PORT MAP (clk, reset, '1', reg_a, A_s);
      U2: square128_nf  PORT MAP (clk, A_s, B_s);
      U3: register128   PORT MAP (clk, reset, '1', B_s, reg_out);
END structural;
```

A.14 128-bit Asynchronous Spreader

```
------------------------------------------------------------------
--    file: spread128_f.vhd   modified: Apr17,2008 10:18AM      --
------------------------------------------------------------------
--    Bryce Barcelo           John Taylor                       --
------------------------------------------------------------------
--    bryce.barcelo@gdc4s.com john.taylor@gdc4s.com             --
------------------------------------------------------------------
--    ext. 63156              ext. 63158                        --
------------------------------------------------------------------
LIBRARY     ieee;
USE         ieee.std_logic_1164.ALL;
USE         ieee.std_logic_unsigned.ALL;
USE         ieee.std_logic_arith.ALL;

ENTITY spread128_f IS
      PORT(
            a     : IN  STD_LOGIC_VECTOR(127 DOWNTO 0);
            c     : OUT STD_LOGIC_VECTOR(127 DOWNTO 0);
            g     : OUT STD_LOGIC_VECTOR(127 DOWNTO 0)
            );
```

```
END spread128_f;

ARCHITECTURE dataflow OF spread128_f IS

BEGIN

PROCESS (a)
BEGIN
      FOR i IN 0 TO 63 LOOP
            g(2*i)      <= a(i) ;
            g(2*i+1)    <= '0';
            c(2*i)      <= a(i+64) ;
            c(2*i+1)    <= '0';
      END LOOP;
END PROCESS;

END dataflow;
```