SECURITY FOR THE PROCESSOR-TO-MEMORY INTERFACE USING FIELD PROGRAMMABLE GATE ARRAYS

By

George E. Sewell

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

Master of Science

In

Electrical Engineering

August 2007

Nashville, Tennessee

Approved:

Professor William H. Robinson

Professor Gabor Karsai

ACKNOWLEDGEMENTS

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422) and the following organizations: AFOSR (#FA9550-06-1-0244) Cisco, British Telecom, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia and United Technologies.

TABLE OF CONTENTS

Chapter **Page**

ACKNOWLEDGEMENTS ..... **ii**

TABLE OF FIGURES ..... **iv**

TABLE OF TABLES ..... **iv**

I: INTRODUCTION **1**

II: SECURITY APPLICATIONS IN FPGAS **4**

1) Security 4
2) Field Programmable Gate Array Platform 5
A) Logic Implementation on an FPGA *6*
B) Capabilities of an FPGA *8*
C) FPGA vs. ASIC *9*
3) Security and FPGAs 11

III: DES IMPLEMENTATION ON THE FPGA **13**

1) Structure of a basic DES algorithm 13
2) Structure of the algorithm on the FPGA 15

IV: IMPLEMENTATION OF THE TEST SYSTEM **19**

1) Processor Design 22
2) Memory Controller 25
3) Encryption Interface 26
4) Test System 29
5) Test Setup 30

V: ANALYSIS METHODOLOGY **31**

1) Baseline Performance Experiments 32
2) Complete System Setup 34

VI: RESULTS AND ANALYSIS/DISCUSSION **36**

VII: CONCLUSION **39**

REFERENCES **42**

APPENDIX OF FIGURES AND VHDL CODE **45**

TABLE OF FIGURES

Figure 1 - Logic Block Array 6

Figure 2 - Basic Logic Block 6

Figure 3 - Part Cost vs. Quantity 10

Figure 4 - Sample DES Design 13

Figure 5 - Feistel Block 15

Figure 6 - Feistel Function 16

Figure 7 - Key Generator 17

Figure 8 - Altera Cyclone II Logic Block 20

Figure 9 - Sample Processor Cycle 22

Figure 10 - SRAM Latency Table and Sample Read Cycle 26

Figure 11 - Processor Test Setup 32

Figure 12 - Encryption Test Setup 34

Figure 13 - Final Assembled System 35

Figure 14 - Settle Time vs. Instruction Type 36

Figure 15 - Sampled Stabilization Time 37

TABLE OF TABLES

Table 1 – Instruction List for the RISC processor 24

CHAPTER I

INTRODUCTION

The ever-increasing speed and reliability of Field Programmable Gate Arrays (FPGAs) make them attractive platforms for security applications. Security is a very processor-intensive process, so having a secondary chip in the form of a co-processor or other circuit dedicated to the task can reduce the computational burden on the main processor and allow it to execute other tasks. Furthermore, some security applications require a fast response time which can only be achieved with dedicated hardware circuits. FPGAs provide the flexibility and resources to implement these specialized tasks.

Security exists not only on the software level but on the hardware level. Various types of attacks include downloading physical memory to access data, monitoring communication channels to eavesdrop on messages, or reverse-engineering integrated circuits to discover their function. One vulnerable location in a computer system is the interface between a microprocessor and its external memory. An attacker could monitor the exchanges between the microprocessor and external memory by listening on the interconnect bus. Also, the contents of the external memory could be downloaded for analysis [1]. Encrypting memory utilized by the processor can provide some protection against this form of access, but inserting an encryption module between the processor and memory has its challenges. Memory accesses are often the performance bottleneck, so extra processing time for encryption can degrade the application performance [2]. The

encryption module will also require transparency to avoid redesign of the processor-to-memory interface.

This work utilizes an FPGA in order to design hardware-level encryption [3] that provides a transparent interface for memory accesses with minimal impact on the normal operation of the processor. To demonstrate this functionality, the Data Encryption Standard (DES) algorithm [4] has been implemented in hardware using an FPGA. This module was used as an encryption/decryption device between a test processor and memory. Results from the simulation show that the encryption operates within the theoretical bounds of the cycle time for memory access. The hardware implementation of the test system validates the simulation results. The timing and the resource utilization of the hardware implementation have been analyzed for the processor, memory, and the encryption both separately and as a full system.

This thesis is broken down into several chapters:

- Chapter Two motivates the usage of FPGAs in security applications.
- Chapter Three discusses the DES algorithm and how the algorithm is translated into an FPGA implementation.
- Chapter Four describes the processor and memory components implemented in the test system.
- Chapter Five describes the analysis methodology to evaluate the processor, memory and encryption in the test system.
- Chapter Six presents the results of this project.

- Chapter Seven concludes the discussion on securing the interface between a processor and memory and describes the implications for a full-scale computing system.

CHAPTER II

SECURITY APPLICATIONS IN FPGAs

1) *Security*

Security, by definition, is the safeguarding of an individual's or organization's assets against danger (e.g., theft, destruction, or unwarranted modification). Security, from a computer standpoint, becomes necessary in any case where we want to protect data from others who are not authorized to see, obtain, or manipulate it. Basic digital security consists of managing three aspects: (1) confidentiality, (2) integrity, and (3) availability [5]. Maintaining confidentiality ensures that the data is unable to be read by someone without authorization. This is the most basic of security concerns; while data may be easily accessible, you do not want an attacker to read it or utilize it. The integrity property differs from confidentiality in that it involves making sure that the data remains unchanged from its original form. There are two forms of integrity: (1) data integrity and (2) source integrity. Data integrity must verify that the data transmitted has not been falsified or altered. Source integrity must confirm that the origin of the data is legitimate. Simple integrity methods, such as a parity checker, can verify the data and detect if corruption occurs, while more complex methods, such as error correction, enable the recovery of corrupted data. Finally, availability involves maintaining the ability for authorized viewers to access secured data. While availability is one of the more difficult of the security priorities, it generally refers more to preventing attackers from

intercepting data requests and keeping services from being brought down, as is the case in a Denial of Service Attack [6]. The work presented in this thesis focuses upon maintaining confidentiality within an individual computer system.

2) Field Programmable Gate Array Platform

Field Programmable Gate Arrays (FPGAs) occupy an interesting and useful niche in computer engineering because of their programmability. This capability makes it a suitable platform to prototype integrated circuit designs. FPGAs can consist of several hundred to several thousand functional logic blocks arranged in a grid with communication channels lying in between the blocks. While designs can vary, Figure 1 shows the typical island-style architecture [7] where each logic block is surrounded by programmable interconnects and channels. Each logic block contains some basic logical elements, memory, and a look-up table (LUT), which can be configured to perform a small function or task. Individually, each logic block can only perform simple tasks, but when chained together, more complicated objectives become possible. To implement more complex designs, logic blocks communicate with each other via channels, i.e., the programmable interconnect that lies in the grid spaces between the blocks. Where channels cross, there are control switch blocks with additional logic that determine signal routing. For reprogrammable FPGAs, the configuration settings for logic blocks and channels are maintained by an SRAM, Electrically Erasable Programmable Read-Only Memory (EEPROM) or Flash ROM.

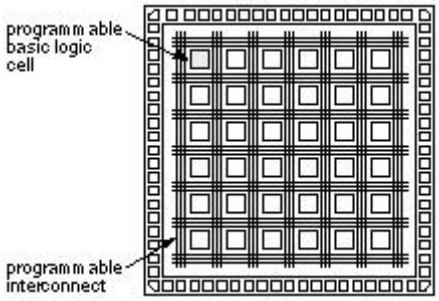


Figure 1 - Logic Block Array [8]

A) Logic Implementation on an FPGA

A basic logic block, or logic element, consists of three parts: a look up table, a storage element, and a multiplexer to select internal signals (Figure 2). A sample lookup table may contain four input pins and one output pin which enables it to perform a simple 4-bit function such as a 4-input AND gate. The storage element consists of a simple clocked D flip-flop that, when enabled, will store the output of the lookup table. The final output of the logic block is determined by a switching block which chooses a line for the output channel.

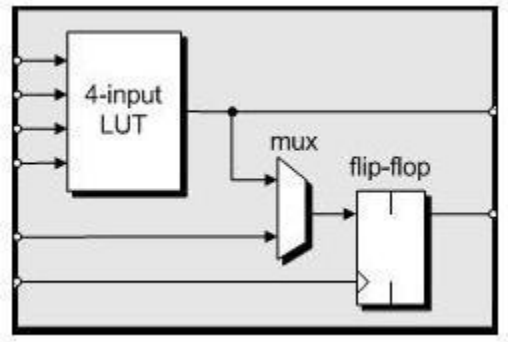


Figure 2 - Basic Logic Block [9]

The basic logic block can be expanded in many ways to add versatility [9]. The first, and easiest, is stacking the logic blocks and creating “arrays” of varying sizes. A typical size may be 8 or 16, which can handle a byte of data collectively. Another method to increase performance is to increase the number of inputs into the look-up table. This enables more complex look-up tables, but also increases the size of the table exponentially. Larger LUTs may not be utilized as efficiently as the typical 4-input LUT. Additional components can include dedicated logic gates for AND, OR, and XOR, as well as registers, bit-shifting components for arrays, and carry chains for faster addition. Finally, certain components can be specialized by inserting dedicated hardware for commonly used functions, such as a multiply-accumulate operation or memory.

Channels are the means through which data can be transmitted between logic blocks [10]. Channels lie in the row and column space between the logic blocks in the architecture. Furthermore, if the logic blocks are stacked, then the channels will be likewise stacked. Each channel generally has multiple parallel lines for communication so that several signals can utilize the same channel, which in turn reduces the complexity for routing. A switch block is position where row and column channels meet. Switch blocks contain all the necessary connections that allow signals to transfer from row channels to column channels, and vice versa. As such, it does not matter where a logic block is located relative to the next one in a function chain as long as it is possible to route a signal from the prior output to the input of the next. The process of routing signals between logic blocks, while complex, is the basis for the flexibility in an FPGA.

B) Capabilities of an FPGA

FPGAs at their basic level could be used to implement simple logic circuits by assigning the input pins to a logic function programmed into the device. However, this does not utilize some of the more important features. The key to getting the most out of the FPGA is by using the advanced features a typical FPGA may provide: language abstraction and reprogrammability.

The basis for abstraction comes from the hardware description languages. Using schematics and block diagrams to describe the dataflow, while possible, is impractical for larger systems. Thus programming languages are used. The two most common languages are VHDL (VHSIC hardware Description Language) [11] and Verilog [12], both of which provide structure for defining logical blocks and behavior of FPGA devices. Electronic Design Automation (EDA) tools first convert the code and/or schematic diagrams into formal logic functions. The EDA tool will then organize the functions into logic blocks, followed by placing and routing the design within the actual chip. The final step in the process is the generation of a configuration file which can be used to program the FPGA.

VHDL was used for this project. Basic VHDL design utilizes blocks or 'entities.' Entities have their behavior controlled by the architecture that defines how signals will flow from the inputs to the outputs of the entity. In order for an entity to accomplish a task, it must have an output, but not necessarily an input. Within the architecture, signals are manipulated by functions which range from simple logic gates to mathematical operations and sub-architectures. Programmers are able to define their own functional architectures and use them just like any high-level language. Signals themselves can have

complex values [13]. Signals represent data lines on the bit-level. Beyond the simple 1 and 0 values, they can possess other electrical values such as weak, strong, unknown, or high-impedance. VHDL is used to define specifically the behavior of an electrical device, so it is important for the programmer to understand the operation of signals on the data lines and either utilize or make allowances for them.

The final key advantage to FPGAs as development platforms comes from their ability to be reprogrammed. Instead of manufacturing an entire chip to test a design, the FPGA can be programmed, tested, and reprogrammed should the current design fail to function as desired. This final aspect is the most important feature when considering a platform for system prototyping.

C) FPGA vs. ASIC

Application Specific Integrated-Circuits, or ASICs, is the umbrella definition that applies to any chip that can be designed to be configurable to a particular task [8]. While FPGAs can fall under this definition, convention keeps them separate. Fundamentally, most ASIC chips are structurally similar to FPGAs, the primary difference being that ASICs are typically one-time programmable. Instead of memory cells to program an ASIC device, they are manufactured with anti-fuses [14] to allow users to configure the desired function into the chip or are manufactured with metal interconnect according to the specified design.

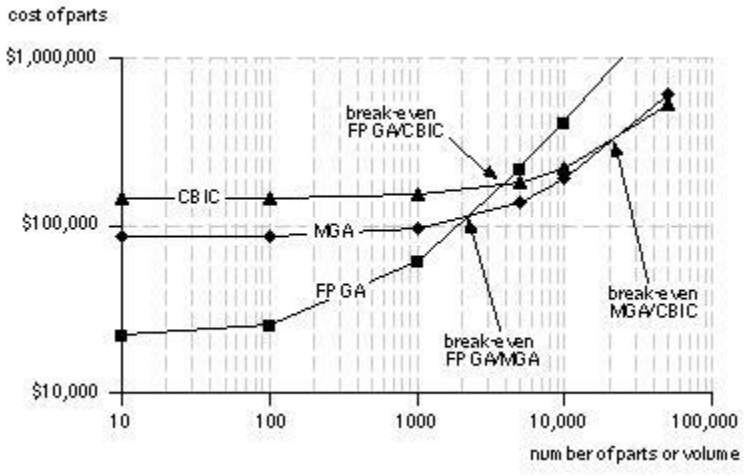


Figure 3 - Part Cost vs. Quantity [7]

The reprogramming aspect of an FPGA comes at a cost with respect to ASIC designs. While the chips are fundamentally similar, FPGAs sacrifice speed and cost for versatility. Per-part, FPGAs are more expensive, however they require a lower initial design cost since you do not need to replace the FPGA with each failed design. Figure 3 shows the volume-to-cost analysis for FPGAs and two kinds of ASICs: Masked Gate Array (MGA) and Custom Cell-Based ASIC (CBIC). The initial fixed cost for FPGAs is considerably lower than the MGAs and CBICs. However, as the volume of parts increases, ASIC implementations become more competitive. One final aspect of note is that an ASIC can also operate faster than an equivalent FPGA. Since the ASIC is hardwired with metal lines, there are no additional routing delays that result from the channel switching. Furthermore, designers can perform additional optimizations to the ASIC design prior to manufacturing since they have greater control of the layout of the integrated circuit.

3) Security and FPGAs

FPGAs are an attractive option for security applications because of their reprogrammability [15]. This enables operators to manage security algorithms in one contained package as well as to change security methods and algorithms without having to replace their hardware. Common security applications for FPGAs include secure co-processors and network security algorithm implementations [16][17]. While it is useful to know the uses of FPGAs in security applications, it must also be noted that FPGAs are vulnerable to a variety of attacks themselves [3]. The three prominent means of attacking an FPGA are through black box, read back, and side channel [18].

Black Box attacks on FPGAs are an extremely basic method for attempting to discover the functionality encoded on the chip. The basis of this attack is to feed every possible input combination into the FPGA input pins and monitor the resulting outputs. The inner workings of an FPGA may be determined through the use of maps and truth tables. This method is time consuming and becomes exponentially difficult as the quantity of inputs and outputs increases on the FPGA. Additionally, if an FPGA contains one or more state machines, the entire technique can be rendered invalid.

The next viable attack is to read back the configuration data. Reading back the data from the FPGA is useful for programmers who need to debug their projects as the implementation may be different from what they realized. Read-back is also useful to check for software errors resulting from prolonged use, such as accidental bit flips or even hard errors when used in radiation environments. As this is a common and logical method for attacking an FPGA, most developers include debug mode flags that enable or disable read-back for the contents of memory.

Higher up on the security schemes are the side-channel attacks. A side-channel attack is defined as any method that does not use a standard input/output scheme to monitor the activity of an FPGA. Common side channels can be as simple as watching non-output pins for leaked signals while the more difficult ones include power demand analysis and monitoring electromagnetic radiation. Power-demand analysis is the procedure of taking an FPGA under-test and watching the consumption of power by the chip over time [19][20]. Certain logic operations and functions require a differing amount of power to run and as a general rule, the more logic elements activated by a process the more power the FPGA will consume. By watching the transient power consumption of the chip, it is possible to gain an understanding of its current state. While this does not directly provide any real information about the content of the FPGA, coupled with I/O monitoring, this can be useful to understanding the operation of an FPGA and can signify which outputs to monitor. The monitoring of electromagnetic (EM) radiation from an FPGA works in a similar fashion by understanding how much EM radiation is being emitted by the chip. By mapping the EM radiation to different areas of the chip, the operational state of the FPGA can be deduced.

CHAPTER III

DES IMPLEMENTATION ON THE FPGA

1) ***Structure of a basic DES algorithm***

The Data Encryption Standard (DES) algorithm is fundamentally a cryptographic block cipher [4]. Therefore, for each block input, you receive another unique block with essentially a one-to-one mapping. Furthermore, DES does not involve block chaining and can be used for random access, which is essential to a cryptographic bridge between processor and memory. More advanced methods of the DES algorithm are available for use since the cryptographic strength of normal DES is under scrutiny [21]. While there are more complex forms of DES [22][23], for this implementation the normal structure of DES is used.

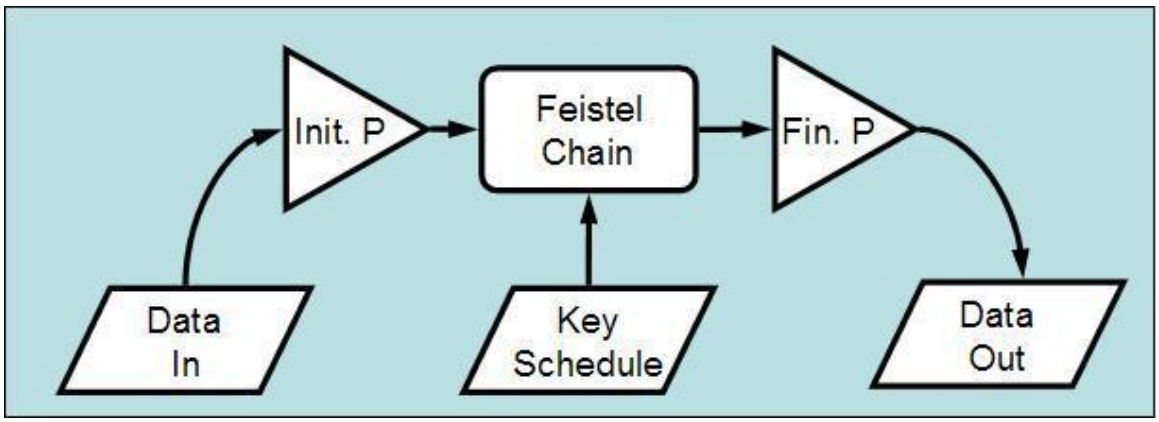


Figure 4 - Sample DES Design

A sample DES block is constructed of four parts: (1) Initial Permutation, (2) the Feistel Chain, (3) the Key Schedule, and (4) the Final Permutation (Figure 4). The strength of the DES algorithm lies not in the permutations, but the process through which it generates keys for the Feistel function and its repeated application. The Feistel chain contains 16 individual Feistel Blocks, and for each block the key schedule provides a unique key.

Following the initial permutation, the data block is split in half, requiring that the inputs possess an even number of bits. Within the Feistel chain, the block halves get placed on either an XOR function or the Feistel Function. The Feistel function takes the unique key for that block and performs a series of combinations and permutations on the data block. This is discussed in more detail later. The output of the Feistel function is then fed to the XOR to be combined with the first half of the data block. For each following Feistel block, the halves are alternated. After the sixteenth block, the resulting data is then fed through the output permutation, and the encryption process is completed.

The decryption process for DES utilizes the exact same structure as encryption; however the operation occurs in reverse order. First the data is fed into the output permutation. Then, starting at the last Feistel block, the data moves to each previous block up to the first, and finally passes through the input permutation. The primary advantage of this operation is that both the encryption and decryption processes can utilize similar hardware circuitry.

2) Structure of the algorithm on the FPGA [24]

Construction of the DES cipher occurred in 3 parts: the Feistel Chain, the Key Generator, and the Permutation blocks. The Feistel Chain is composed of sixteen repeated individual Feistel blocks that actually perform the cryptographic process. The Key Generator, or Key Schedule, constructs each of the sixteen unique keys used in each link of the Feistel Chain. The Permutation Blocks, of which there are two, randomize the bit order of the data inputs and outputs by an arbitrary, yet consistent, pattern.

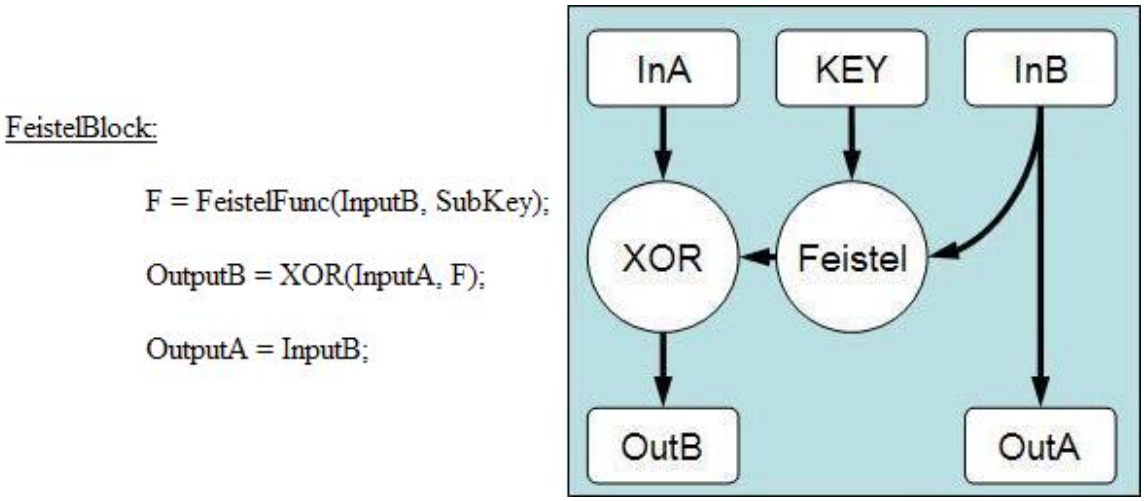


Figure 5 - Feistel Block

The Feistel Block (Figure 5) contains both the Feistel function and the XOR combination. It receives the SubKey from the Key Generator and the input data block that is then split into two halves: InputA and InputB. InputB is fed along with the SubKey into the Feistel function. The Feistel function outputs a new data block, which is then XOR'ed with InputA. The result of the XOR is then fed out as OutputB, while InputB is fed to OutputA unchanged. The next Feistel Block will utilize OutputA for its InputA,

and OutputB for InputB. By alternating the halves, each instance of the Feistel Block will encipher on a separate half of the data.

FeistelFunc:

F1 = XOR(SubKey, InputB);

F2 = S1(F1) & S2(F1) &
S3(F1) & S4(F1);

F3 = Permutation(F2);

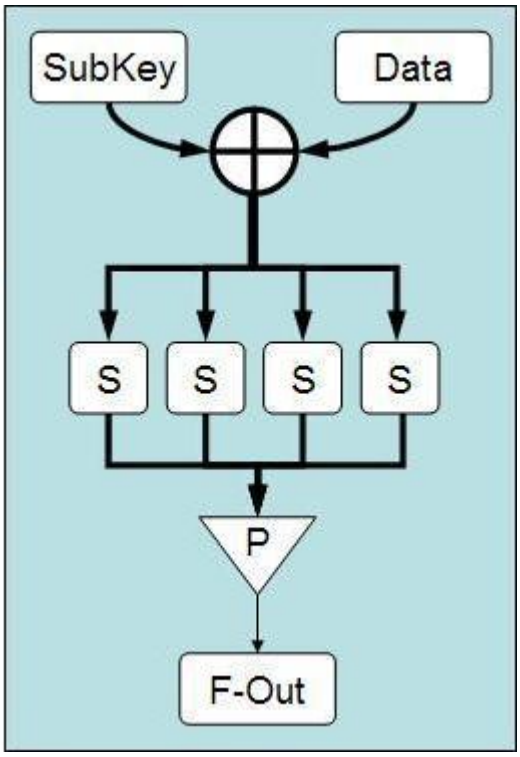


Figure 6 - Feistel Function

The cornerstone to this, however, is the Feistel Function (Figure 6). The first step is to XOR the SubKey with InputB. The next step is to break up the result into equal pieces and feed them into the S-Block. The S-Block is a basic look up table (LUT) that reduces the number of input bits using an arbitrary rubric. The final part is a permutation block that shuffles the bits from the S-Block.

The Key Generator (Figure 7), which produces each sub-key, receives two inputs: (1) the Base Key and (2) the Decrypt signal. The Base Key is the complete DES key from which the individual keys are constructed. To begin, the Base Key is split in half to create

two parts (PartA, PartB). For each of the sixteen keys, each part is rotated by one bit and is fed into a permutation function to create the specific SubKey.

HalfA(n+1) = RotateLeft(HalfA(n));

HalfB(n+1) = RotateLeft(HalfB(n));

SubKey(n) = Permutation(HalfA(n), HalfB(n));

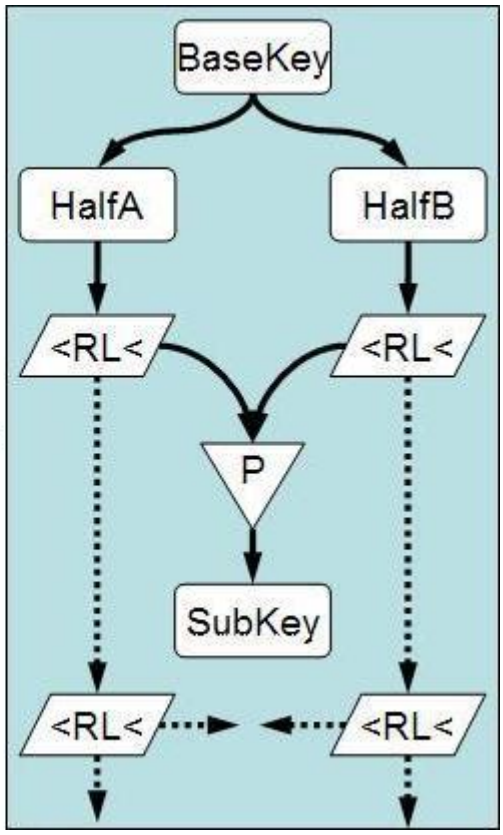


Figure 7 - Key Generator

The permutation function itself is a static compilation of an equal number of individual bits from PartA and PartB. As such, each bit is used at least once to contribute to the sub keys to provide a unique sub-key for each Feistel block. Finally the Decrypt signal indicates to the Key Generator when the DES is going through the decrypt cycle. When Decrypt is active, the keys are constructed in the same method, but are output in the reverse order, such that SubKey(16) becomes SubKey(1), and so on.

Finally, the Permutation Blocks serve a dual purpose for this implementation of the DES algorithm. The first purpose is to provide pre- and post-permutations to the data, the output permutation being the inverse of the input permutation. Simply changing the order of the bits offers no additional cryptographic strength; however the blocks serve a more useful purpose. In order to keep the implementation on the FPGA symmetric, and reduce the number of logic elements on the chip, the permutation block interprets the Decrypt signal (same as the one fed to the Key Generator) to swap the first and second halves of the data for the decrypt cycle. This ensures that the block will be decrypted instead of re-encrypted with the same key.

CHAPTER IV

IMPLEMENTATION OF THE TEST SYSTEM

For this project, the ultimate goal is to test the latency and the transparency of the encryption module. To determine the latency, cycle times of the test processor will be sampled to observe how encryption affects the overall performance. Of the utmost importance to any FPGA project is the idea of propagation delay. It is important to understand how long it takes for a signal, upon being clocked or receiving new inputs, to show an output change and to stabilize to a new output level. This end-to-end delay determines the maximum speed at which the FPGA can operate to ensure that all data reaches the output before the next cycle tries to use that data. This will be discussed primarily in the performance experiment section.

The hardware prototype used an Altera DE2 Development and Education Board [25]. While the board itself contains a large number of peripherals, there are only a few that concern this project. First and foremost is the FPGA itself, which is an Altera Cyclone II EP2C35F672C6 FPGA [26]. This design contains 68,416 logic elements with which to create designs as well as 622 I/O pins to control outputs and other peripherals. Embedded in the design are 150 pre-configured 18x18 multipliers and 1.1 Mbit of usable memory. The Cyclone II is optimized to run at a maximum clock speed of 402.5 MHz, while the embedded devices operate at 250 MHz. The standard Cyclone II logic block differs from a standard simple logic block in that it provides additional logic for controlling clock inputs, a global reset, and carry chain additions. Figure 8 shows a

Cyclone II logic block. Among the numerous other items on the DE2 board, this project utilized a bank of eighteen red LEDs, a bank of eighteen switches, two external clocking chips rated for 50 MHz and 27.5 MHz, and finally a 256K x 16-bit SRAM chip. The final important piece of hardware included is the USB Blaster programming interface.

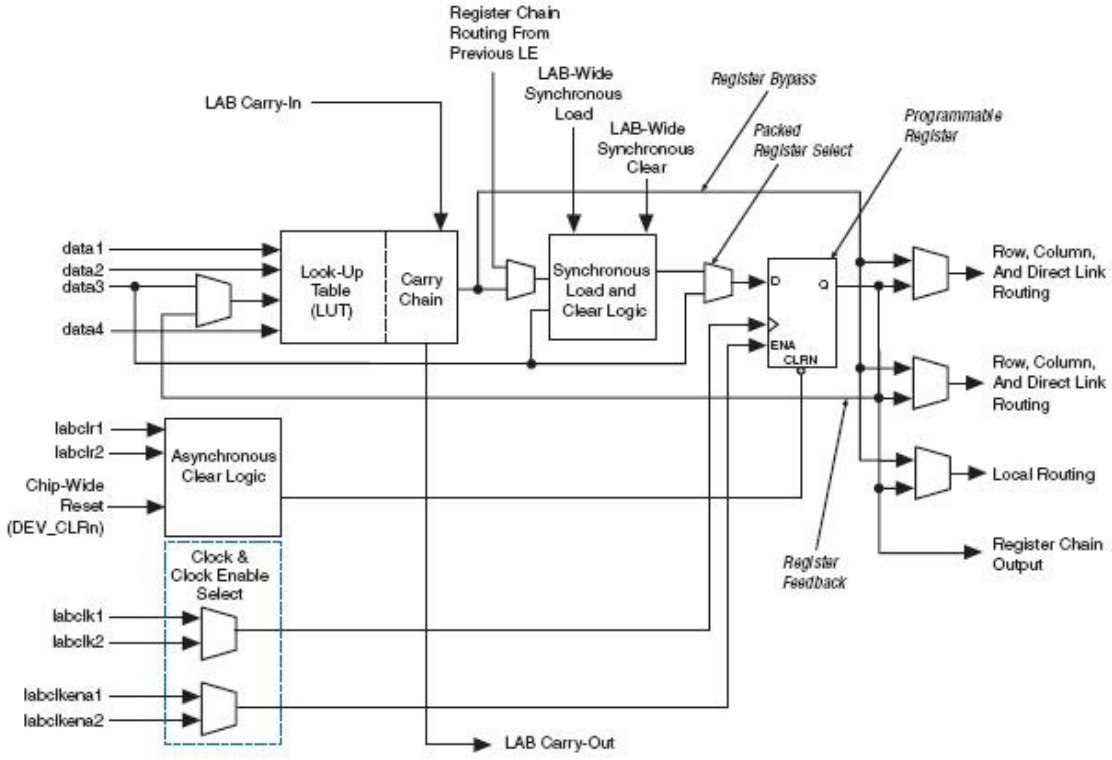


Figure 8 - Altera Cyclone II Logic Block

There are two methods for programming the FPGA on the board: active and passive. The passive mode serves to one-time program the FPGA for single use running. When the FPGA loses power again, it will revert to the saved program in memory. The active mode, in addition to configuring the FPGA for the program, also places the configuration data into the FPGA's memory such that when the FPGA is powered on the

next time it will remember this new configuration. For the purposes of this project, the FPGA was only configured in passive mode as it was unnecessary to fully reprogram the chip.

This project was designed using Quartus II software, version 6.1 [27], provided by Altera and utilized ModelSim [28] to perform timing analysis functions. Quartus II is an EDA tool used for design entry using VHDL or schematics, design synthesis, and device configuration for Altera-based FPGAs. ModelSim is a logic simulator for the hardware description language. It uses accurate timing models provided by Altera to emulate the Cyclone II device. All of the data reported here is obtained from ModelSim and then reorganized into Excel graphs. ModelSim works by receiving the actual programming file normally sent to the FPGA and then emulates the FPGA under test. ModelSim is used because there is no practical way to extract internal timing information directly from the FPGA.

In this section, we will focus on the transparency of the design. Transparency is tested by identifying any necessary modifications, if any, to the processor in order to implement the encryption successfully. This project is constructed with three elements: (1) processor, (2) encryption, and (3) memory. For each of the three elements, the basic design will be discussed. Then the section will describe any additional design considerations that were made in order to make it compatible with the encryption. Full schematics of the design as well as the corresponding VHDL code are available in the appendix.

1) Processor Design

The basic structure of this Reduced Instruction Set Computing (RISC) processor is simple and designed specifically to run basic programs to test the memory interface. It primarily consists of an ALU, a data register, and a program counter. At the beginning of each clock cycle, the program counter is incremented (Figure 9). The instruction (OP code) is then fetched and decoded into instruction information and data. The ALU then performs the current instruction based on the OP code fetched from memory. At the end of the cycle, any writes to memory or outputs occur. Finally, the ALU also provides a data register to store a single data string for use in the mathematical functions.

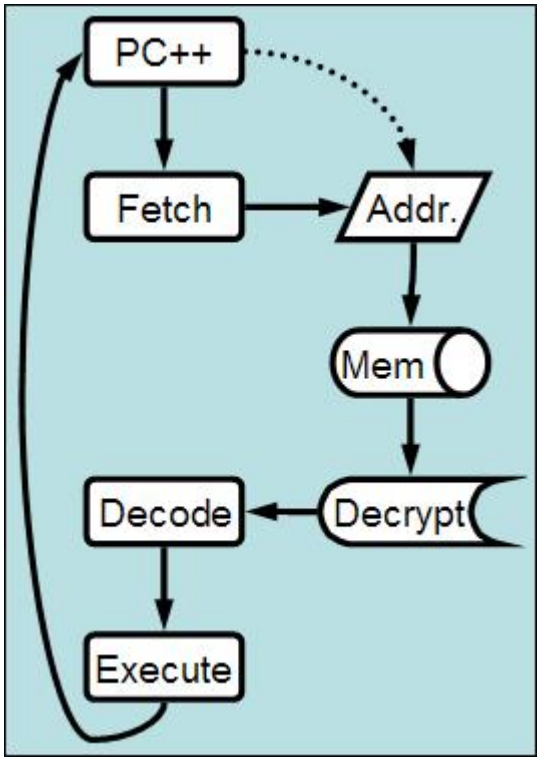


Figure 9 - Sample Processor Cycle

The addressable memory is 256K words, which maps to an 18-bit wide address. Therefore, internal memory addresses and the program counter are 18 bits wide. The word width of the memory is 16 bits wide. During the decode phase of operation, the processor reads in the 16-bit OP code and splits it into the instruction and data portions. For this RISC implementation, there are currently nine instructions, which require 4 bits to code and leave 12 bits for the immediate field. Of the ten instructions available to this processor, four utilize the immediate field of the OP code. Whenever an instruction does not need that field, it is set to all zeros.

Each of the different OP codes corresponds to a unique function in one of two categories: manipulation and mathematical. There are four manipulation functions that perform the interface and control portions of the processor. The LOAD function pulls the data portion of the OP code and puts it in the data register, overwriting the current content. The no-op function skips a processor cycle. An LEDPIO function moves the current contents of the holder to the light emitting diode processor input/output register. The LEDPIO controls the content of the LED display on the DE2 Board. The JUMP instruction operates by resetting the program counter, adding in the immediate field of the OP code, then finally re-feeding the address line. The four mathematical functions operate directly on the data register. The ADD instruction sums the immediate field with the data register and returns the result to the data register. The SUB instruction, likewise, subtracts the immediate field from the data register and returns it the data register. The INC and DEC instructions respectively increment and decrement the current contents of the data register by one. The final two uncategorized functions in the instruction set perform functions similar to no-op in the ALU. Their primary usages are for the program

counter and decoder. These OP codes, set to all zeros and all ones, signify the initialization states of the processor. When executed, the data register is reset to zero and the addresses are zeroed, similar to a reset. Additionally, when the all-zeros instruction is used, it halts the processor operation, signifying the end of the program code. Table 1 provides a full list of the instructions and their associated OP codes for the processor developed for this study.

Table 1: Instruction List for the RISC processor

Instruction	**Code**	**Detail**
END	0000	Stops processor operation until Initialize
ADD	0001	Addition of DATA to ALU Register
SUB	0010	Subtraction of DATA from ALU Register
Increment	0011	Increment the ALU Register by 1
Decrement	0100	Decrement the ALU Register by 1
NOOP	0101	No Operation, do nothing
LOAD	0110	Set ALU Register equal to DATA
Output	0111	Set output buffer to the contents of the ALU Register
GoTo	1000	Set the Program Counter equal to DATA
Initialize	1111	Initializes the processor, Reset Program Counter & ALU

The single largest limitation to this processor is the JUMP instruction: no program can be longer than 4095 individual instructions. Another limitation is the lack of branching instructions. This reduced the amount of advanced logic that can be utilized in

the test program. For this current usage, these limitations are unimportant to the entire goal of this project. However if this processor were to be used elsewhere, it would require a modification. More features that should be included would be a status register containing flags for various ALU conditions, instructions to handle branching conditions, and double-wide instructions to remove the limitations on the number of OP codes and instruction data width.

2) Memory Controller

The memory chip used is a generic SRAM chip included on the DE2 development kit for this project. The SRAM chip is a 256K by 16-bit memory array yielding a 16-bit data width and 18-bit addressing [29]. The worst case response time for this chip is 15 ns, resulting from the way in which the memory cell locks the values. The read cycle is longer than the write cycle. The memory chip cannot be simulated within ModelSim with respect to the processor and encryption, so it is assumed that the chip is working at worst-case speed at all times.

The controller for the memory has been implemented as the interface from the processor to the memory. The memory possesses several inverted control inputs. The inputs of most concern are the Chip Enable ($\overline{CE}$), the Output Enable ($\overline{OE}$), and the Address and Data (Dout) lines. The basic operation states that when the chip enable is clocked, it locks the value for the address, retrieves the contents of the specified memory cell, and returns the data to the data line. When the Output Enable is asserted, and the chip enable is clocked, the memory locks the address and the data lines and stores the value of the data line into the specified memory address. According to the table in Figure

10 provided from the datasheet, the read cycle incurs the largest amount of delay. This is important as read operations occur every processor cycle (i.e., the instruction fetch). Figure 10 also illustrates a typical read cycle for this SRAM chip

Symbol	Parameter	Min.	Max.	Unit
t_{RC}	Read Cycle Time	15	—	ns
t_{AA}	Address Access Time	—	15	ns
t_{OHA}	Output Hold Time	3	—	ns
t_{ACE}	$\overline{CE}$ Access Time	—	15	ns
t_{DOE}	$\overline{OE}$ Access Time	—	7	ns
t_{HZOE}(2)	$\overline{OE}$ to High-Z Output	0	6	ns
t_{LZOE}(2)	$\overline{OE}$ to Low-Z Output	0	—	ns
t_{HZCE}(2	$\overline{CE}$ to High-Z Output	0	8	ns
t_{LZCE}(2)	$\overline{CE}$ to Low-Z Output	3	—	ns
t_{BA}	$\overline{LB}$, $\overline{UB}$ Access Time	—	7	ns
t_{HZB}(2)	$\overline{LB}$, $\overline{UB}$ to High-Z Output	0	5	ns
t_{LZB}(2)	$\overline{LB}$, $\overline{UB}$ to Low-Z Output	0	—	ns
t_{PU}	Power Up Time	0	—	ns
t_{PD}	Power Down Time	—	15	ns

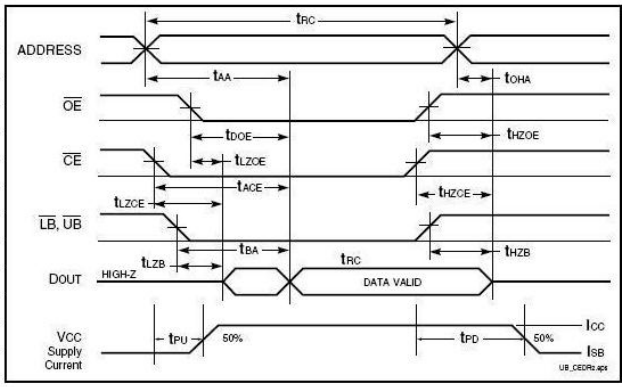


Figure 10 - SRAM Latency Table and Sample Read Cycle

3) Encryption Interface

While any encryption could be used theoretically, we are limited to block ciphers only. Typical communication protocols will use stream ciphers whereby each encrypted block will have its contents encrypted with relation not only to the key, but also to the previous block. Stream ciphers cannot be used in the processor-to-memory interface because memory requires random access. DES was chosen because it provides a reasonable amount of security strength for a block cipher without being encapsulated into a stream cipher.

The actual implementation of the DES algorithm occurred with both block diagrams and VHDL code. As discussed above, there are three main blocks: the Feistel block, the Key Generator, and the Permutations. Additionally, there is one extra block

which stores the key that is used in the encryption. Currently this key is set statically, but in theory it can be changed manually or configured with a full set of different keys.

First, the key generator operates by taking the main base key and calculating 16 sub keys for each of the 16 Feistel blocks. The sub keys are generated in a similar manner to the operation of the DES and begin by breaking the key in half. The base key in this implementation is 24 bits wide, differing from standard DES of 54, to accommodate the reduced data size of the data word (16 bits). The base key is broken into equal chunks of 12 and placed in two separate shift registers. A permutation function selects 6 bits from each shift register and assembles them in a random order to produce the sub key of length 12. At the end of each step, the shift registers rotate left by 1 bit before the process starts over again for the next sub key. This process repeats 16 times for each sub key.

The Feistel chain in this case operates exactly the same as a normal DES implementation, differing only from the standard block size of 64 bits. The data width starts at 16 bits and is split into two equal pieces of 8 bits each. The first half is fed into the XOR combination. The second half is fed into the Feistel block. The Feistel block operates by taking the 8 data bits, expanding it to twelve by adding zeros, and then XORs the new data with the 12-bit sub key from the key generator. Next, this new 12-bit block is broken into two groups of six bits and fed into the S-blocks. Each S-block is a look up table that transforms the six bits into four bits based on an arbitrary, and non-linear, association. Standard DES also uses a six-to-four transformation but increases the number of S-blocks to eight. The final step of the Feistel block involves putting the final eight re-combined bits through one final permutation. The resulting 8 bits goes into the XOR with the first half of the data. The next link in the Feistel chain utilizes the output

from the XOR as the input to the Feistel block, and the previous input is placed on the current XOR.

At the beginning and the end of the Feistel chain are the two permutation blocks. For this implementation, the blocks perform two functions. First, the permutation blocks will apply a symmetric randomization to the bits and break the block into its respective data halves for entry into the Feistel chain. Secondly the block will watch for the decoding flag and swap the data halves when in decryption mode. The permutation is designed to increase obfuscation even though it does not add any additional security strength to the DES block. Furthermore, the end permutation is designed to mirror the first one such that it is undone at the end of the Feistel chain.

There are some notable limitations to this implementation of DES. First and foremost comes from the size of the key and the data block [30][31]. Standard DES uses a 64-bit data block and a 56-bit key. From the data block alone, this significantly reduces the strength of the DES operation, making it much easier to brute force [21]. The data block is 16 bits because it is limited by the size of the memory data channel. Standard DES encryption strength relies on the sub-key permutation and the S-Blocks. The sub-key permutation determines the portion of the base key to utilize for a particular Feistel block. The S-blocks provide an arbitrary, non-deterministic bit alteration to the final key. Reducing the key size reduces the sub-key variance, which also reduces the cipher strength. Similarly, reducing the size of the S-Blocks limits their cryptographic significance. The key size was chosen to be 24 bits from working backwards through the DES implementation. It was sufficient in this case to maximize the permutation block while minimizing the resource utilization on the FPGA. Theoretically, the key can be of

any size and the S-blocks are utilized to reduce the size of the Feistel block output to the data width.

4) Test System

The finished product contains four parts: (1) the processor, (2) the memory controller, (3) the encryption block, and (4) the pre-program memory. The basic modules have connections between the processor and encryption, and between encryption and the memory controller. The pre-program memory, discussed in the next section, serves a special purpose and is connected directly to the processor.

Several design implementations for the final construction of the CPU/Encryption/Memory system were created to help accommodate the DES but maintain the same functionality as discussed above. Keeping in mind the end-to-end delay, at the beginning of the clock cycle, the data processed from the previous cycle must be completely stable before it can be used for the next cycle. In the first version of the system, clocking the processor triggered the program counter to increment followed by memory fetch, decode, and execution. While simple to program, this implementation possesses the longest end-to-end delay and is subject to variations in memory delay. It also varies the processing time for instructions. To improve on this design, two revisions were devised in order to reduce the delay by removing the time to read memory within the current processor cycle. The first revision operated by initiating the fetch cycle on clock high, which increments the program counter and sends signals to memory and returns the data through the encryption. On clock low, the processor takes the fetched data, decodes it, and executes the corresponding instruction. The second revision utilized

a form of pipelining by dividing the processor into two sections, fetch and execute. On clock high, the processor begins by decoding and executing the current instruction provided by memory. It then finishes by using the program counter to fetch the next instruction. In this method, the processor is always executing on the previous instruction, giving sufficient time to run the decryption; however, program length is increased through the inclusion of a NOOP after jumps, and the processor is initialized to a NOOP. Splitting up this design, it becomes easier to manage the dataflow for the processor by forcing all the sequences with the heaviest delay to occur in one half of the clock cycle and the processor to operate in the other half.

5) Test Setup

The test setup for this design takes place in two stages: (1) static code encryption and (2) code execution. In the first stage the processor reads the program data from pre-program static memory. Each clock cycle, the processor takes an instruction from the pre-program memory and sends it through the encryption to the memory controller and writes it to the SRAM chip. This process continues until the program finds the stop code of all-ones. After this, the processor enters execution mode where it begins by resetting the program counter and data register, and accesses the first instruction from the SRAM chip. Once in execution mode, it continues to perform whatever instructions it is doing until it sees the all-zeros, which signifies end-of-program.

CHAPTER V

ANALYSIS METHODOLOGY

For this experiment there are two goals. First is to gauge the performance hit from the addition of the encryption. Primarily, this was measured by examining the response time between the processor and memory, first the baseline without encryption and then with the encryption. The maximum clock speed and instructions per second can be calculated with this information. The end result is that the effectiveness of the encryption scheme can be evaluated. The second goal is to determine the amount of on-chip resources required to implement this design scheme. The implications which follow stem from size and cost; smaller and more inexpensive FPGAs can be utilized to perform the same task.

The test process was an iterative simulation using ModelSim, which provides a near-perfect virtualization of the FPGA design under examination. Each design model was recompiled before simulation. In order to test the maximum speed of this model, four values need to be known: (1) worst-case response for the ALU, (2) worst-case access time for memory, (3) worst-case response for the DES, (4) and average-case for DES. By summing the worst-case delays required to execute one instruction cycle, a general idea for the maximum clock cycle for this processor can be determined.

1) Baseline Performance Experiments

ALU:

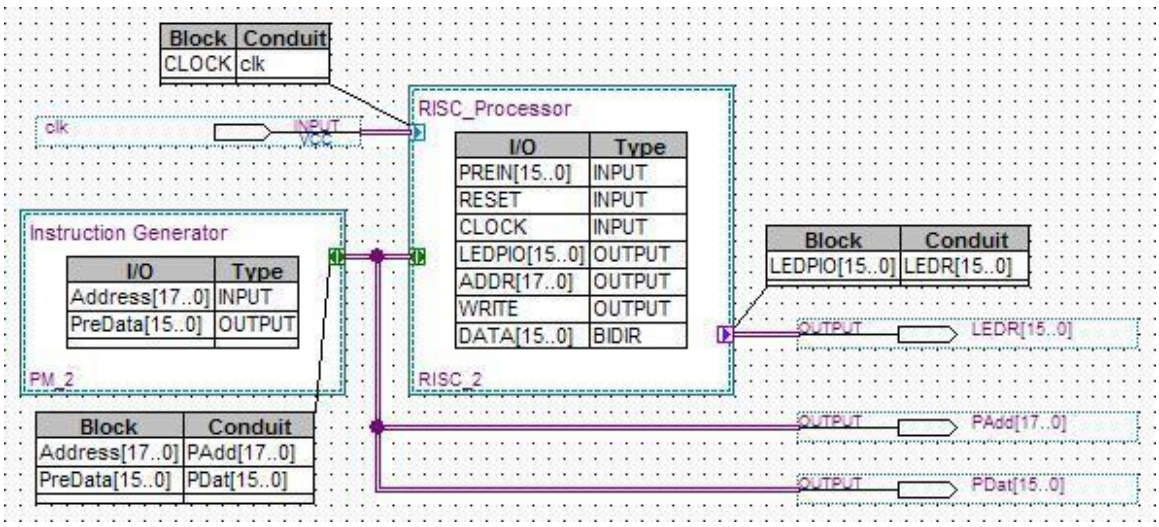


Figure 11 - Processor Test Setup
[Quartus II Block Diagram]

It is important to test to see how long it takes for the CPU to execute one full cycle and therefore determine its delay time. To do this, it was important to remove the encryption and memory from the equation and isolate the processor. To do so, the memory and encryption blocks were removed from the block diagram yielding the design in Figure 11. Since the instruction delay was the target of this test, each instruction was placed on the memory input prior to the processor being clocked for the current cycle. Replacing the memory, an instruction generator was inserted that ignored the addressing signals for the actual output; instead it used those signals as a clock to generate the next instruction. Understandably, each instruction will have a different delay time, and

therefore they needed to be tested individually. Each time the instruction generator was clocked, it would output the current test instruction and a randomized bit-set for the data portion.

Memory:

The testing of memory to prove that it operated within its ranges proved difficult, if not impossible. The only thing that could be tested is that it worked and how much additional overhead the memory controller added to the response time. The memory controller itself was written to be a simple interface. The majority of the data fed into it was pass-through, and the rest of the values were static. Testing the memory showed the memory controller added no additional time to memory operation. Therefore, the default values provided by the data sheet were assumed.

Encryption:

Testing the DES algorithm was accomplished in a similar manner to the processor: by isolating it from the other components. In order to determine the average and worst case response time for this implementation of the DES algorithm, the logic was fed a set of randomized values. Figure 12 shows the setup for the DES system. Included in this test was just the encryption routine. DES is a symmetric algorithm and therefore both the encryption and decryption process will take approximately the same amount of time to complete. The key used was generated randomly before being programmed into it. However the value of the key is not essential in calculating overall delay for this system. This stems from the fact that the key remains constant and the sub-key outputs

from the key generator will remain constant during the entirety of the operation. Additionally, changing the key will invalidate any data saved in memory and it is undesirable to change it during operation. Finally, from this data set, the amount of time it took for the DES output to register and the amount of time it took for the output to stabilize were recorded.

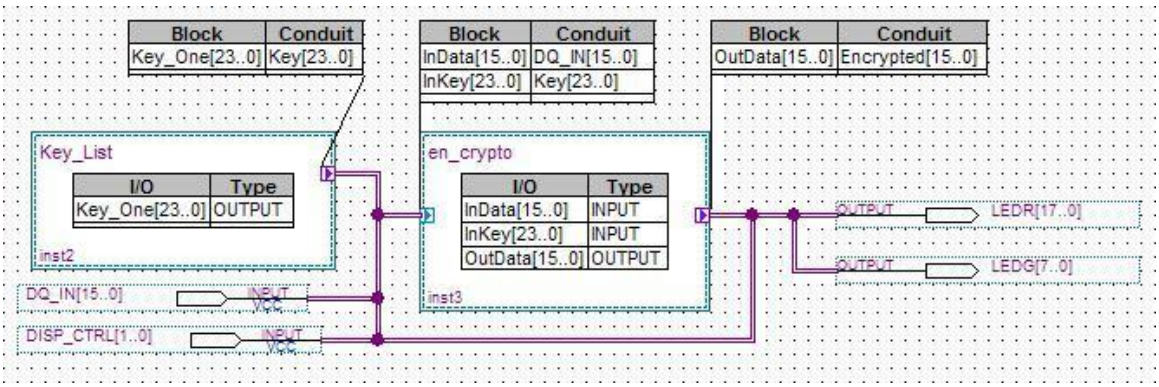


Figure 12 - Encryption Test Setup
[Quartus II Block-Diagram]

2) Complete System Setup

Re-assembling the parts for the full system test corresponds to Figure 13. The system was compiled for the Cyclone II FPGA on the DE2 kit then verified in hardware to show that all the experimental data was correct. If the encryption is working properly, then the hardware implementation is successful.

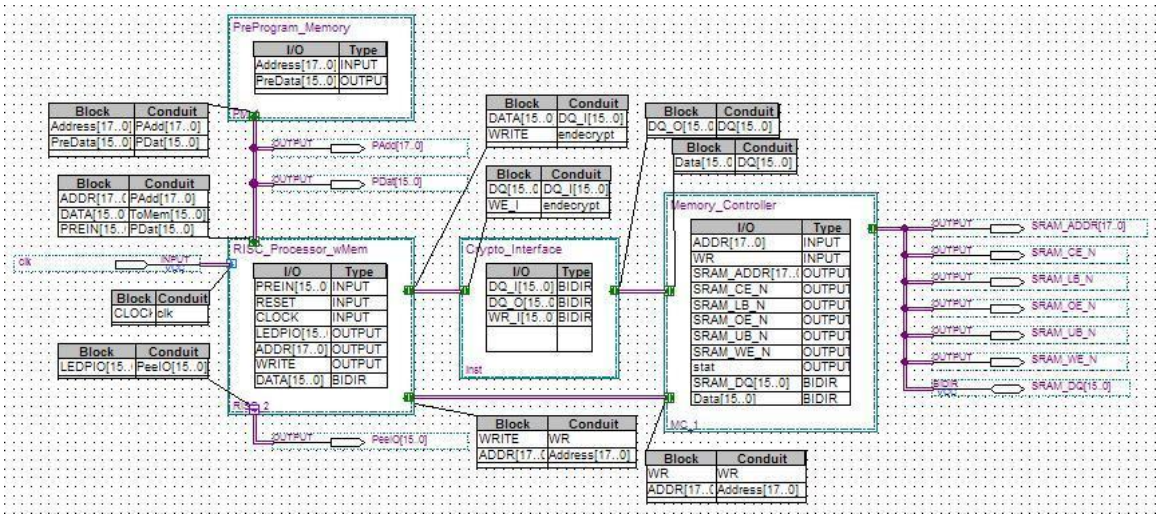


Figure 13 - Final Assembled System
[Quartus II Block Diagram]

CHAPTER VI

RESULTS AND ANALYSIS/DISCUSSION

The simulated tests for both the processor and the DES were constructed using the Quartus II software, version 6.1 [27] and simulated with both the Quartus tool and with ModelSim [28]. The total utilization of the Cyclone II FPGA chip came to approximately 10% of the total number of logic blocks available.

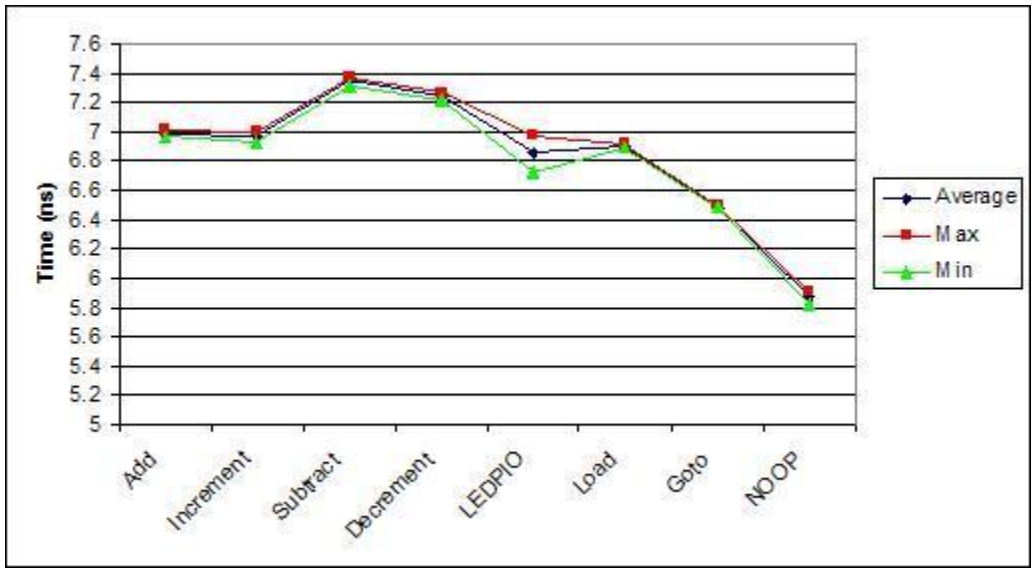


Figure 14 - Settle Time vs. Instruction Type

The chart in Figure 14 shows the average, minimum, and maximum response times for each instruction in the processor. Each instruction was sampled five times, and from the timing analysis the average, minimum and maximum times were calculated. More samples did not change the deviation. Derived from the data, the quickest

instruction is the NO-OP at 5.862 ns while the worst-case instruction is Subtract at 7.375 ns. The average execution time for this processor is 6.828 ns. The instruction with the highest variance is the LEDPIO due to it interfacing with the control bank. The end result shows that the worst possible case for the processor, which equates to Subtract, comes out to be approximately 7.4 ns.

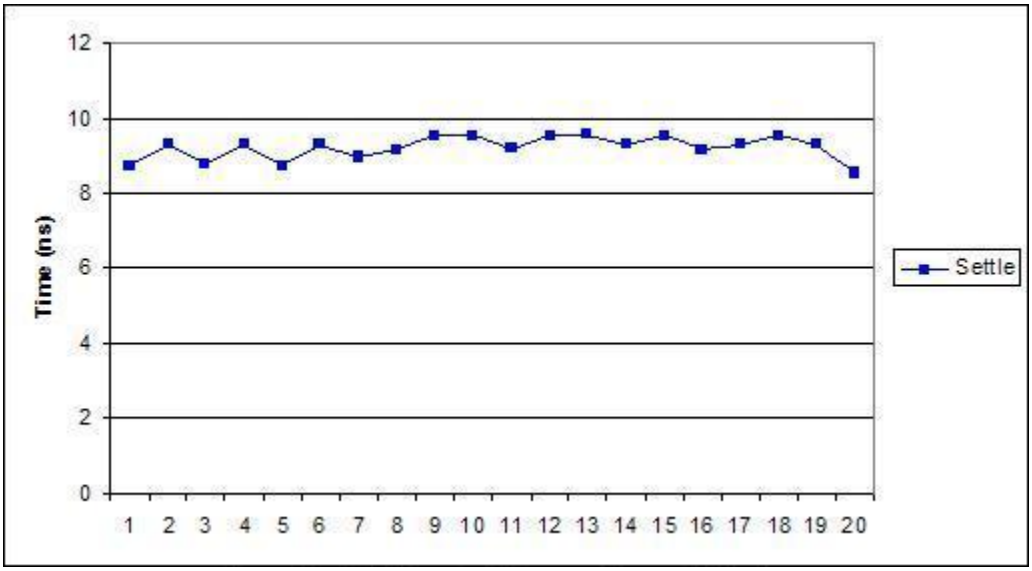


Figure 15 - Sampled Stabilization Time

The DES testing provided similarly useful results. Prior inspection dictated that the maximum response time for each item would be approximately 10 ns, therefore the waveform generator was set to change patterns to a new random value every 20 ns to prevent any outliers from causing inconsistencies in subsequent data points. Starting at 0 ns up to 400 ns, 21 data points were taken as displayed in Figure 15. Each data point shows the amount of time taken for the data, after being clocked, to stabilize into a usable output. As can be seen, there is not much deviation among the data. Average time from

start to stable is 9.204 ns, the deviation is 0.311 ns, and the worst case is 9.558 ns. Using this data, the worst-case response can be set to no more than 10 ns when coupled with the other components.

The full-scale system analysis requires summing each of the worst-case times for the components: 10 ns from the DES, 7.4 ns from the CPU, and 15 ns from the Memory. This brings the total delay to 32.5 ns for a worst-case cycle. The DE2 development board used for testing this design provides two hardware clocks: 50 MHz and 27.5 MHz. Making this equivalent to the cycle time, the 50 MHz allows for a delay up to 20 ns while the 27.5 MHz clock provides approximately 36.37 ns. With the baseline limitation of 22.5 ns, the 50 MHz clock cannot be used. Adding the encryption increases the cycle from 22.5 to 33.5 ns which remains within the 36 ns.

Within the scope of this project, the data shows that it is feasible to add encryption to the processor-to-memory interface while not altering the functionality of the processor under test. That being said, while it did not alter the functionality, it did affect the maximum speed at which this processor could operate. Any amount of additional encryption provided to the memory channel must be given some extra delay time in order to perform the necessary calculations.

CHAPTER VII

CONCLUSION

Trustworthy computer systems protect the access of sensitive information by an unauthorized agent (i.e., an attacker). However, security vulnerabilities exist which can allow a system to be compromised by an attacker. Information can be accessed or altered by malicious efforts when it leaves the boundary of trust (e.g., the microprocessor). The purpose of this work was to design an encryption module, using an FPGA, which would serve as in interface between a processor and its memory. The goal was to determine whether this could be done at a minimum of interference to the proper operation of a processor. An FPGA was used as the hardware platform for the prototype of the test system. The greatest challenges were in creating a working processor and memory architecture.

This thesis presented a method to add DES encryption to a processor-to-memory interface. Based on the testing results, there will be a performance penalty due to the encryption. However, the latency for encryption can be hidden within the normal operation of the test processor. In order to allow for this encryption, an adjustment of the clock timing was not required over the performance of the base system. While this encryption module does not address all the security challenges of the processor-to-memory interface, it does provide some protection from download attacks to the physical memory. In addition, the flexibility of the FPGA platform achieves the goal of

transparency by incorporating additional security features into the existing system without extensive modification.

The test system can be used as a basis to draw some insights into large-scale systems. The basic RISC processor used in this project can only perform simple mathematical operations and read data from memory. An extension to this work would incorporate one of the numerous embedded processor designs into the FPGA to replace this test processor. Prior to using the current design, a NIOS II processor from Altera [32] was examined; however, it failed integrate properly into the test system. Since an advanced processor was not a requirement for this project, the processor was scaled down to its current form in order to obtain a functional system. This test system also assumed a single-chip implementation where all components resided on the FPGA.

To extend this to a more modular design, the application would utilize the FPGA itself as the bridge containing the encryption while processor and memory are independent devices. With more advanced processors, the timing of the memory access becomes extremely important because memory access is typically the performance bottleneck. Ten nanoseconds, which equates to a maximum speed of around 100MHz, would severely limit the speed of the processor. Should this be utilized on a standard CPU, placing the encryption module as an interface between cache and main memory would be a more reasonable choice. However, the encryption module should remain within the trusted boundary of the processor. This would improve performance by encrypting an entire cache line instead of individual memory accesses. The DES algorithm could be used at full strength (i.e., 64 bits) to encrypt an entire cache line. Another advantage to separating the CPU from the FPGA is the relaxation of the resource

requirements for the encryption. This, in turn, would enable more complexity in the security module (i.e., stronger encryption). The fundamental point to remember is that security adds operational latency, and in high speed designs any extra latency reduces the maximum potential speed at which the processor can operate.

REFERENCES

[1] Samyd, D. et al., “On a new way to read data from memory” *First International IEEE Security in Storage Workshop*, 2002

[2] Mahapatra, N. R., Venkatrao, B., ‘The processor-memory bottleneck: problems and solutions” *Crossroads*, ACM Press, Vol. 5, Issue 3, 1999

[3] Wollinger, T., Guajardo, J., & Paar, C., “Security on FPGAs: State-of-the-Art Implementations and Attacks” *ACM Transactions on Embedded Computing Systems*, Vol. 3, No. 3, Aug. 2004

[4] Smid, M. E. & Branstad, D. K., “Data Encryption Standard: Past and Future” *Proceedings of the IEEE*, Vol. 76, No. 5, May 1988

[5] Stallings, William, “Cryptography and Network Security: Principles and Practice” *Prentice Hall College*, 2006

[6] Chang, R. K. C., “Defending against flooding-based distributed denial-of-service attacks: a tutorial” *IEEE Communications Magazine*, Oct. 2002

[7] Rose, J., El Gamal, A., and Sangiovanni-Vincentelli, A., "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, pp. 1013-1029, 1993.

[8] Smith, Michael John Sebastian, Application-Specific Integrated Circuits – The Book *Addison-Wesley Publishing Co.*, June 1997

[9] Maxfield, Clive, The Design Warrior's Guide to FPGAs: *Elsevier*, 2004.

[10] Betz, V., and Rose, J., "Circuit design, transistor sizing and wire layout of FPGA interconnect," in *1999 Custom Integrated Circuits Conference*, 1999, pp. 171-174.

[11] Ashenden, Peter J., The Designers Guide to VHDL *Morgan Kaufmann Publishers*, 1998

[12] Palnitkar, Samir, Verilog HDL (2nd Edition) *Prentice Hall*, 2003

[13] Standard 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability, IEEE, 1993.

[14] Greene, J., Hamdy, E., and Beal, S., "Antifuse field programmable gate arrays," *Proceedings of the IEEE*, vol. 81, pp. 1042-1056, 1993.

[15] Zambreno, J., Honbo, D., Choudhary, A., Simha, R., and Narahari, B., "High-performance software protection using reconfigurable architectures," *Proceedings of the IEEE*, vol. 94, pp. 419-431, 2006.

[16] Smith, S. W. & Weingart, S., "Building a high-performance, programmable secure coprocessor" *Computer Networks* Vol. 31, 1999

[17] Crowe, F., Daly, A., Kerins, T., & Mamane, W., "Single-Chip FPGA Implementation of a Cryptographic Co-Processor" *IEEE International Conference on Field-Programmable Technology*, 2004

[18] Wollinger, T. & Paar, C., "How Secure Are FPGAs in Cryptographic Applications?" *Lecure Notes in Computer Science* 2003

[19] Standaert, F.-X., Peeters, E., Rouvroy, G., and Quisquater, J.-J., "An overview of power analysis attacks against field programmable gate arrays," *Proceedings of the IEEE*, vol. 94, pp. 383-394, 2006.

[20] Standaert, F. X., tot Oldenzeel, L. O., Samyde, D., and Quisquater, J. J., "Power Analysis of FPGAs: How Practical is the Attack?" *Lecture Notes in Computer Science*, 2003

[21] Coppersmith, D., "The Data Encryption Standard (DES) and its strength against attacks" *IBM Journal of Research & Development* Vol. 38 Issue 3, May 1994

[22] Merkle, R. C. & Hellman, M. E., "On the Security of Multiple Encryption" *Communications of the ACM*, Vol. 24, No. 7, July 1981

[23] Coppersmith, D., Johnson, D. B., Matyas, S. M., "A proposed mode for triple-DES Encryption" *IBM Journal of Research and Development*, Vol. 40, Issue 2, 1996

[24] Wong, K., Wark, M., & Dawson, E., "A Single-Chip Implementation of the Data Encryption Standard (DES) Algorithm" *IEEE Global Telecommunications Conference*, 1998

[25] Altera's Development and Education Board (DE2), http://www.altera.com/education/univ/materials/boards/unv-de2-board.html

[26] Altera Cyclone II Device Handbook, http://altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf

[27] Quartus II 6.1, http://altera.com/literature/lit-qts.jsp

[28] ModelSim 6.1, http://www.model.com/resources/resources_manuals.asp

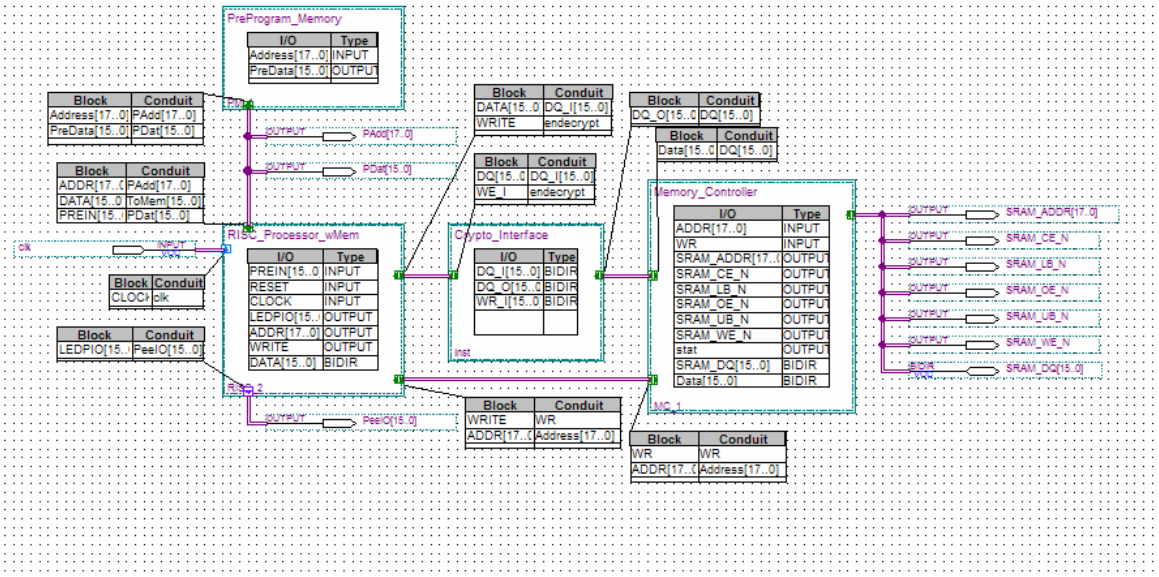
[29] ISSI IS61LV25616 Specification Sheet, 256k x 16 High Speed Asynchronous CMOS Static Ram with 3.3V Supply

[30] Blaze, M., Diffie, W., Rivest, R. L., Schneier, B., Shimomura, T., Thompson, E., & Wiener, M., "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security" *A Report by an Ad Hoc Group of Cryptographers and Computer Scientists*, Jan. 1996

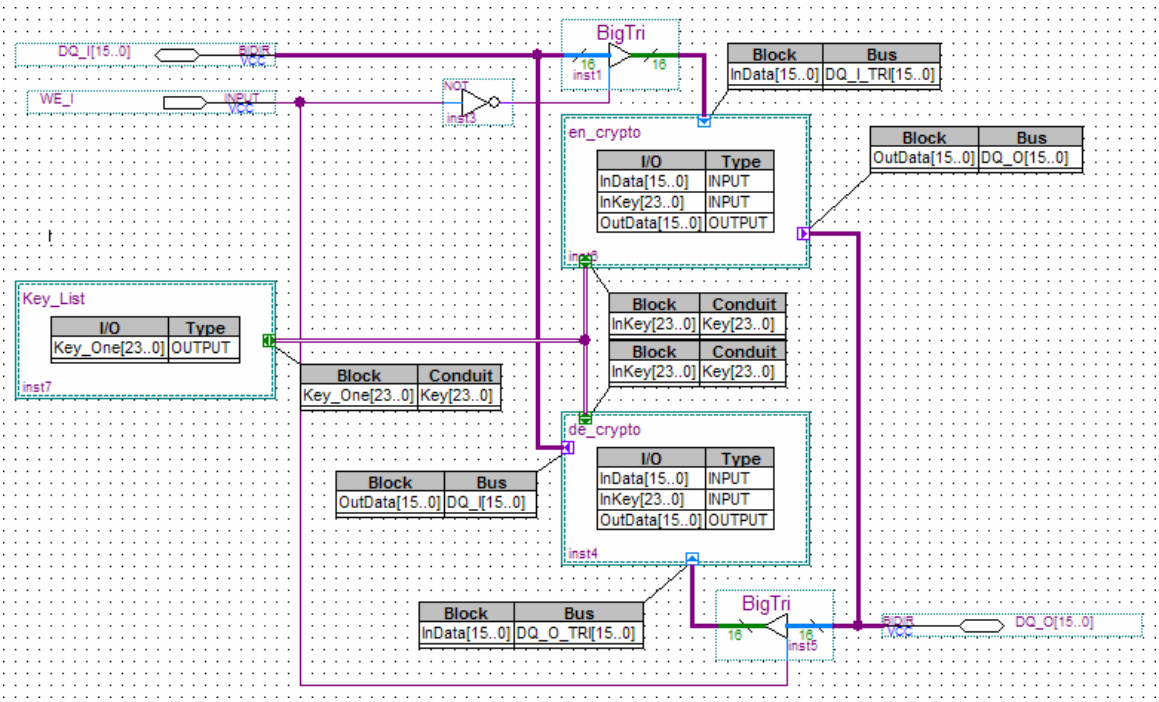
[31] Lenstra, A. K. & Verheul, E. R., "Selecting Cryptographic Key Sizes" *Journal of Cryptology*, Vol. 14, 2001

[32] NIOS II, http://altera.com/literature/lit-nio2.jsp

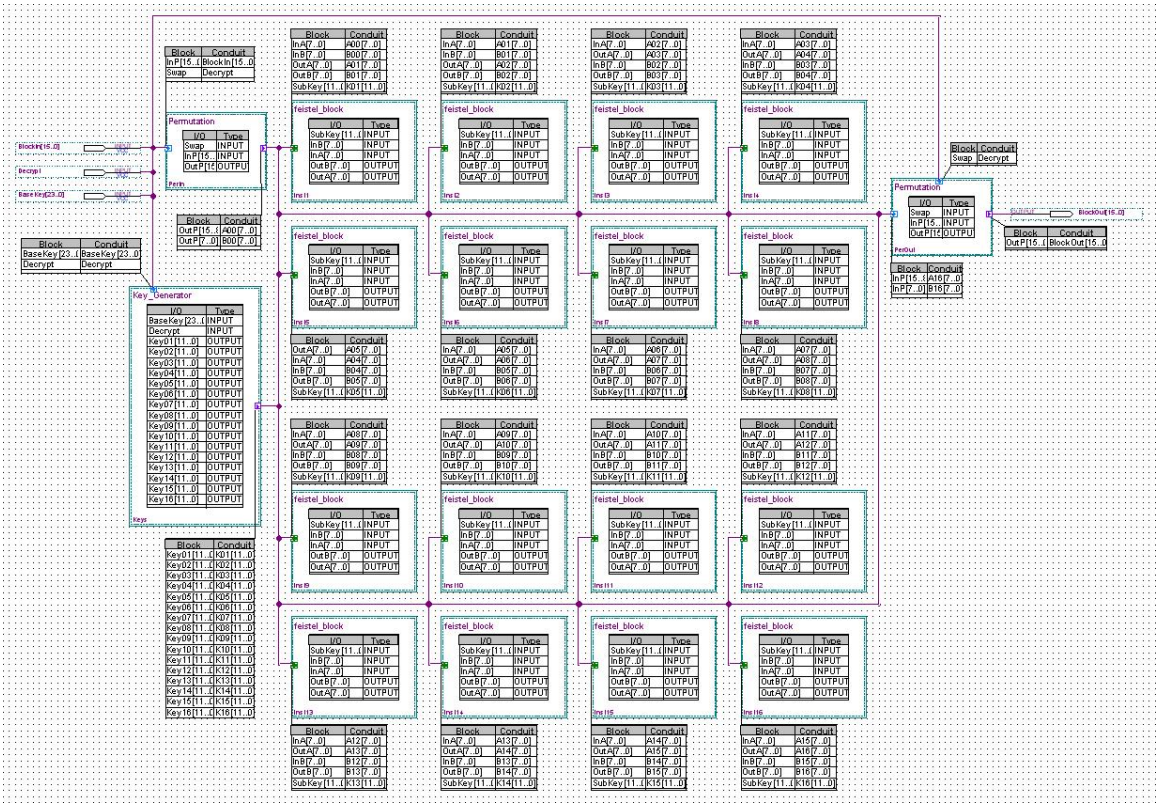
APPENDIX OF FIGURES AND VHDL CODE



Complete System Block Diagram



Cryptography Interface Block



Feistel Chain (inside the Encrypt and Decrypt blocks)

Permutation Block VHDL Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


--  Entity Declaration

ENTITY Permutation IS
	-- {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
	PORT
	(
		Swap : IN  STD_LOGIC;
		InP  : IN  STD_LOGIC_VECTOR(15 downto 0);
		OutP : OUT STD_LOGIC_VECTOR(15 downto 0)
	);
	-- {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

END Permutation;


--  Architecture Body

ARCHITECTURE Permutation_architecture OF Permutation IS
BEGIN
	PROCESS(Swap, InP)
	BEGIN
		IF Swap='1' THEN
			OutP(15 downto 8) <= InP( 7 downto 0);
			OutP( 7 downto 0) <= InP(15 downto 8);
		ELSE
			OutP <= InP;
		END IF;
	END PROCESS;
END Permutation_architecture;
```

Key Generator Block Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

--  Entity Declaration

ENTITY Key_Generator IS
        -- {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
        PORT
        (
                BaseKey : IN  STD_LOGIC_VECTOR(23 downto 0);
                Decrypt : IN  STD_LOGIC;
                Key01   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key02   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key03   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key04   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key05   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key06   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key07   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key08   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key09   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key10   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key11   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key12   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key13   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key14   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key15   : OUT STD_LOGIC_VECTOR(11 downto 0);
                Key16   : OUT STD_LOGIC_VECTOR(11 downto 0)
        );
        -- {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

END Key_Generator;


--  Architecture Body

ARCHITECTURE Key_Generator_architecture OF Key_Generator IS
        SIGNAL ka0, kb0       : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k1,  ka1,  kb1  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k2,  ka2,  kb2  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k3,  ka3,  kb3  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k4,  ka4,  kb4  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k5,  ka5,  kb5  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k6,  ka6,  kb6  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k7,  ka7,  kb7  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k8,  ka8,  kb8  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k9,  ka9,  kb9  : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k10, ka10, kb10 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k11, ka11, kb11 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k12, ka12, kb12 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k13, ka13, kb13 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k14, ka14, kb14 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k15, ka15, kb15 : STD_LOGIC_VECTOR(11 downto 0);
        SIGNAL k16, ka16, kb16 : STD_LOGIC_VECTOR(11 downto 0);
BEGIN
```

```
PROCESS (BaseKey, Decrypt)
BEGIN
        -- Generate base-key pair
        ka0  <= BaseKey(23 downto 12);
        kb0  <= BaseKey(11 downto  0);
-- One
        -- Rotate to generate first sub-key pair
        ka1  <= ka0(10 downto 0) & ka0(11);
        kb1  <= kb0(10 downto 0) & kb0(11);
        -- Permutate sub-key
        k1   <= ka1(8) & kb1(10) & ka1(2) & kb1(6) & ka1(4) & kb1(11)
                & ka1(11) & kb1(0) & ka1(5) & kb1(9) & ka1(10) & kb1(7);
-- Two
        -- Rotate to generate second sub-key pair
        ka2  <= ka1(10 downto 0) & ka1(11);
        kb2  <= kb1(10 downto 0) & kb1(11);
        -- Calculate sub-key
        k2   <= ka2(8) & kb2(10) & ka2(2) & kb2(6) & ka2(4) & kb2(11)
                & ka2(11) & kb2(0) & ka2(5) & kb2(9) & ka2(10) & kb2(7);
-- Three
        -- Rotate to generate first sub-key pair
        ka3  <= ka2(10 downto 0) & ka2(11);
        kb3  <= kb2(10 downto 0) & kb2(11);
        -- Calculate sub-key
        k3   <= ka3(8) & kb3(10) & ka3(2) & kb3(6) & ka3(4) & kb3(11)
                & ka3(11) & kb3(0) & ka3(5) & kb3(9) & ka3(10) & kb3(7);
-- Four
        -- Rotate to generate first sub-key pair
        ka4  <= ka3(10 downto 0) & ka3(11);
        kb4  <= kb3(10 downto 0) & kb3(11);
        -- Calculate sub-key
        k4   <= ka4(8) & kb4(10) & ka4(2) & kb4(6) & ka4(4) & kb4(11)
                & ka4(11) & kb4(0) & ka4(5) & kb4(9) & ka4(10) & kb4(7);
-- Five
        -- Rotate to generate first sub-key pair
        ka5  <= ka4(10 downto 0) & ka4(11);
        kb5  <= kb4(10 downto 0) & kb4(11);
        -- Calculate sub-key
        k5   <= ka5(8) & kb5(10) & ka5(2) & kb5(6) & ka5(4) & kb5(11)
                & ka5(11) & kb5(0) & ka5(5) & kb5(9) & ka5(10) & kb5(7);
-- Six
        -- Rotate to generate first sub-key pair
        ka6  <= ka5(10 downto 0) & ka5(11);
        kb6  <= kb5(10 downto 0) & kb5(11);
        -- Calculate sub-key
        k6   <= ka6(8) & kb6(10) & ka6(2) & kb6(6) & ka6(4) & kb6(11)
                & ka6(11) & kb6(0) & ka6(6) & kb6(9) & ka6(10) & kb6(7);
-- Seven
        -- Rotate to generate first sub-key pair
        ka7  <= ka6(10 downto 0) & ka6(11);
        kb7  <= kb6(10 downto 0) & kb6(11);
        -- Calculate sub-key
        k7   <= ka7(8) & kb7(10) & ka7(2) & kb7(6) & ka7(4) & kb7(11)
                & ka7(11) & kb7(0) & ka7(5) & kb7(9) & ka7(10) & kb7(7);
-- Eight
        -- Rotate to generate first sub-key pair
```

```
        ka8  <= ka7(10 downto 0) & ka7(11);
        kb8  <= kb7(10 downto 0) & kb7(11);
        -- Calculate sub-key
        k8   <= ka8(8) & kb8(10) & ka8(2) & kb8(6) & ka8(4) & kb8(11)
                & ka8(11) & kb8(0) & ka8(5) & kb8(9) & ka8(10) & kb8(7);
-- Nine
        -- Rotate to generate first sub-key pair
        ka9  <= ka8(10 downto 0) & ka8(11);
        kb9  <= kb8(10 downto 0) & kb8(11);
        -- Calculate sub-key
        k9   <= ka9(8) & kb9(10) & ka9(2) & kb9(6) & ka9(4) & kb9(11)
                & ka9(11) & kb9(0) & ka9(5) & kb9(9) & ka9(10) & kb9(7);
-- Ten
        -- Rotate to generate first sub-key pair
        ka10 <= ka9(10 downto 0) & ka9(11);
        kb10 <= kb9(10 downto 0) & kb9(11);
        -- Calculate sub-key
        k10  <= ka10(8) & kb10(10) & ka10(2) & kb10(6) & ka10(4) & kb10(11)
                & ka10(11) & kb10(0) & ka10(5) & kb10(9) & ka10(10) & kb10(7);
-- Eleven
        -- Rotate to generate first sub-key pair
        ka11 <= ka10(10 downto 0) & ka10(11);
        kb11 <= kb10(10 downto 0) & kb10(11);
        -- Calculate sub-key
        k11  <= ka11(8) & kb11(10) & ka11(2) & kb11(6) & ka11(4) & kb11(11)
                & ka11(11) & kb11(0) & ka11(5) & kb11(9) & ka11(10) & kb11(7);
-- Twelve
        -- Rotate to generate first sub-key pair
        ka12 <= ka11(10 downto 0) & ka11(11);
        kb12 <= kb11(10 downto 0) & kb11(11);
        -- Calculate sub-key
        k12  <= ka12(8) & kb12(10) & ka12(2) & kb12(6) & ka12(4) & kb12(11)
                & ka12(11) & kb12(0) & ka12(5) & kb12(9) & ka12(10) & kb12(7);
-- Thirteen
        -- Rotate to generate first sub-key pair
        ka13 <= ka12(10 downto 0) & ka12(11);
        kb13 <= kb12(10 downto 0) & kb12(11);
        -- Calculate sub-key
        k13  <= ka13(8) & kb13(10) & ka13(2) & kb13(6) & ka13(4) & kb13(11)
                & ka13(11) & kb13(0) & ka13(5) & kb13(9) & ka13(10) & kb13(7);
-- Fourteen
        -- Rotate to generate first sub-key pair
        ka14 <= ka13(10 downto 0) & ka13(11);
        kb14 <= kb13(10 downto 0) & kb13(11);
        -- Calculate sub-key
        k14  <= ka14(8) & kb14(10) & ka14(2) & kb14(6) & ka14(4) & kb14(11)
                & ka14(11) & kb14(0) & ka14(5) & kb14(9) & ka14(10) & kb14(7);
-- Fifteen
        -- Rotate to generate first sub-key pair
        ka15 <= ka14(10 downto 0) & ka14(11);
        kb15 <= kb14(10 downto 0) & kb14(11);
        -- Calculate sub-key
        k15  <= ka15(8) & kb15(10) & ka15(2) & kb15(6) & ka15(4) & kb15(11)
                & ka15(11) & kb15(0) & ka15(5) & kb15(9) & ka15(10) & kb15(7);
-- Sixteen
        -- Rotate to generate first sub-key pair
```

```
                ka16 <= ka15(10 downto 0) & ka15(11);
                kb16 <= kb15(10 downto 0) & kb15(11);
                -- Calculate sub-key
                k16  <= ka16(8) & kb16(10) & ka16(2) & kb16(6) & ka16(4) & kb16(11)
                        & ka16(11) & kb16(0) & ka16(5) & kb16(9) & ka16(10) & kb16(7);
        -- Output based on Encryption/Decryption selector
                IF Decrypt='1' THEN
                -- Decrypt Keys
                        Key01 <= k16;
                        Key02 <= k15;
                        Key03 <= k14;
                        Key04 <= k13;
                        Key05 <= k12;
                        Key06 <= k11;
                        Key07 <= k10;
                        Key08 <= k9;
                        Key09 <= k8;
                        Key10 <= k7;
                        Key11 <= k6;
                        Key12 <= k5;
                        Key13 <= k4;
                        Key14 <= k3;
                        Key15 <= k2;
                        Key16 <= k1;
                ELSE
                -- Encrypt Keys
                        Key01 <= k1;
                        Key02 <= k2;
                        Key03 <= k3;
                        Key04 <= k4;
                        Key05 <= k5;
                        Key06 <= k6;
                        Key07 <= k7;
                        Key08 <= k8;
                        Key09 <= k9;
                        Key10 <= k10;
                        Key11 <= k11;
                        Key12 <= k12;
                        Key13 <= k13;
                        Key14 <= k14;
                        Key15 <= k15;
                        Key16 <= k16;
                END IF;
        END PROCESS;
END Key_Generator_architecture;

------------------------
-- Sub-Key Permutation
--
-- 11 <= a8        10 <= b10
-- 09 <= a2        08 <= b6
-- 07 <= a4        06 <= b11
-- 05 <= a11       04 <= b0
-- 03 <= a5        02 <= b9
-- 01 <= a10       00 <= b7
```

Feistel_Block VHDL Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


--  Entity Declaration

ENTITY feistel_block IS
	-- {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
	PORT
	(
		SubKey : IN  STD_LOGIC_VECTOR(11 downto 0);
		InB    : IN  STD_LOGIC_VECTOR(7  downto 0);
		InA    : IN  STD_LOGIC_VECTOR(7  downto 0);
		OutB   : OUT STD_LOGIC_VECTOR(7  downto 0);
		OutA   : OUT STD_LOGIC_VECTOR(7  downto 0)
	);
	-- {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

END feistel_block;


--  Architecture Body

ARCHITECTURE feistel_block_architecture OF feistel_block IS
	SIGNAL f1     : STD_LOGIC_VECTOR(11 downto 0);
	SIGNAL f2, f3 : STD_LOGIC_VECTOR(7  downto 0);
BEGIN
	PROCESS (InA, InB, SubKey, f1, f2, f3)
	BEGIN
		-- First Feistel function step (XOR)
		f1   <= SubKey XOR InB;
		-- Second Feistel function step (S-Box LUT, see below)
		-- S1
		CASE f1(11 downto 9) IS
			WHEN "111" => f2(7 downto 6) <= "00";
			WHEN "110" => f2(7 downto 6) <= "01";
			WHEN "101" => f2(7 downto 6) <= "10";
			WHEN "100" => f2(7 downto 6) <= "11";
			WHEN "011" => f2(7 downto 6) <= "00";
			WHEN "010" => f2(7 downto 6) <= "01";
			WHEN "001" => f2(7 downto 6) <= "10";
			WHEN "000" => f2(7 downto 6) <= "11";
		END CASE;
		-- S2
		CASE f1(8  downto 6) IS
			WHEN "111" => f2(5 downto 4) <= "00";
			WHEN "110" => f2(5 downto 4) <= "01";
			WHEN "101" => f2(5 downto 4) <= "10";
			WHEN "100" => f2(5 downto 4) <= "11";
			WHEN "011" => f2(5 downto 4) <= "00";
			WHEN "010" => f2(5 downto 4) <= "01";
			WHEN "001" => f2(5 downto 4) <= "10";
			WHEN "000" => f2(5 downto 4) <= "11";
```

```
                END CASE;
                -- S3
                CASE f1(5  downto 3) IS
                        WHEN "111" => f2(3 downto 2) <= "00";
                        WHEN "110" => f2(3 downto 2) <= "01";
                        WHEN "101" => f2(3 downto 2) <= "10";
                        WHEN "100" => f2(3 downto 2) <= "11";
                        WHEN "011" => f2(3 downto 2) <= "00";
                        WHEN "010" => f2(3 downto 2) <= "01";
                        WHEN "001" => f2(3 downto 2) <= "10";
                        WHEN "000" => f2(3 downto 2) <= "11";
                END CASE;
                -- S4
                CASE f1(2  downto 0) IS
                        WHEN "111" => f2(1 downto 0) <= "00";
                        WHEN "110" => f2(1 downto 0) <= "01";
                        WHEN "101" => f2(1 downto 0) <= "10";
                        WHEN "100" => f2(1 downto 0) <= "11";
                        WHEN "011" => f2(1 downto 0) <= "00";
                        WHEN "010" => f2(1 downto 0) <= "01";
                        WHEN "001" => f2(1 downto 0) <= "10";
                        WHEN "000" => f2(1 downto 0) <= "11";
                END CASE;
                -- Third Feistel function step (permutation)
                f3(7)<=f2(6);
                f3(6)<=f2(4);
                f3(5)<=f2(2);
                f3(4)<=f2(0);
                f3(3)<=f2(7);
                f3(2)<=f2(5);
                f3(1)<=f2(3);
                f3(0)<=f2(1);
                -- Final step in block, XOR and Output
                OutA <= InB;
                OutB <= f3 XOR InA;
        END PROCESS;
END feistel_block_architecture;

-- This Feistel function is an adaptation of the DES function
-- XOR -> S-Block -> Permutation
--
-- use four sets of two for output
-- Input : 109 876 543 201  (12-bit)
-- Output:  76  54  32  10   (8-bit)
--
-- S-Box LUT:
-- 111 = 00         110 = 01
-- 101 = 10         100 = 11
-- 011 = 00         010 = 01
-- 001 = 10         000 = 11
--  IN = OUT       IN = OUT
--
-- Permutation
-- 0 1 2 3 4 5 6 7
-- 1 3 5 7 0 2 4 6
```

Processor Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--  Entity Declaration

ENTITY RISC_Processor_wMem IS
	-- {{ALTERA_IO_BEGIN}} DO NOT REMOVE THIS LINE!
	PORT
	(
		PREIN  : IN    STD_LOGIC_VECTOR(15 downto 0);
		RESET  : IN    STD_LOGIC;
		CLOCK  : IN    STD_LOGIC;
		LEDPIO : OUT   STD_LOGIC_VECTOR(15 downto 0);
		ADDR   : OUT   STD_LOGIC_VECTOR(17 downto 0);
		WRITE  : OUT   STD_LOGIC;
		DATA   : INOUT STD_LOGIC_VECTOR(15 downto 0)
	);
	-- {{ALTERA_IO_END}} DO NOT REMOVE THIS LINE!

END RISC_Processor_wMem;


--  Architecture Body

ARCHITECTURE RISC_Processor_wMem_architecture OF RISC_Processor_wMem IS


BEGIN
	-- ALU Process
	PROCESS (CLOCK)
		-- Procesor Register(s)
		VARIABLE holder : STD_LOGIC_VECTOR(11 downto 0)
					:= "000000000000";
		-- Program Command
		VARIABLE opcode : STD_LOGIC_VECTOR( 3 downto 0);
		-- Program Data
		VARIABLE opdata : STD_LOGIC_VECTOR(11 downto 0);

		VARIABLE rst    : STD_LOGIC := '1';
		VARIABLE prog   : STD_LOGIC := '1';
		VARIABLE pcount : STD_LOGIC_VECTOR(17 downto 0)
					:= "000000000000000000";
	BEGIN
		IF (CLOCK'event and CLOCK = '1') THEN
			ADDR <= pcount;
			pcount := pcount + '1';

			IF (prog = '1') THEN
			-- In programming phase
				WRITE <= '1';
				DATA <= PREIN;
				IF (PREIN = "1111111111111111") THEN
```

```
                        prog   := '0';
                        pcount := "00000000000000000";
                        DATA <= "ZZZZZZZZZZZZZZZZ";
                    END IF;
                ELSE
                -- In processor phase
                    WRITE <= '0';
                    -- Fetch
                    -- Decode
                    opcode := DATA(15 downto 12);
                    opdata := DATA(11 downto 0);
                    -- Execute
                    CASE opcode IS
                        WHEN "0000" => holder := holder;
                        -- ADD:  Add HOLDER with OPDATA
                        WHEN "0001" => holder := holder + opdata;
                        -- SUB:  Subtract OPDATA from HOLDER
                        WHEN "0010" => holder := holder - opdata;
                        -- INC:  Increment HOLDER
                        WHEN "0011" => holder :=
                                            holder + "000000000001";
                        -- DEC:  Decrement HOLDER
                        WHEN "0100" => holder :=
                                            holder - "000000000001";
                        -- NOOP: Do nothing
                        WHEN "0101" => holder := holder;
                        -- LOAD: Set HOLDER equal to OPDATA
                        WHEN "0110" => holder := opdata;
                        -- Out:  Write HOLDER to LEDPIO
                        WHEN "0111" => LEDPIO <= "0000" & holder;
                        -- GoTo: PCOUNT <= OPDATA
                        WHEN "1000" =>
                            pcount := "000000" & opdata;
                            ADDR   <= pcount;
                        -- EOP:  LED  <= FF;
                        WHEN "1001" => rst := '1';
                        -- NOOP if unrecognized OP code
                        WHEN OTHERS => holder := holder;
                    END CASE;
                END IF;
            END IF;
        END PROCESS;
END RISC_Processor_wMem_architecture;
```