Design and Characterization of a Hardware Encryption Management Unit for Secure Computing Platforms

Anthony J. Mahar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Mark T. Jones
Cameron D. Patterson

June 3, 2005
Blacksburg, Virginia

Keywords: secure computing, encryption, cryptography, secure processors

Design and Characterization of a Hardware Encryption Management Unit for Secure Computing Platforms

Anthony J. Mahar

(ABSTRACT)

Software protection is increasingly necessary for a number of applications, ranging from commercial systems and digital content distributors, to military systems exposed in the field of operations. As computing devices become more pervasive, and software more complex, insufficiencies with current software protection mechanisms have arisen. Software–only and data–only protection systems have resulted in broken systems that are vulnerable to loss of software confidentiality and integrity.

A growing number of researchers have suggested that hardware encryption mechanisms be employed to enforce software protection. Although there are several competing architectures, few offer the necessary protection while remaining compatible with modern computing systems and models. The Virginia Tech Secure Software Platform is the first architecture to achieve both increased protection and usability.

This thesis presents the design and implementation of a fast, flexible Encryption Management Unit (EMU) for Virginia Tech Secure Software and compatible platforms. The design is capable of providing decryption of program instructions residing in page–sized sections of memory, without modification to the core processor. The effect of the EMU is modeled with varying application types and system loads. Lastly, a benchmark designed to measure actual performance was created to measure the actual performance of the EMU and validate the models.

Acknowledgments

I would like to thank my committee chair, adviser and instructor Dr. Peter Athanas. I am extremely grateful for the mutual trust and respect that has developed during my time at Virginia Tech. His patience and continual review of this work has been instrumental in the development of this thesis.

I would like to thank Dr. Mark Jones as an adviser and committee member. His advice on academia and matters of life have proved extremely enlightening.

I would like to thank Dr. Cameron Patterson for serving as a committee member. I have learned much from Dr. Patterson on personal happiness in and outside of academia.

All three of these individuals have earned so much respect and admiration for the support and confidence they have offered over the years. Each of their own personal career paths and successes has provided much inspiration and demonstrates such exciting opportunities. It is an honor and pleasure to have worked with each of them.

Additionally, I would like to thank the members of the Virginia Tech Secure Software Project research group: Justin Stroud, Benjamin Muzal, and especially Joshua Edmison. It is a rare opportunity to work in a team with such incredible cohesion, individual responsibility, and communication.

I would like to thank Luna Innovations, especially Jonathan Graf and Barry Polakowski, for providing the joint research funding and objectives for the Virginia Tech Secure Software Project.

I would like to thank the old crew from the Virginia Tech Configurable Computing Lab, who have made my time at Virginia Tech one of the most enjoyable and memorable. These true friends include Stephen Craven, David Lehn, Neil Steiner, Alex Poetter, and Jesse Hunter.

I would like to thank Barry Mapen for being a wonderful friend and mentor, Dr. Ian Greenshields for encouraging graduate studies and providing superb insight into academia, and Roger Leege for instilling an enjoyment of engineering and research disciplines.

Lastly, and most importantly, I would like to thank my parents, Christine and Joseph Mahar. Their unwavering support, love, and interest in my academic career has provided a continued source of strength and inspiration that I am eternally grateful for. I certainly would not have been successful without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are a number of key motivations for conducting this line of research. Reasons include opportunities to advance the state of the art in the field, to conduct research with a successful platform, and to solve deficiencies within the field. These motivations directly support the contributions presented in this work.

1.1 Motivations

The primary reason for conducting this research is to contribute to the Virginia Tech Configurable Computing Lab's Secure Software Project [1]. Software protection is currently an extremely active research area with proposed solutions coming from leaders in the computer industry and academic institutions. This research involves the development of methods for maintaining the confidentiality and integrity of software during distribution, in local storage and memories, and during execution.

While there exist other proposed architectures in this research, however most either lack complete protection protection or require mechanisms that do not integrate well in modern platforms. These systems often include required trust in parts of software, or requiring

modifications, sometimes fundamental, to the operating system and processor. The inherited trust model used by many software protection systems [2] has already been demonstrated [3] as readily exploitable. When a trusted module is compromised it can become a staging point for further attacks.

Systems relying less on an inherited trust models [4] [5] typically require fundamental changes to the way software operates and how the processor and operating system interact with the software. This includes running software inside individual virtual machines, addition of secure instructions and registers, or lack of shared libraries, shared memory, and standard inter–process communication mechanisms. These systems may provide software protection, however they cannot easily incorporate into modern computer systems due to the fundamental changes and incompatibilities with current software models.

The Virginia Tech Secure Software architecture is the first platform to provide increased software protection while maintaining standard development and computing models. This architecture avoids using special instructions or special operations to interact with shared memory, libraries, or other applications. This permits the developer to maintain their compiler tool flow. To increase protection, selected groups of instructions in the executable are tagged and encrypted, and then distributed to systems that will run the application. As protected instructions remain encrypted, the software can be distributed over standard and potentially insecure channels. When the application is executed on a Secure Software architecture, partial credentials stored in the executable are merged with a set of user supplied credentials securely on the chip, producing a set of one or more decryption keys.

The Secure Software architecture enforces an execute–only policy on protected instructions. This is achieved in several ways, depending on the architecture configuration. In some cases, only the instruction–side bus interface has a decryption unit, in other instances, when there is also a data–side decryption unit, the system can forces the use of different keys for each side. Without a corresponding data–side decryption unit and key set, the CPU cannot directly read or write protected instructions. This secure support is achieved without

modification to the processor core.

With the VT Secure Software architecture, trust is not required of the operating system or any other software module. Although the operating system (OS) associates groups of instructions, in the form of memory pages, with a key, it is only aware of a pointer to a key in a hardware key table, and can never access the keys under any circumstance. To establish this relationship between groups of instructions and key pointers, the system extends the memory page handling mechanisms already used in modern operating systems.

The second motivation for this work stems from additional deficiencies in the secure computing field of research. From the existing set of secure architectures, only a few attempt to model the performance effect of their protection mechanisms. When modeling is provided, there is not sufficient verification of the model, which is often due to lacking an actual implementation to verify with. Application–based benchmarks are performed in architectural simulators using assumed latencies of secure extensions. These simulations, however, do not properly reflect true latencies that are often discovered only during implementation. Additionally, these simulated benchmarks fail to represent certain non–deterministic processor behavior, or the variability found within modern multi–tasking operating systems.

The third motivation results from the necessity to properly benchmark and measure latencies associated with the instruction–side interface of processors. There are many benchmarks available, including benchmarks for parallel computing, operating system characteristics, disk I/O, memory bandwidth, memory latencies, and more. Memory latency benchmarks are the most important benchmark when characterizing software protection platforms. The software protection extensions can add delay when fetching instructions from memory, due to the cryptographic process. In the present implementation of the Secure Software Platform, protections are afforded primarily to instructions, therefore a benchmark is required to correctly measure the latency of memory fetch on the instruction–side bus. As an instruction–side benchmark does not exist either commercially or in the public domain, one such benchmark had to be developed.

1.2 Contributions

Within the Virginia Tech Secure Software Project there are several core components of research, each contributing to the increase of protected software execution. These research components include hardware encryption management, secure key management, operating system extensions, formal methods of security validation, and integration with parallel computing environments. The primary contribution of this work is the design and implementation of an Encryption Management Unit, or *EMU*, to support the execution of protected instructions.

During the development of the EMU, considerable effort was given to design, employing several important software and hardware engineering techniques. The goals in using these techniques were to focus on the overall design problem, while avoiding implementation details. A cohesive set of modules was created to permit reuse of the EMU and its internal modules, and to remain adaptable to continual changes in EMU requirements. This is necessary as the overall Secure Software architecture improves and develops. When the high–level design was complete, implementation of the EMU required that latency be reasonably minimized given the constraints on area and time of the prototype.

The second contribution of this work is modeling the behavior of the EMU within a standard multi–tasking platform. Mathematical models were derived to determine the effect on performance of the EMU under various decryption algorithms and modes. These models also account for variations in application type and operating system loads in a multi–tasking system.

To verify the accuracy of these models, a synthetic benchmark, called *iBench*, was created for measuring instruction–side bus memory latencies was researched, created, and verified for accuracy. With the validity of *iBench* established, the benchmark was used to directly evaluate the VT Secure Software platform under several system profiles. The results were also used in verification of the mathematical models previously derived.

1.3 Implementation Platform

The hardware platform used for implementation was a Xilinx Virtex–II Pro [6] Field Programmable Gate Array (FPGA). The Virtex–II Pro contains an IBM PowerPC 405 embedded processor [7] surrounded by reconfigurable logic. The FPGA is mounted on a Xilinx ML–310 [8] embedded development board, which contains a representative set of memory types and peripherals found on modern computers.

These features provide a comprehensive hardware/software platform running with a complete Linux operating system. The reconfigurable fabric of the FPGA allows the instantiation of the EMU and other components on the same physical chip of the the processor. The Xilinx Embedded Development Kit [9] (EDK) is a design tool that provides high level management of hardware component cores instantiated in the FPGA. With this tool, cores are easily added, removed, parametrized, and attached to standard bus structures within the system, creating a configurable system–on–chip (SoC). These features allow the rapid integration of VT Secure Software Project hardware extensions into the embedded SoC design.

1.4 Thesis Organization

This thesis is organized in the following manner. Chapter 2 presents background information on the secure software field of research, the Virginia Tech Secure Software Project, and related work. Chapter 3 details the design of the Encryption Management Unit and its interaction within the system. Chapter 4 provides mathematical models to determine the effect of the EMU with respect to multi–tasking environments. Chapter 5 describes *iBench*, the benchmark created to measure latency of the EMU and memory subsystem. Chapter 6 provides performance results of the EMU under several platform configurations. Chapter 7 discusses future directions of this work. Finally, Chapter 8 concludes this thesis with a summary of the contributions of this work.

Chapter 2

Background

This section provides a background required to provide proper context for the work presented in this thesis. Important areas include the history of secure computing, backgrounds on different types of software threats, systems with functional similarities to the EMU, and methods for benchmarking secure software systems.

2.1 Secure Computing

Security has maintained an important role in software since the beginning of digital computing. Computers are commonly used to perform operations on sensitive data where the data and the computational instructions must be protected. Initially, digital computing devices operating on sensitive information were physically isolated, and protection required physical security. These devices were operated by personnel with appropriate authorization, resulting in an implicit line of trust between secure software and authorized user. However, as computing devices became more pervasive and interconnected, the need for secure computing in insecure locations and without trusted users became necessary.

Initial attempts [10] into software protection produced processors augmented with various

protection mechanisms. Features included specialized instructions for cryptographic acceleration, encryption of instructions and data in external memory, physical package anti–tamper mechanisms, and external memory address obfuscation. At the time, it was thought that these processors could run a single application securely, providing encryption and obfuscation for any instruction or data placed into external memory. Only recently has other research shown [11] these protections were inadequate.

The fundamental task of a processor is to execute instructions. With the secure crypto–processor, instructions are decrypted as they are retrieved from memory and then executed. Even if instructions reside as encrypted in external memory, it is possible to randomly alter the contents of memory and observe the system reaction to executing the decrypted random value. Starting with the processor's boot instruction, an attacker can guess possibilities for encrypted instructions and place them at that initial boot location. If there is an observable effect, such as access to memory mapped I/O, the guesses can be refined until a complete instruction with a desired effect is determined. Iterating for several instructions allow a small encrypted program to be constructed by an attacker, which can then be used as a staging platform to access protected memory from within the processor.

A typical consumer of secure processors at the time, and one that highlights a common scenario in the era, were financial institutions. Remote computing nodes such as bank terminals and automated teller machines were located outside physically protected domains, and were potentially exposed to attacks by adversaries in possession of the devices. These machines did not need much processing power, but instead required extreme integrity and confidentiality of software instructions and data. As software distribution and installation was considered secure, and the processor ran a single dedicated program, attacks were limited to software in memory and memory buses. The intent of adversaries included modification of program data to alter transactions, or snooping a processor's buses for account theft. Software protection from these threats required software confidentiality and integrity of program data and instructions when in memory outside the processor core.

Over the past decade there has been a growing push towards enabling secure computing in modern multi–tasking platforms. There are many reasons for this, such as the growth of software piracy [12] [13], protection of digital media in new content distribution markets, protection of intellectual property of proprietary algorithms, and more. Simply upgrading processor architectures and cryptographic macros from the first generation of secure processors does not provide a complete solution required of securing modern computing models. Encrypting all external memory with the same encryption key still leaves the system vulnerable to internal threats from other malicious software or faulty operating systems. Developers could not rely only on a specific set of trusted software to be running. Instead, there were many applications running concurrently, and many without user or developer trust.

Modern computing platforms are capable of running multiple applications concurrently. This is possible through features such as virtual memory and supervisory operating systems. Security had to be handled in a manner that would allow one application to be protected from external threats in addition to other software running on the processor, operating system or otherwise. This required that secure extensions to a multi–tasking platform must provide security mechanisms to isolate one secure application from any other application or operating system not sharing the same set of access rights.

One approach to solving this problem was through various forms of obfuscation. Obfuscation relies on the ability to confuse an adversary attempting to gather information about software operations. Techniques included code obfuscation, where program instruction execution flows and data formats are mangled, and bus protocols that would attempt to hide the execution flow or data access patterns of external memory. While both were easy to implement and did not impact performance much, these approaches were either rejected [14] on provable of insecurity, or quickly broken [3].

A number of secure multi–tasking architectures using hardware cryptographic protections were developed from the realization that these previous mechanisms were insufficient. Each one differed in their level of use in trusted software components, their method of hardware

protection, and what protections were offered. The protections for software reduced into two primary classifications: software confidentiality and software integrity. Some architectures also define data confidentiality and integrity as part of software protection. [15] further discusses the area of research, the different secure mechanisms, and their security implications.

Although these architectures provide software protection through various mechanisms, another important metric for success is the difficulty of integration into actual hardware–software architectures. Some architectures, such as [16] and [17], require modification of the core processing unit with additional instructions and special registers. While the goals of these architectures offer state of the art capabilities, they are difficult to prototype and to implement; commercial off the shelf (COTS) processor cores are not easily be used with these systems. Furthermore, mechanisms for common abilities such as inter–process communication and shared memory are not sufficiently addressed.

At the time of this work, only two architectures are known to have demonstrated successful implementation. These include the Virginia Tech Configurable Computing Laboratory Secure Software architecture [1] and the Trusted Computing Group (TCG) Trusted Platform Module (TPM) specification [2]. The TCG architecture has already seen wide–spread adoption, mostly through backing from leaders in the operating system and processor markets. Unfortunately, it relies on the inherited trust model and is left vulnerable to attacks on external memory.

2.2 Software Threats

There are two primary threats to software applications that protective architectures must address. Although *software protection* is a term with many meanings, it is considered for the purposes of this work, and the Virginia Tech Secure Software Project, to offer confidentiality and integrity of an application's instructions from internal software and external physical threats. While data protection is a related area of research, it does not necessarily improve

the protection of software instructions.

Software confidentiality offers protection from threats such as unauthorized or unlicensed copying of programs, including piracy and reverse engineering of algorithms. Providing software confidentiality has an impact on the protection of intellectual property, both in terms of copying and reverse engineering. Without confidentiality, a program running outside a developer's secure domain can be considered disclosure of private information to the public, competitors, and malicious users.

Software integrity provides protection from threats that modify program instructions. This ensures secure software is running the way the developers programmed it to. Furthermore, the user can maintain trust that the program has not been altered beyond what the developer created. Maintaining this integrity assists in the protection from software viruses, worms, or malicious uses that rely on code modification to mount attacks.

2.3 Virginia Tech Secure Software Architecture

Members of the Virginia Tech Configurable Computing Laboratory have been conducting research into secure architectures that provide software integrity and confidentiality in a modern computing system. Developed under the project title Secure Software, the goal is to provide these software protections in hardware, without placing trust in the operating system or any other software component. To ease integration, the security features extend and compliment development flows, operating systems, and processor architectures without fundamentally altering normal operation.

In the current phase of research, the VT Secure Software architectures provide software protection through increased confidentiality and integrity of program instructions. The single secure processor architecture is currently defined, however the project is extensible to multiprocessing and parallel processing environments. The granularity of this protection is provided to individual memory pages of an application, and protected instructions remain

encrypted at all times, until they enter the processor's instruction–side interface. Instructions never reside in unencrypted form in external memory, secondary storage, or on the processor's local bus.

2.3.1 Paged Protection

Most modern computing platforms operate using the concept of virtual paged memory. Pages are fundamental units of memory that both the operating system and the hardware memory management unit (MMU) recognize. Virtual paged memory allows a program memory space to be virtually represented in the physical system memory space. Using entries in a MMU translation table, the translation look–aside buffer (TLB), memory accesses to a program's memory space are translated from its virtual page address to the physical page address. Pages are mapped in and out of memory as necessary, with different pages from different programs populating the physical memory space. This allows many programs to exist in memory at the same time, and allows programs larger than system memory to have a subset of its pages loaded at one time.

It was determined in the Secure Software Project that page–level encryption was the best method for providing software protection, from a hardware and software perspective for several important reasons. First, different encryption keys can be used for different program pages. This can help strengthen the use of encryption for a secure program and prevent replay attacks between separate pages. Second, different secure applications can share memory pages while using different keys for the non–shared memory. This allows programs to support secure shared memories while maintaining their own private spaces.

Lastly, the mechanisms required to handle page level encryption directly extend operations in place in the operating system and MMU. The OS already provides on–demand paging, where an entry in the TLB is created after an attempt to access a virtual address not already present. Operations that load the TLB on demand can also load the page–key mappings in the Secure Software architecture at the same time. If the page–key associations are

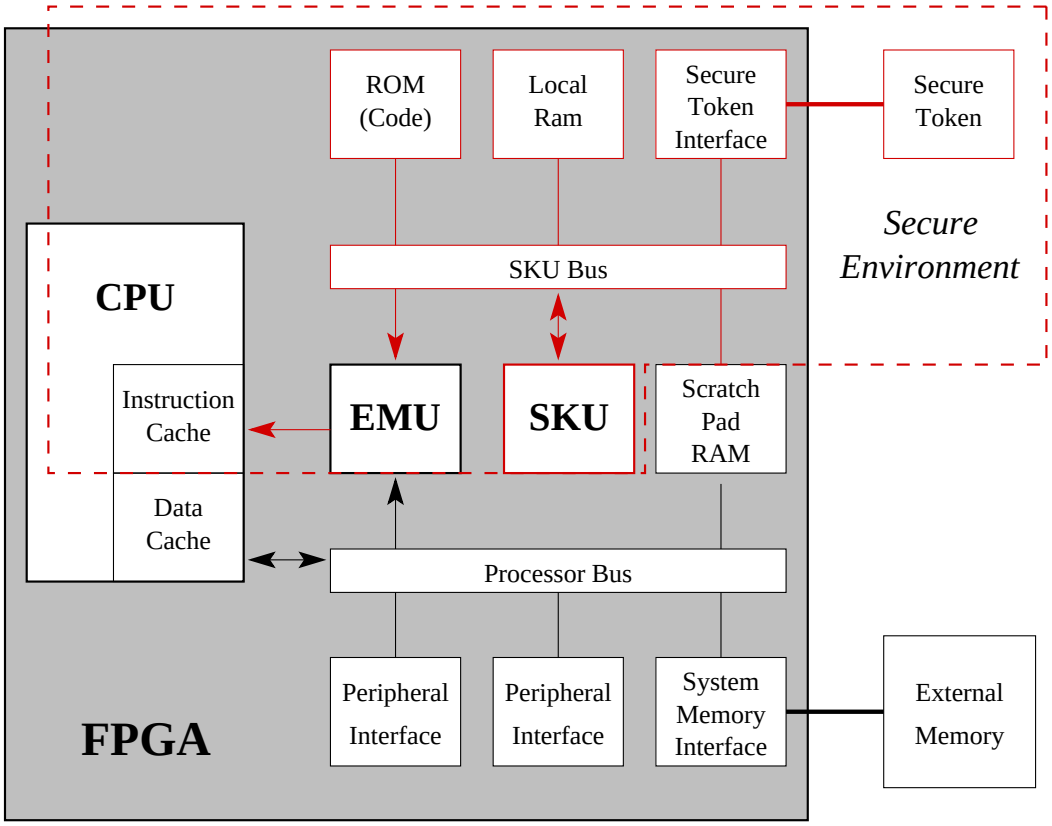

Figure 2.1: Single Processor Secure Software Architecture

established when TLB entries are added, then the instructions requested on the bus are guaranteed to be in the page–key association tables.

2.3.2 Primary Components

There are three primary components to the Virginia Tech Secure Software architecture. Figure 2.1 depicts the single secure processor architecture of the Virginia Tech Secure Software program, and the relationships between the primary components.

CPU/Operating Systems

The first component is the standard CPU with a memory management unit (MMU) to map virtual program address to the physical memory space. The operating system running on the CPU recognizes security tags and sections within each secure executable. The secure executable format use the same standard formats as a normal executable, but make use of the optional sections. Extra information in the secure executable indicates which executable instruction pages are encrypted, the cryptographic key identifiers for those pages, and partial security credentials for the keys. The operating system assigns page to key mappings at the same time it handles the virtual to physical address mappings in the translation look–aside buffer. Keeping these operations together ensures memory fetches on the bus will always reference up to date page encryption information.

Secure Key Management Unit

The second primary component of the Secure Software Project is the Secure Key Management Unit (SKU). This device is responsible for combining executable credentials and user credentials to create a proper cryptographic key. It is usually either a state machine or a small processor with its own private on–chip bus and memory. It receives information from the CPU containing partial executable credentials, a request for key generation, and where in a key table to place the generated key. An external secure token containing user credentials communicates with the SKU over a protected link. This provides the second half of the credentials used to create the keys.

The SKU resides on the same physical die as the processor but isolated from direct access by CPU. There is a set of intermediate buffers that the CPU and SKU use to communicate for key generation. This buffer isolates the SKU from the CPU. Similarly, the SKU is isolated from the external token. The secure token is accessed through a slave interface on the SKU bus. The SKU requests individual reads and writes from slave interface, which forwards

them to the external token. It is not possible for the external token to access the internal SKU bus.

Encryption Management Unit

The third primary component is the Encryption Management Unit (EMU). The EMU extends the MMU behavior of page address translation. Similar in operation to the TLB, the EMU provides hardware physical page–to–key and page–to–ancillary data mappings and look–ups. These mappings are possible through a series of look–up tables and table search units. The EMU uses this information with an internal decryption unit to selectively decrypt the instruction stream.

Many modern processors support separate data and instruction–side bus interfaces at some level. These processors, including the latest architectures from Intel [18] and IBM [19] feature separate data and instruction caches. These features allow the EMU to work with each stream, data and instruction, together or on an individual basis depending on the performance and security objectives.

The EMU is supports various decryption methods and cryptographic engines by inserting a desired cryptographic unit into the EMU. Cryptographic modes, such as direct block or counter modes, can also be chosen based on application. The keys stored in the EMU reflect the cryptographic key for a particular page, while ancillary data contains supporting information for certain cryptographic modes, such as page counters for counter mode encryption. Information supplied to the decryption component in the EMU indicates if a page is encrypted in addition to the encryption key and ancillary data if encrypted.

Although the EMU provides software protection, it must also protect other sensitive operations of the Secure Software architecture from the CPU. The EMU does not allow access to keys by the CPU. The CPU writes to a set of search tables in the EMU that point a page to a particular key slot in a key table. The CPU is only able to create mappings.

Likewise, the SKU can only write to the key table, but cannot access any part of the CPU–managed page search tables or to the CPU bus.

2.3.3 Lifetime of a Secure Application

This section provides the lifetime of a secure executable. It is useful in the illustration of high–level interactions with the primary VT Secure Software components. Lifetime is defined here as the period in which an application is conceived to the point in which it is terminated by a user.

1. Program is compiled using standard developer tool–chains.
2. Developer encrypts desired code pages of the executable. Additional sections in executable are added to specify which pages are encrypted, and to provided partial credentials needed by the SKU to generate keys.
3. Secure executable is distributed in its encrypted form using secure or insecure channels.
4. Secure executable is executed on a VT Secure Software compatible architecture (from local storage, networked storage, RAM disk, etc).
5. OS recognizes security flags and credentials in executable file, flags internal process structures with identifiers indicating which key index is associated with which page.
6. Standard on–demand paging is performed. Encrypted pages also follow this on–demand operation, loading security tables in the EMU when the TLB is loaded. Pages that are encrypted remain encrypted in external memory.
7. OS provides run–time key replacement strategy if key table is too small for all keys of all encrypted applications running.
8. All instruction transactions are compared against page status information in the EMU to determine encrypted status, and associated key and ancillary data.

9. If status is encrypted, EMU decryption unit decrypts the instruction transaction and returns result to the CPU. Otherwise, the instructions pass directly to the processor.

10. On application termination and during execution, pages are replaced under normal operations as other pages are paged into memory. New pages will replace the memory contents and security flags in the EMU with ones that reflect the status of the new page.

2.3.4 Protections

The primary goal of the Virginia Secure Software Project is to provide increased software protection. The software protections defined include instruction confidentiality and integrity. These protections are offered through the architecture's ability to keep instructions in encrypted form throughout the distribution, storage, and fetch process until entering the CPU instruction–side bus interface. This section describes how these methods provide each security measure.

Software Confidentiality

Instruction encryption helps enforce software confidentiality. Instructions, which are loaded into memory, can only be executed after proper decryption on the instruction–side bus. Without the key, deciphering the encrypted version of a program's instructions is not feasible when proper encryption techniques are used. In the initial Secure Software architecture the platform does not contain a data–side bus decryption unit, and therefore cannot directly read or write properly encrypted instructions.

Furthermore, the processor does not have read or write access to the decryption unit or the key table. Therefore, the processor access an unencrypted instructions located on or off chip, or access the cryptographic keys and decipher the instructions in software. As the EMU is located on the instruction–side bus interface, and not the data–side, even a secure

application cannot read its own instructions in decrypted form.

Software Integrity

Software integrity is also supported through the use of instruction encryption. When properly using a modern symmetric block cryptographic routine, it is extremely difficult, and not computationally possible, to alter a single instruction with another encrypted instruction.

Block cryptographic modes operate on a group of data usually consisting of several instructions in one decrypt or encrypt operation. Robust cryptographic algorithms provide a large change in encryption or decryption output, regardless of how few bits change in the corresponding input. This prevents correlation between cryptographic pre–images and encrypted results. The initial Secure Software architecture uses direct block encryption modes.

An attacker trying to alter a subset of instructions within a cryptographic block, through internal software or external memory modification, could not cycle through a set of guesses and refine their attack to execute an instruction of their choosing. Any alteration to the encrypted instructions before execution, however small, would cause the corresponding block of instructions to decrypt into a pseudo–random set of bits likely consisting of invalid instructions. At the very least, execution flow would be completely altered and usually results in segment faults.

Furthermore, as instructions are decrypted on the instruction–side bus in this iteration, it is not possible for a secure application to meaningfully modify its own instructions, meaning if a technologically advanced adversary manages to break a single block of instructions, it does not open a back door into the protected instruction space of the program. The attack must be carried out on every other block of instructions as well.

2.4 Related Memory Modification Units

The design of the Encryption Management Unit has drawn inspiration from several systems and architectures. These units posses a set of protections and implementation strategies that are reflected in the EMU. The common characteristic of each system is to selectively apply a particular function on an instruction stream based on certain criteria. Related secure architectures to the Virginia Tech platform are described in [20].

2.4.1 IBM CodePack

The first system has a strong history with the processor used in the implementation of the Secure Software architecture. IBM CodePack [21] [22] is a hardware/software feature developed for reducing the size of code in memory for embedded applications. To reduce code size, application instructions are compressed after compilation. It uses architectural features of the IBM PowerPC processor [23] to support decryption. Notably, it uses a compression flag within the PowerPC memory management unit's (MMU) translation look–aside buffer (TLB). This flag is exported from the TLB to a dedicated line on the Processor Local Bus (PLB) during memory transactions.

A normal application running on the PowerPC consists of a series of 32–bit instructions, with each instruction containing an operator followed by one ore more operands. From one instruction to the next there is often little correlation between the 32–bit values. However, by splitting instructions into 16–bit halves, there is much higher correlation between the corresponding halves for each instruction. Separately compressing each half–instruction channel offers significant compression potential.

A substitution method based on frequency is used to replace repetitive 16–bit units with shorter bit alternatives. One drawback to this encoding method is that the least frequent units are represented by a set of bits longer than the original 16–bit format. These components do not occur often, and it is likely that the compressed instruction space will still

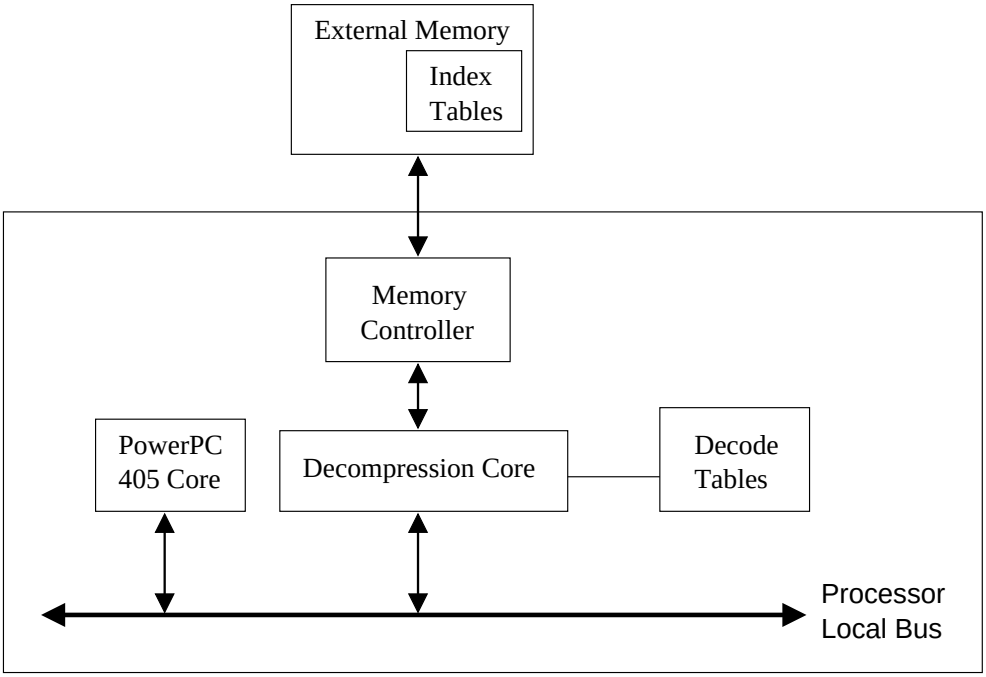

Figure 2.2: IBM CodePack Architecture

require less memory than the original uncompressed version.

With respect to address locations, the compressed version of the executable does not directly correspond to the original uncompressed version. Using variable length encoding produces a mismatch between the locations and sizes of instructions compared to the original set. To solve this problem, the IBM CodePack system uses an index table stored in external memory to find the base location of an instruction *group*. By representing large 128–byte groups of uncompressed instructions in the table, instead of maintaining each instruction address, the table size is significantly reduced. When a request for a compressed target arrives, the base location for the compressed block is looked up in the index table, the block is decompressed, and the target instruction is returned. The relation to the Encryption Management Unit is with the decompression strategy of the hardware portion of CodePack. The CodePack system retrieves information from tables in external memory, fetches the compressed memory block, and returns the requested decompressed values. In a similar fashion, the EMU retrieves information from tables in hardware in parallel with the encrypted

memory fetch, and returns the decrypted version. CodePack differs with the EMU in its location in the processor bus structure. As CodePack is not concerned with security, it is located on the external memory controller, while the EMU is placed directly between the processor instruction–side bus interface and the processor bus.

Although CodePack utilizes the MMU and the *compressed* PLB signal to determine the compression status for a page, the EMU does not. First, the content addressable memories and tables used in the EMU are extremely fast, using the *compressed* bus signal to indicate encryption would offer very little performance advantage. Secondly, the EMU must remain reasonably flexible to various architectures, and the *compressed* flag is PowerPC specific.

As both systems modify instruction streams, it is natural to attempt to draw conclusions about EMU performance based on CodePack performance. However, these two systems do not have similar performance characteristics. The CodePack system has quite a large buffer, 16 instructions, and this buffer is completely filled when a requested instruction is not in the buffer. Furthermore, as the index tables are stored in external memory, there is a high penalty for instruction fetch look–ups.

Surprisingly, even with the external memory table look–up and decompression routines, there are still applications that exhibit performance gains when using CodePack [24]. This is likely attributed to high instruction execution locality within a block of instructions, which results in the CodePack unit performing as a small pre–fetching cache. The EMU, on the other hand, retrieves and modifies only the requested instructions. As shown in Chapter 4, the EMU is strictly a delay element in the memory pipeline, and can never contribute to a performance increase.

2.4.2 XOM

The XOM architecture, or Execute Only Memory [25], provides similar instruction protections as the Secure Software Encryption Management Unit. It attempts to ensure instruc-

tions can only be executed, never read or modified, through decryption of instructions and data as they are passed through the cache system and into the processor. XOM uses application level cryptographic keys to isolate applications from each other. Each key creates what XOM refers to as cryptographic *compartments*. Unlike the EMU, these keys are stored in encrypted form in external memory and are verified against a private hash whenever a key is retrieved. Similar to the EMU, the cryptographic unit of the XOM system uses a symmetric cryptographic algorithm for decryption of instruction blocks.

Although both the Secure Software EMU and XOM provide similar execute–only instruction capabilities, their design and use within their respective systems are very different. First, XOM does not support page–level key associations. It takes the approach of application–level key associations. To identify what key should be used, or even if the application is encrypted, special instructions and registers were added to manually start and stop the XOM unit. As keys are stored in external memory, a further set of registers and instructions were added handle where in external memory the application key resides.

The Secure Software EMU uses a page–level encryption scheme that allows the hardware to efficiently mirror the TLB in establishing page–key relationships. This eliminates the manual start/stop triggering required by XOM, and the costly key look–up and decryption from external memory. Another problem involved with XOM start/stop triggering is with context switches and interrupts. When these occur, the XOM unit is still enabled and any instruction fetch of the interrupt handler will be incorrectly decrypted using the previous compartment key.

The XOM solution is to rewrite the interrupt vectors into a private XOM memory, and install a wrapper for interrupts. These wrappers disable the XOM unit before proceeding passing on to the true handler. This problem is avoided in the Secure Software system, as the EMU is aware of the encrypted status of a page, and if the interrupts are not flagged as encrypted the the cryptographic unit is bypassed.

2.5 Benchmarking

Software protection provides security for instructions, which likely requires alteration to the performance of the instruction–side bus of the processor. Additional latency from security mechanisms play a critical role in the overall performance of the secure application. To measure this effect, a benchmark capable of testing the instruction fetch performance is required.

In the realm of benchmarks there are two basic directions: synthetic benchmarks and application benchmarks. Synthetic benchmarks typically focus on testing particular features of a system, such as memory bandwidth, I/O latency, and more. They consist of applications designed specifically for performing the tests. Application benchmarks are entire programs, or representative code of programs, run on the test platform. Between different platform configurations, they illustrate how a particular application will perform. Furthermore, as parameters change, it is possible to the effect of that change, and how well targeted applications run with the new settings.

There are many effective application benchmarks available for an array of platforms. Many publications in the area of secure software favor the SPEC benchmark to demonstrate the effect their protection mechanisms have on performance. The SPEC CPU [26] suite of benchmarks consists of floating–point and integer based benchmarks that perform various functionality, such as chemistry and physics modeling, C/C++ and hardware description language (HDL) compilers, text interpreters, compression routines, and more. However, the problem with the SPEC benchmark suite is the selection of benchmarks do not effectively stress the instruction–side bus interface. Current research [27] reveals that the SPEC CPU suite is insufficient for analyzing systems that effect memory latency of instruction fetches.

Although an alternative set of application benchmarks have been recommended to more effectively stress the instruction–side bus, it was decided that if this work was to improve upon the existing benchmark standards a synthetic benchmark that could directly measure

instruction–side latency was required.

In the area of synthetic benchmarks there are two primary types: benchmarks that stress particular platform characteristics, and benchmarks that calculate integer or floating–point operations per second (OPS and FLOPS). NAS NPB [28] and Splash 2 [29] offer good characterization for serial and parallel computing systems when determining operations per second. The problem with OPS and FLOPS based synthetic benchmarks is they usually attempt to store as many instructions in cache to maximize data calculation throughput. This is precisely the opposite effect needed to test the instruction–side bus interface.

LMBench is suite [30] [31] of synthetic benchmarks designed to test various aspects of the Linux operating system and the underlying hardware. It provides validated measurements of content switching times, I/O latency and bandwidth, network latency and bandwidth, file system performance, and more. Of particular interest is an individual benchmark within the suite to measure memory latency. The *lat_mem_rd* module tests the various latencies of the memory subsystem from the data–side bus interface.

Essentially, a linked list is traversed in memory using a fixed length stride. The linked list is circular to allow for wrap around. A single instruction is used to jump to the next list element, allowing execution time across many fetches to be dominated by memory latencies. Generally, time for the data fetch instructions are divided by the number of fetches to provide the average latency for traversal of a given area of memory.

Initially, the size of memory traversed is relatively small, and is able to fit entirely inside data cache. The size of memory used increases for each iteration until the maximum size has been reached, which is usually many times larger than either the cache or the size of memory accessible by entries in the processor TLB. When the entire set of data for the linked list is small enough to fit inside data cache, the execution time is dominated by the latency of a cache fetch. As the size of the linked list grows, the time stalled on data is dominated by the next subsystem in the memory hierarchy.

Depending on the architecture, when the local cache hierarchy is exhausted, the next

subsystem is external memory or external cache. The majority of time spent on data cache misses is due to external memory fetches. As the area for the linked list grows even further, a point comes where the page tables are insufficient to represent the memory area for the benchmark. The average latency of the TLB miss is dependent on how many additional data fetches occur within that page. For instance, a stride with the length of an entire page will show higher latency for TLB misses than a stride with multiple hits per page. One fetch per page has a higher average penalty than multiple fetches per page. When plotted, each memory subsystem is distinguished by a unique plateau of latency with respect to memory size.

The LMBench results for data memory subsystem latency can be seen in Figure 5.1. Validation of LMBench latency test accuracy is included in [30]. Unfortunately, as the LMBench latency benchmark uses the data–side bus interface, it does not stress the instruction–side at all. *iBench*, described later, uses the instruction–side bus interface to obtain results, not from data fetches, allowing effective measurement of the EMU.

Chapter 3

Design

This chapter presents the Encryption Management Unit from multiple levels of abstraction. Requirements are provided to assess the necessary set of functionality that the EMU must achieve. These requirements also constrain the problem scope that the EMU must solve in its design. The specification section presents the primary modules, constraints on the functional implementation, and the protocols used for interaction between the high level modules. In the last section, detailed information will be provided on the design of each of the modules.

3.1 Encryption Management Unit Requirements

This section defines the requirements of the Encryption Management Unit within the Virginia Tech Secure Software Project for instruction protection.

3.1.1 Scope

The purpose of the Encryption Management Unit is to provide selective decryption of software instructions requested by the Central Processing Unit. Selection for decryption is

determined on a memory page basis, where instruction within an encrypted page will be decrypted, or bypassed if not within an encrypted page. The design and implementation of this unit satisfies the hardware extensions required by the Virginia Tech SecSoft Architecture for software protection.

3.1.2 Functional Integration

The Encryption Management Unit is intended to augment a typical Central Processing Unit without modification. To facilitate instruction protection, the EMU is should be installed on the same physical die as the CPU or within a secure multi–chip module, such that it is located directly between the CPU instruction–side bus interface and the processor bus. In addition to CPU and memory bus interfaces for decrypting the instruction stream, two additional bus interfaces are necessary to allow configuration of the EMU from the CPU and SKU. Although the design provides flexibility with many processor architectures, this prototype will extend the IBM PowerPC 405 embedded processor.

3.1.3 Operational Concept

The EMU is placed in the path of the instruction stream flowing into the CPU. The EMU observes memory transactions requested by the CPU and can actively modify the transaction request. Using the transaction address, the EMU performs a search of its internal tables to determine if the transaction is part of an encrypted page. If the page is determined to not be encrypted, the instructions are passed directly to the CPU without modification. If it is determined that the page is encrypted, then the encrypted instructions and additional information, such as the decryption key and ancillary data, are passed to a decryption mechanism, where it is decrypted and forwarded to the CPU.

Two configuration bus interfaces are necessary to configure the EMU and load encryption information into the internal tables. One is necessary for communication with the CPU, for

loading tables with page information and ancillary data. Another interface is necessary for communication with the SKU, for loading keys only the SKU has access to.

3.1.4 Functional Requirements

The following is a list of functional requirements of the EMU. These requirements define the goals of the EMU and constrain the design to the necessary functionality.

1. *Decrypt Instruction Stream* The system must decrypt the instruction stream when criteria for decryption is met.

2. *Bypass Instruction Stream Decryption* The system must not decrypt the instruction stream when criteria for decryption is not met.

3. *SKU subsystem must be isolated* The system must support isolation and protection of the SKU. Any interface of the EMU that attaches to any part part of the SKU subsystem must not expose or modify any form of internal information, including bus transactions and SKU machine state. The EMU must only recognize information explicitly sent by the SKU, such as in requirement 7.

4. *Keys must be isolated* The keys generated by the SKU and stored in the EMU key table must be completely isolated from any direct access or inference by the CPU.

5. *Searchable Tables* The system must be able to search tables for encryption criteria and the associated page key and ancillary data. If status of the page is not encrypted, then the returned page key and ancillary data return an unknown value.

6. *Load Page Tables* The CPU must be able to load the following information into specific locations in the page search tables:

    - Page to be searched
    - Key Slot pointer associated with page

- Ancillary Data associated with page

7. *Load Key Tables* The SKU must be able to supply keys into specific slots in the key table.

8. *Clock Domain Synchronization* The system must provide cross-clock domain synchronization between the primary clock domains. Primary clock domains are shown in Figure 3.2

3.2 Encryption Management Unit Specification

This section defines the high level specification of modules and their interaction within the system. Each module represents a cohesive functional abstraction within the system. Well defined interfaces provide proper coupling between the modules, and allow modules to be modified or replaced without effecting the remaining system.

3.2.1 Functional Constraints

Listed are the constraints in implementing the EMU with in the Virginia Tech SecSoft platform, specifically with the Xilinx Virtex–II Pro FPGA. Although these constraints are necessary for this particular implementation, the system remains reasonably flexible to changing bus widths, page sizes, key widths, and more.

- The processor for augmentation is the IBM PowerPC 405 embedded processor.
- Memory pages are 4 kilobytes wide.
- The instruction–side PLB bus of the CPU operates using three distinct transaction modes:
  - Single beat transfer, of 64–bits per beat

  - Four–word line transfer, transferred in two 64–bit beats
  - Eight–word line transfer, transferred in four 64–bit beats

- The page tables contain 64 entries.
- Decryption/Encryption block widths are 128–bits wide.
- The page containing the address of the requested memory transaction will always be found in the search tables.
- The page containing the address of the requested memory transaction will only be in one slot in the search tables.
- The instruction–side bus of the CPU uses the IBM Processor Local Bus (PLB) standard.
- The instruction–side bus of the CPU is a read–only bus interface.
- The Page Table Control Interface communicates with the CPU over an On–Chip Peripheral Bus (OPB) bus.
- The Key Table Control Interface communicates with the SKU over an OPB bus.
- The instruction–side bus operates with a clock rate of 100 MHz.
- The CPU OPB bus operates with a clock rate of 100 MHz.
- The SKU OPB bus operates with a clock rate of 100 MHz.

3.2.2 Primary Functional Units

Based on the requirements in Section 3.1.4, there are a number of primary functional units that become distinguishable. These units handle specific functional tasks necessary for the complete operation of the EMU. These high level modules are depicted along with their interaction in Figure 3.1.

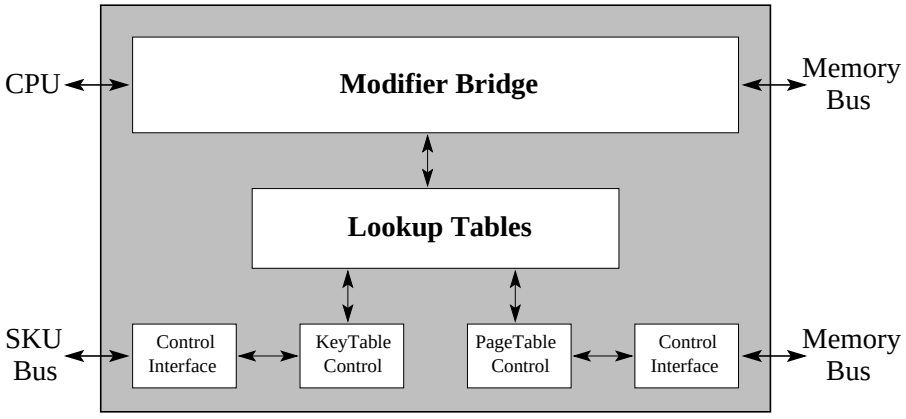

Figure 3.1: EMU Primary Functional Units

Bus Modifier / Bridge Unit

The Bus Modifier / Bridge unit provides by–passable active modification to instruction stream transactions. It is located directly between the CPU instruction–side bus interface and the processor bus. Criteria for selecting between decryption and bypass modes are queried from the Table Unit, using the page address of the requested transaction. The Bridge unit should allow future security extensions, such as instruction masking. The bridge satisfies requirements 1 and 2.

Although many decryption modes are supported by the EMU, two block encryption modes have been used for the initial configuration of the Secure Software architecture, specifically counter–mode and direct block mode. These are compatible with symmetric block cipher modes such as the Advanced Encryption Standard [32]. AES is well suited for use in FPGA technologies [33] [34] in addition to ASIC implementations [35].

The direct block mode decrypts instructions in 128–bit blocks. In this mode, the transaction type and size requested from the CPU is potentially modified before the request is forwarded to the memory arbiter. Using direct block mode ciphers, all 128–bits of an encrypted block are necessary to return any portion of the decrypted block. Because the PowerPC 405 CPU instruction–side interface can request single beat (64–bit), four–word

line (128–bit), and eight–word line (256–bit) transfer sizes, the single beat mode needs to be expanded to a complete four-word line transfer.

The flow for direct block decryption begins with a transaction request. When the transaction is acknowledged by the bus arbiter, the block decryption unit will retrieve both encrypted status and key information, and the instructions. Once both are retrieved, direct block decryption is performed on the instructions in one operation.

When decryption is complete the result is forwarded to the CPU. If the transaction is not flagged as encrypted, the instructions are forwarded to the CPU unmodified, and without any delay of decryption.

Counter–mode decryption does not use the cipher routine to directly decrypt fetched instructions as with block mode. Instead, a unique counter value associated with a block of data or instructions is *encrypted* using the cryptographic algorithm. This result is XORed against protected instructions to decrypt them. Conversely, as XOR is an invertible operation; these instructions are originally encrypted by XORing against the same encrypted counter value.

Proper cryptographic algorithms do not have any correlation between encrypted results of differing pre–images, even if the pre–images are related to each other through some arbitrary function. Therefore, the counter value itself does not have to be cryptographically strong, but merely different between each addressable block.

To minimize local memory each 128–bit instruction block the ancillary data table is used to provide the most significant bits of a counter for transactions within a page. The lower bits are a function of the transaction address within that page. With this method, each block within the same page and between pages of a protected program can maintain a unique counter value.

Similar to direct block encryption, the flow for counter–mode encryption begins with the transaction address acknowledge, which begins the table search and retrieval of encrypted

status, page key, and additionally ancillary data. If encrypted, two counter values and the associated key are fed into each of the two parallel cryptographic cores for encryption. The counter values are generated from a base page associated counter stored in ancillary data and the address of the transaction within the page.

Incoming instruction data is always placed into a FIFO along with the corresponding word address of the data beat within the full transaction. When encryption of the counters is complete, the unit will pop off the number of data beats in the transaction when they are available. For each data beat from the FIFO, the corresponding XOR segment of the complete map is XORed for decryption, and passed to the CPU. For unencrypted instructions the encryption function is bypassed and the XOR map is loaded with zeroes. As the unencrypted instructions pass from FIFO to CPU, the result of an XOR with zero does not alter the data.

Table Unit

The Table Unit combines all tables necessary functionality for storing and searching page–to–key and page–to–ancillary mappings into one module. This unit has an interface for searching the page mappings to keys and ancillary data. The page address supplied by the Bridge unit is used as the search criteria for retrieving the status. Once the page address is found in the tables, the encryption status is returned along with the associated key and ancillary data of the page. The returned key and ancillary information are invalid if the transaction is not encrypted. The search feature fulfills Requirement 5.

The table unit consists of three look–up tables and a content addressable memory (CAM). The content addressable memory reverses the typical memory operation of retrieving data from an address. Instead, the data is supplied to the CAM where the address it is stored in is returned. This is an efficient way to retrieve the key pointers and ancillary data locations based on the supplied page address.

An issue that must be addressed is the risk of the same page address stored multiple times in the CAM. This is a very unlikely event, especially with large memories and multiple processes running on modern systems. A hardware MMU typically uses a software defined process identification (PID) to define what application is requesting a virtual to physical page address translation. While this PID value does not create a security dependency, it can be useful to ensure a page look–up uses the correct entry, and not an entry from some previous execution. For prototyping, software checks in the operating system can avoid this situation, also without exposing any security risk.

To easily allow multiple pages to use the same key, the unit passes the returned index from the CAM to a key slot look–up table. This look–up table translates the selected page index into a key slot pointer. Pointers to key slot zero indicate the page is not encrypted. All other slot values indicate the page of transaction is encrypted.

The results of the key slot table look–up are used to select the key from the key table. Key values returned to the Bridge Unit for unencrypted pages should be ignored. With encryption status, key, and ancillary data determined, the set of information is returned to the Bridge Unit.

To account for flexibility in timing characteristics of the search and look–up operations the Table Unit uses a timing handshake signal. The unit responds to a *search start* signal from the Bridge unit with *encryption information ready* signals at a time determined by the Table Unit. The response timing signals represent the cycle that their look–up signals are valid.

To load the tables with appropriate information, two dedicated configuration interfaces exist satisfying requirements 6 and 7. The first interface, the page table configuration interface, is used to establish the page–key slot and page–ancillary mappings. The other interface, the key table configuration interface, is used to load the keys into the key slots. Both configuration interfaces are write–only. This separation supports requirements 4 and 3. Although the CPU and SKU access an interrelated set of tables, the CPU cannot write or read keys

Table 3.1: Page Table Control Unit Registers

Register Name	Internal Offset	Address	Access
Page Base Address	0x00	Base + 0x00	Read/Write
Key Slot Index	0x01	Base + 0x04	Read/Write
Page Index	0x02	Base + 0x08	Read/Write
Ancillary Data 0	0x04	Base + 0x10	Read/Write
Ancillary Data 1	0x05	Base + 0x14	Read/Write
Ancillary Data 2	0x06	Base + 0x18	Read/Write
Ancillary Data 3	0x07	Base + 0x1C	Read/Write

from the key table, and the SKU cannot write or read from the page mapping tables.

Page Table Control Unit

The primary purpose of the Page Table Control Unit is to interface the Table Unit with the CPU memory mapped I/O region. All of the registers in the Page Table Control Unit are used to collect all parts before writing to the Table Unit in one transaction. Writing to the page index register acts as a trigger which writes the contents of the page address, key slot, and ancillary data registers into the Table Unit at the location specified by the page index register. Transferring from this unit to the Table Unit can cross clock domains, and therefore must be capable of supporting transmission synchronization, as described in Section 3.2.3.

Table 3.1 lists the registers used by the Page Table Control Unit, the internal offset within the control unit, and each register's relative address within the CPU address space. The complete behavior and operation of the individual registers are listed in Appendix A.

Table 3.2: Key Table Control Registers

Register Name	Internal Offset	Address	Access
Key Slot	0x00	Base + 0x00	Read/Write
Key Word 0	0x04	Base + 0x10	Read/Write
Key Word 1	0x05	Base + 0x14	Read/Write
Key Word 2	0x06	Base + 0x18	Read/Write
Key Word 3	0x07	Base + 0x1C	Read/Write

Key Table Control Unit

The primary purpose of the Key Table Control Unit is to enable the loading of keys into specific slots within the Table Unit from the SKU. Beacause cryptographic keys are often large, and bus widths often relatively too narrow, individual words of the key are first written to registers in the Key Table Control Unit. Writing to the key slot register initiates a transaction of the contents of all key registers to the Table Unit at the slot indicated by the key slot register.

Table 3.2 lists the registers used by the Key Table Control Unit, the register offset within the control unit, and the relative address for the byte–addressable SKU memory bus. The complete behavior and operation of the registers are listed in Appendix B.

Control Interface Unit

The Control Interface Unit abstracts away the underlying bus protocol of the Page and Key Table Control units and a presents generic bus interface. This permits each control interface to be attached to various buses, provided a Control Interface Unit can be created to handle the conversion. Furthermore, because each control unit uses the same generic read/write bus interface, the same Control Interface Unit can be reused with either control unit if each

control unit connects to the same bus protocol. The generic bus protocol is specified in Section3.2.4.

In the prototype design, both the CPU and SKU buses use the IBM On–Chip Peripheral Bus [36] standard. A Control Interface Unit for converting between OPB and the generic bus interface was designed and used for both of the control units.

3.2.3 Asynchronous Protocols

Within the EMU there are several interfaces between modules that may cross different clock domains depending on implementation. Likewise, they must resolve the potential problems involving cross clock domain communication, including clock skew and asynchronous clock signals. The primary clock domains within the EMU are shown in Figure 3.2. A transfer protocol featuring full request/acknowledge handshaking and full synchronization described in [37] was used to overcome the problems associated with crossing clock domains. This features a push based protocol; a sender always initiates the transfer to the receiver.

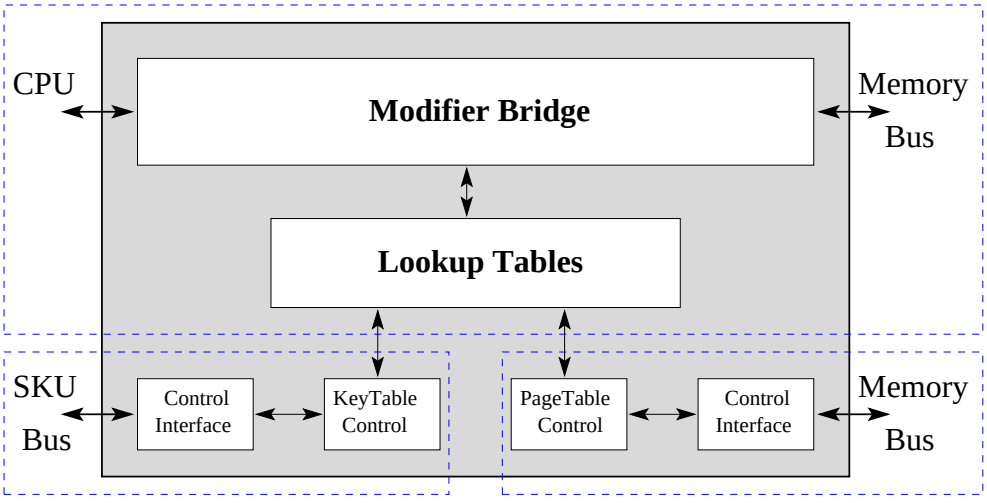

Figure 3.2: EMU Primary Clock Domains

The protocol begins by latching data to be transferred into registers in the sender unit. This data is kept stable in the registers throughout the entire transfer. The output of the

registers in the sender unit are fed directly into the inputs of registers of the receiver unit, without any passive or active logic in the path to improve switching characteristics. The receiver will only load this data into its own registers when synchronization is determined through the handshaking.

Full, two–stage synchronization [37] is used in the receive unit for the *request* signal. This prevents the introduction of metastable signals into the internal logic of the receiver. Similarly, the sender unit filters metastability of the *acknowledge* signal from the receiver through its own full synchronizer. Although two stages incur extra latency in the overall transaction, this greatly minimizes the potential of metastability once the second stage of the synchronizer is latched. The full synchronizer is depicted in Figure 3.3.

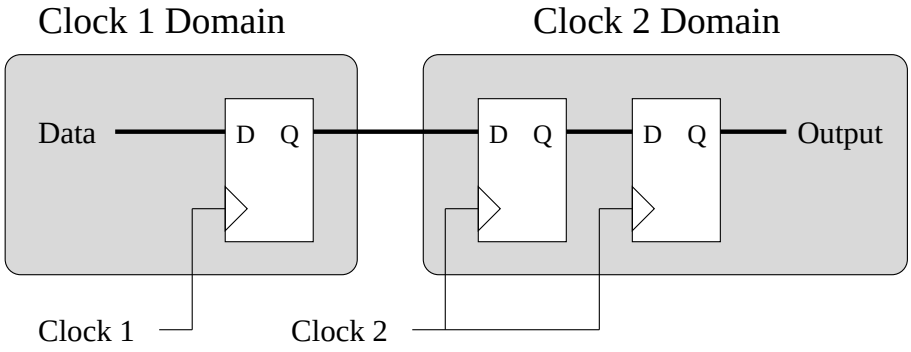

Figure 3.3: Full Synchronization Unit

After the sender registers are stable, the sender initiates the transaction by asserting its *request* signal. Once the receive unit detects the active *request* signal, it enables loading of its registers for one clock cycle. Following this load cycle, the receiver asserts the *acknowledge* signal to indicate receipt of the data.

The sending unit, after its initial request, waits for detection of the acknowledge response. When detected, it de–asserts its own *request* signal. The receiving unit detects the lowering of the *request* signal and de–asserts its *acknowledge* signal. For the receiver, the transaction is now complete. The sending unit finally concludes its transaction after it detects deassertion of the *acknowledge* signal. This full handshake process offers proper asynchronous com-

munication in addition to ensuring each side recognizing completion status of the opposite unite.

For demonstration purposes, Figure 3.4 illustrates the handshaking transfer protocol. Although the two units share the same clock in this depiction, the protocol remains valid for different clock domains as described above.

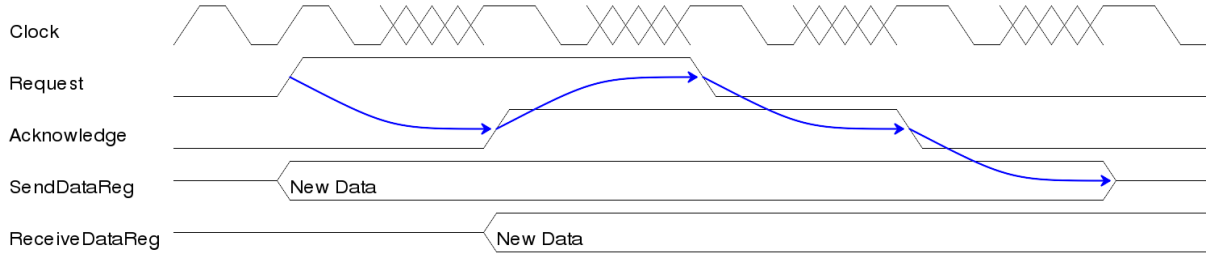

Figure 3.4: Base Asynchronous Protocol

3.2.4 Interface Protocols

This section defines the protocol specifications used for inter–module and external communications of the EMU.

Instruction Bus Protocol

The instruction bus protocol used in the prototype implementation uses the IBM 64–bit Processor Local Bus (PLB) standard [38]. This bus is capable of many read and write transaction types, including single beat, multi–word line transfers, master terminated and slave terminated burst transfer modes, and DMA transfer initiation. Through direct observation and empirical testing of the bus, it was found that the PLB master controller of the PowerPC 405 instruction–side bus is always a 64–bit read–only interface that exclusively uses three transaction modes. These modes include a single beat 64–bit transaction, a four–word (128–bit) line transfer, and an eight–word (256–bit) line transfer.

Reducing the number of supported transaction types reduces the set of necessary signals the bridge must control and recognize. The subset of required signals from the PLB specification are listed in Table 3.3. Full descriptions of this bus, transaction types, and timing are included in the 64–bit PLB Specification [38]. Although delay may be incurred when passing through the Bridge Unit, the association between the *read word address*, *data acknowledge*, and *data* signals must be maintained. Furthermore, the *busy* signal to the CPU must be extended for the additional delay of the EMU.

Table 3.3: Used PLB Instruction–Side Bus Signals

Signal Name	Bits	Description
M_ABus	32	Requested address
M_request	1	Request active
M_size	2	Transaction size
M_abort	1	Request abort
M_BE	8	Byte enable
PLB_MAddrAck	1	Address acknowledge
PLB_MBusy	1	Read busy
PLB_MRdDAck	1	Read data acknowledge
PLB_MRdDBus	64	Read data bus
PLB_MRdWdAddr	4	Read word address

Table Unit Search Protocol

The Table Unit receives both a "search start" signal, *iTableReq*, and a page address to query, *iPageAddress*. It responds a number of cycles later with an encryption status valid signal, *oTableSearchDone*. The encryption status, key, and ancillary data of the page look–up are only valid on the cycle in which *oTableSearchDone* is active.

Search operations returning encrypted and unencrypted status are shown in Figures 3.6 and 3.5. Although both figures show a typical look–up operation, it is possible that *oTableSearchDone* could be one ore more cycles after *iTableReq* is active. In the event where encryption status for a page is false, the corresponding *oPageKey* and *oPageAncillary* signals are invalid.

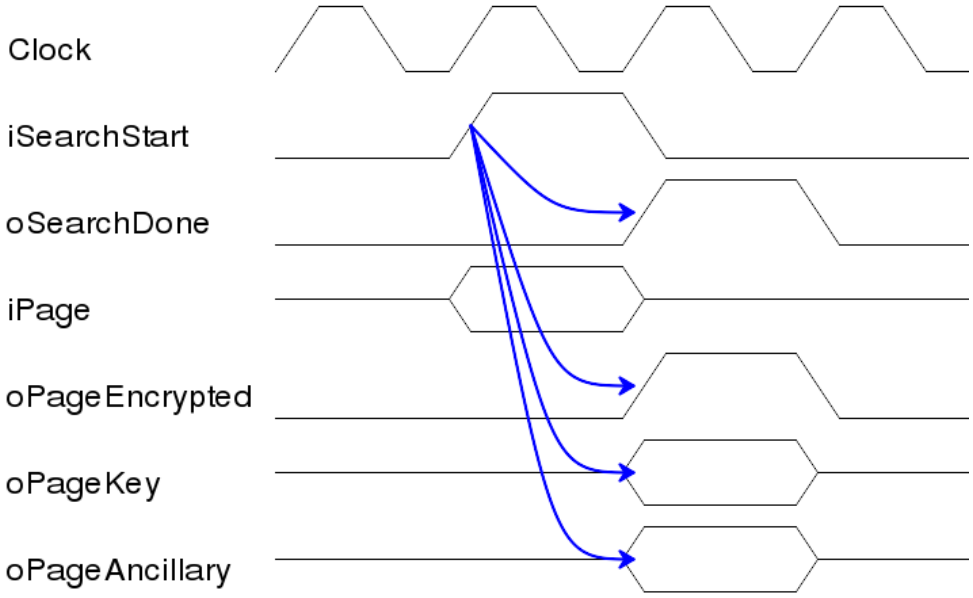

Figure 3.5: Protocol: Table Search, Encrypted Return

Page Table Control Protocol

The page table control protocol crosses clock domains from the Page Table Control Unit to the Table Unit. The asynchronous protocol described in Section 3.2.3 is used to provide the handshaking and data synchronization. The data set transferred during the handshake is shown in Table 3.4.

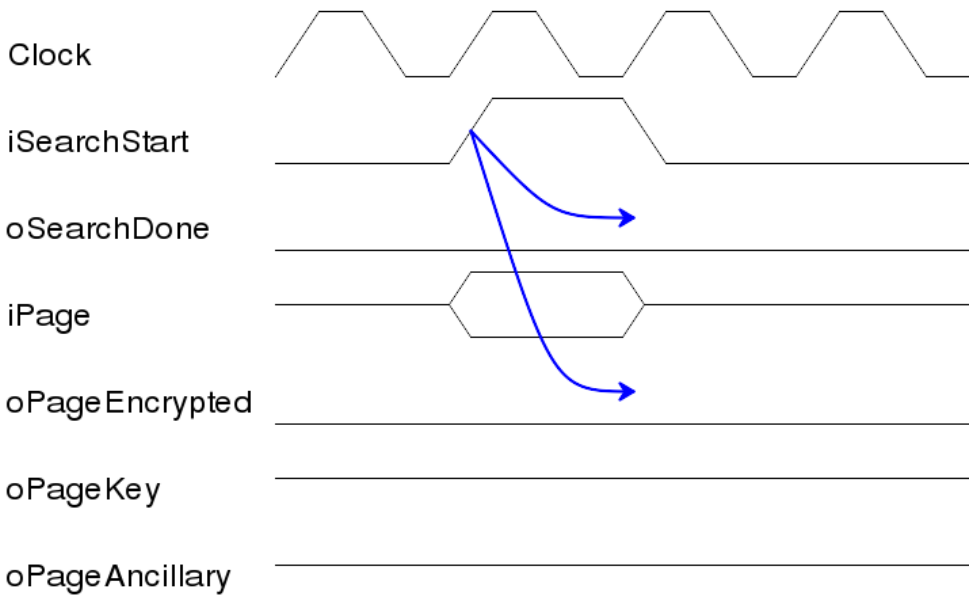

Figure 3.6: Protocol: Table Search, Unencrypted Return

Table 3.4: Page Table Control Protocol Data Signals

Signal Name	Bits	Description
PageBase	20	Base address of the page
KeyIndex	6	Key slot that page points to
PageIndex	6	Table slot to load page–key mapping
Ancillary	128	Ancillary data associated with page

Key Table Control Protocol

The key table control protocol crosses clock domains from the Key Table Control Unit to the Table Unit. The asynchronous protocol described in Section 3.2.3 is used to provide the handshaking and data synchronization. The data set transferred during the handshake is shown in Table 3.5.

Table 3.5: Key Table Control Protocol Data Signals

Signal Name	Bits	Description
KeyIndex	6	Key slot to store key in
Key	128	Decryption key

Generic Control Interface

The generic control interface used by the Control Units provides a fairly common bus interface with dedicated read and write bus lines. The transaction begins with the master device's (CPU or SKU) assertion of the request, address, byte enable, and read–not–write signal. If performing a write operation, the master will also assert its write bus at this time.

The slave device immediately responds with a wait signal to indicate it is processing the transaction. When the slave has completed the operation it asserts its acknowledge for one cycle to indicate either read or write complete. If a read transaction was performed, the read data bus is also asserted by the slave during the acknowledge cycle. The master will end the transaction by de–assertion of the request signal, after detection of the acknowledge signal. The read and write transactions are shown in Figures 3.7 and 3.8 respectively.

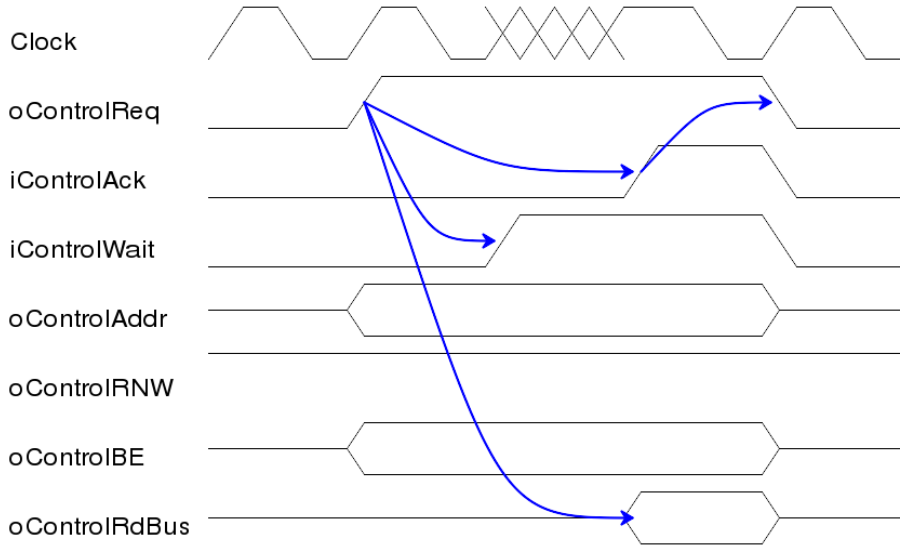

Figure 3.7: Control Interface Protocol: Read

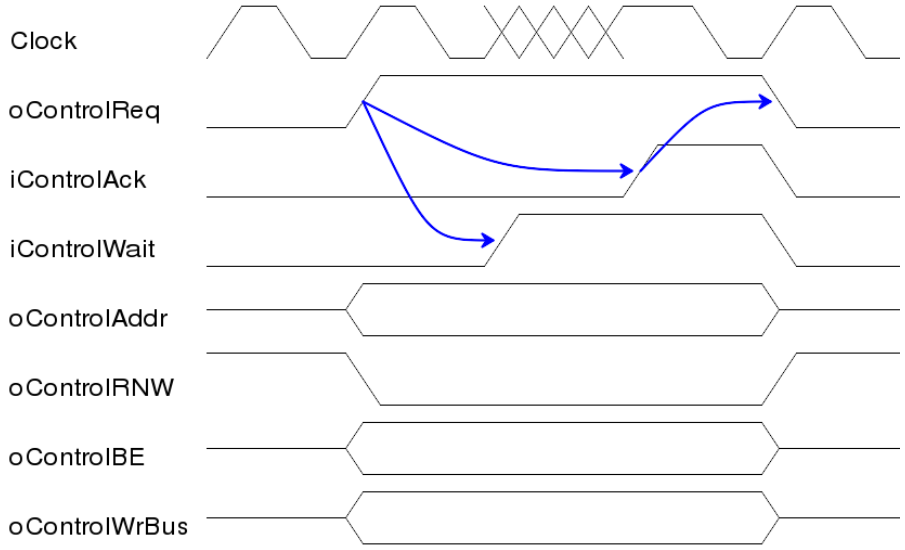

Figure 3.8: Control Interface Protocol: Write

On–Chip Local Bus (OPB) Protocol

Both the Page Table Control Unit and the Key Table Control Unit connect to their respective CPU and SKU buses using the 32–bit IBM On–Chip Peripheral Bus (OPB). Full specifications on this protocol are provided in [36]. Both units are used in memory mapped mode.

3.3 Implementation

This section presents detailed implementation information of components of the Encryption Management Unit in the Secure Software prototype platform.

3.3.1 Bus Modifier / Bridge Unit

The Bus Modifer / Bridge unit provides the high–level modification to the PLB bus instruction stream. The module itself does not perform direct processing, yet it provides a container for a modular pipeline of bus modification subcomponents. Each module uses the subset of PLB signals defined in Table 3.3, and each module interlocks with a CPU–side and a memory–side bus interfaces. At a minimum, two submodules are required: the CPU interface and the Memory interface. Connecting these two modules together ties the memory bus directly to the CPU, without any intermediate active or passive modification.

The primary component inserted into this pipeline is the decryption module. It is inserted using the interlocking subset of PLB signals. This module modifies transaction types, decrypts returned instructions, and initiates search table look–ups. Other modules offering additional functionality can also be easily inserted into this module chain. Examples may include instruction masking, caches, obfuscation routines, and pre–fetching units. Figure 3.9 depicts the modularized and pipe–lined Bridge architecture, including potential modules performing additional functionality.

Two cryptographic block modes were implemented for the Virginia Tech Secure Software prototype. Each mode, when combined with different cryptographic algorithms, can allow decisions between trade–offs between security levels and performance to be made. In both cases the actual cryptographic routine is outside the scope of this work, as both modes are fully compatible with generic interfaces to most common algorithms.

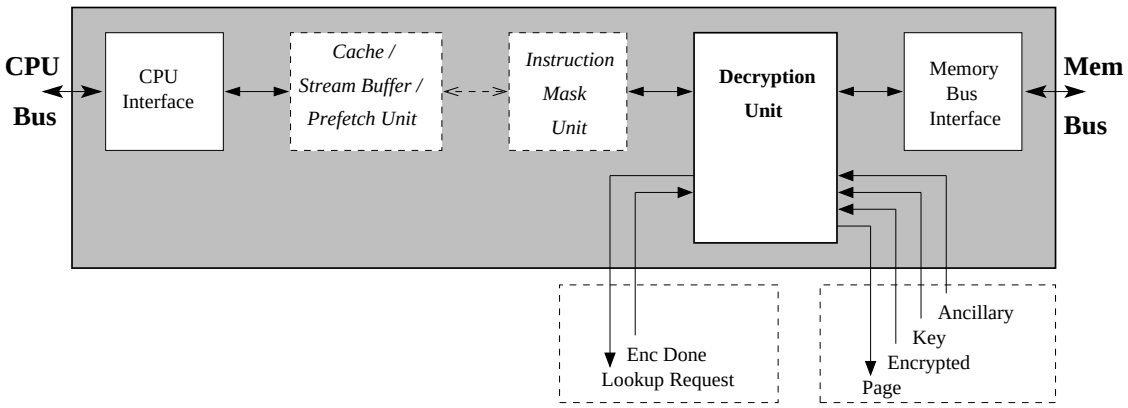

Figure 3.9: Bridge Unit

Block Mode Decryption Module

The decryption modules are the most complex components in the system, as they are ultimately responsible for the primary instruction flow. The PLB bus is a relatively complex bus protocol, and even though a subset of transaction types are used, it still requires resources for proper handling of addressing, target word first, and different transaction sizes, in addition to meeting timing requirements. Decryption of instructions further adds to this complexity.

The direct block–mode module utilizes many data flow elements, including registers and muxes, around a central finite state machine. There are four primary phases of operation in the Block Mode Decryption module. The initial phase places the module into a wait state. Here, the module waits until a PLB transaction begins, signified by the assertion of request by the CPU, de–assertion of abort by the CPU, and acknowledgment of address by the memory arbiter.

While in the wait state, transfer address qualifiers are passed directly between the CPU and memory arbiter. These signals are inhibited in all other states. Additionally, any transaction requests smaller than a 128–bit unit are expanded to 128 bits. Lastly, when the transaction is recognized, a request for encryption status and information is sent to the Table unit.

The second phase processes both the arriving requested instructions and the results of the Table unit look–up. Counters are used to determine how many data beats have arrived, and how many are required for read completion. As the data beats arrive, they are automatically placed into a 256–bit buffer, enough to hold the largest transaction the PowerPC 405 instruction–side interface. The location in the buffer to place the data beats is determined by the corresponding read word address also by the memory arbiter.

The Table unit returns encryption status before the memory completes the read transaction. The encrypted status drives a multiplexer to select the memory driven data bus when

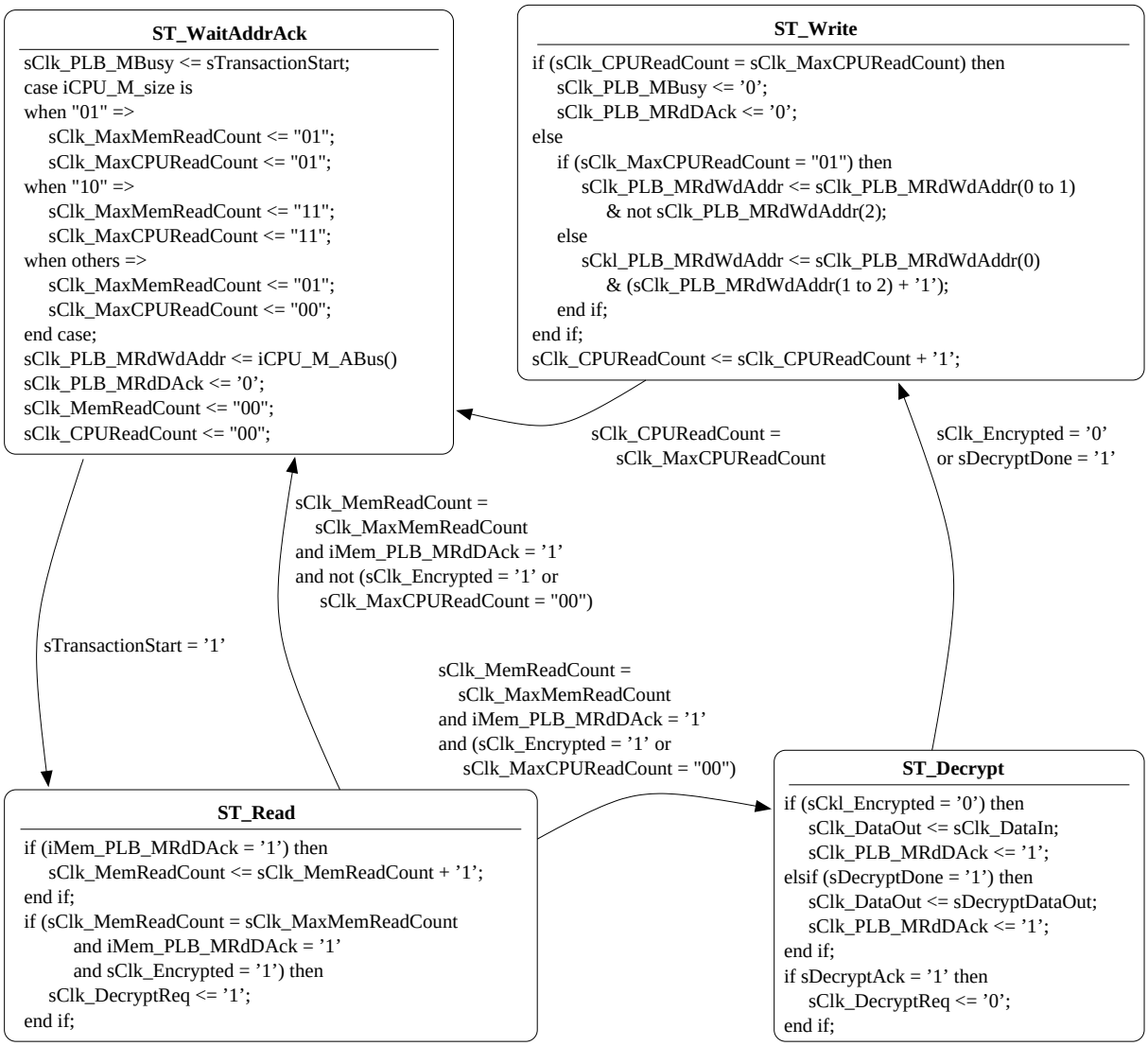

Figure 3.10: Block Mode Decryption Finite State Machine

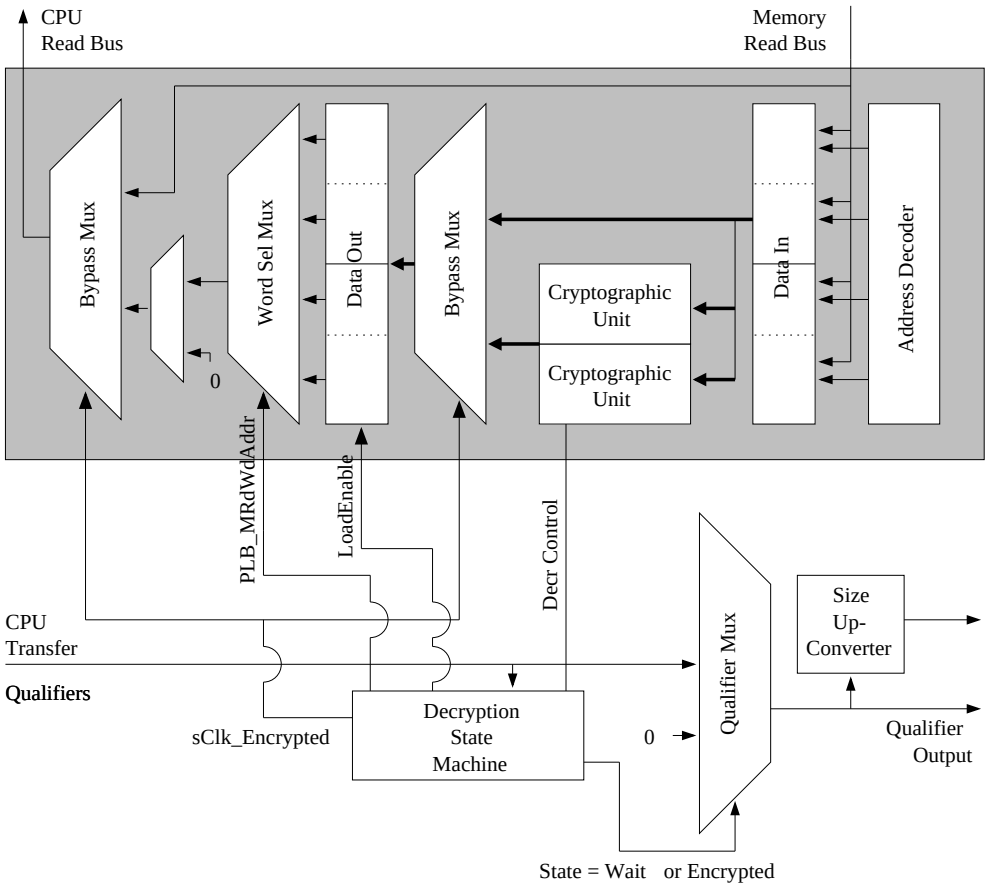

Figure 3.11: Block Mode Decryption Module

not encrypted, or the internally generated data bus when encrypted. Incidentally, this multiplexer is also driven for 64–bit transactions, as they require further processing. When the last data beat arrives, if the transaction is encrypted or if it has been scaled up from 64–bits to 128–bits, the machine transitions to a decryption phase. Otherwise, the transaction was not encrypted and the data beats have already been passed to the CPU, at which point it returns to the waiting state.

The third phase is reached when the transaction is encrypted, it has been scaled up from 128–bits, or both. In the even that an unencrypted 64–bit block was transferred, the input buffer is loaded directly into the output buffer, and the machine transitions to writing the data beat. Alternatively, if the transaction was encrypted then the state will drive the request lines to the cryptographic unit and wait for the cryptographic unit to return from completion. When complete, the decrypted blocks are loaded into the output buffer.

The last phase strictly writes the specific transaction requested by the CPU. This state maintains the target–word–first capability, and in doing so requires that word address counters properly use the correct data beats in the buffer, depending on transaction size. As the CPU data bus multiplexer is still driven during this phase, the phase selects the proper data beats from the output buffer, asserts the correct read word address, and asserts its data acknowledge signal to the CPU. When the requested transaction data beats have been sent, the machine transitions back to waiting for the next transaction.

Counter Mode Decryption Module

Compared to the Block Mode Decryption module, the Counter Mode Decryption module processes instructions as a stream. There are no restrictions on the size of the transaction or the ordering. This significantly reduces the processing complexity of the system.

The Counter Mode Decryption module is divided into three primary states. The first state, and associated logic, is very similar to the Block Mode wait state. As before, this

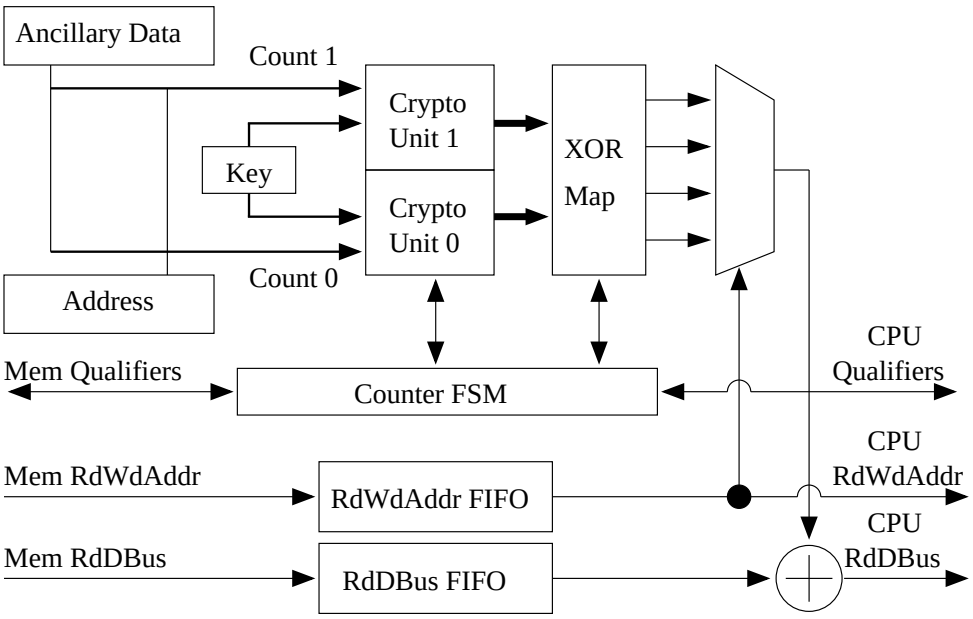

Figure 3.12: Counter Mode Decryption Module

state simply waits for a transaction to begin. The criteria for this is the same as in Block Mode. Additionally, transaction address qualifiers to the CPU and memory arbiter are in pass through mode only when the machine is in this state, and inhibited in all others. Lastly, when a transaction is begun, the Table search is also initiated.

The second phase begins when a transaction begins; however, the state is dependent on the encryption of the counter values. All data beats and corresponding word addresses are loaded into corresponding FIFO units, where they will remain in their proper sequence until the proper counter value has been encrypted. Because single beat transactions do not assert the read word address, the word address is manually injected into the FIFO based on the requested transaction address.

The functionality of this state is driven by the results of the Table search. If the transaction is not encrypted, then an output XOR map is loaded with zero values. Otherwise, a look–up that indicates encryption automatically triggers the encryption process in the cryptographic units. The state waits for the cryptographic units to complete, at which point

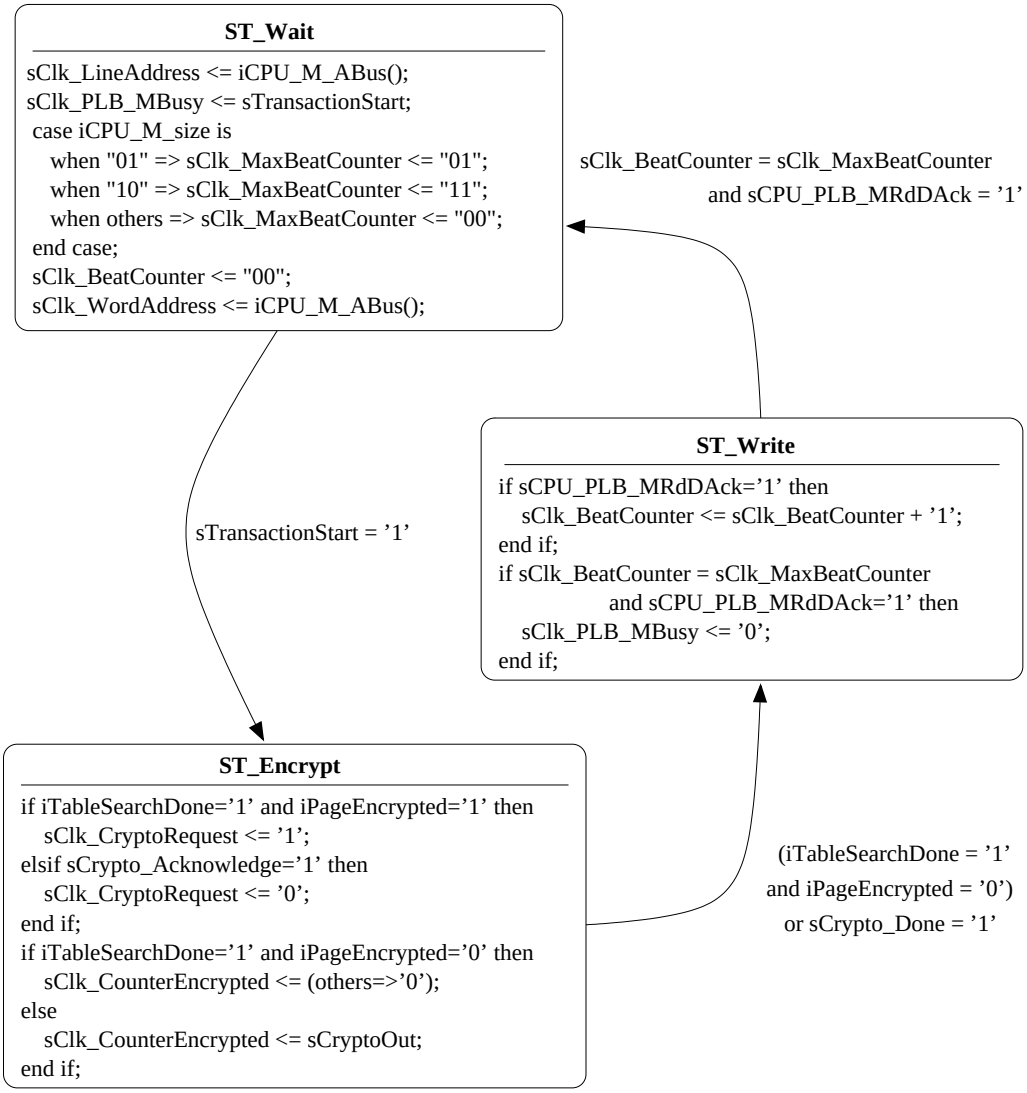

Figure 3.13: Counter Mode Decryption Finite State Machine

their encrypted result is loaded into the output XOR map and the state is complete.

The last phase of this module pulls the requested number of transaction data beats from the data FIFO. The value to XOR against the data beat is selected from the 256–bit XOR map based on the corresponding word address. If the transaction was not encrypted, then the data beat is XORed against zeros, resulting in no change, otherwise the proper encrypted XOR patter is applied to the data beat.

3.3.2 Table Unit

The Table Unit features three look up tables and one content addressable memory (CAM). The CAM [39] is an IP core supplied by Xilinx, and provides efficient speed and low area requirements. It receives a 20–bit page address from from its search interface and searches for the slot that page was written to in its memory. The CAM was configured without a registered output, so the results of the search are valid on the cycle following a change of input address.

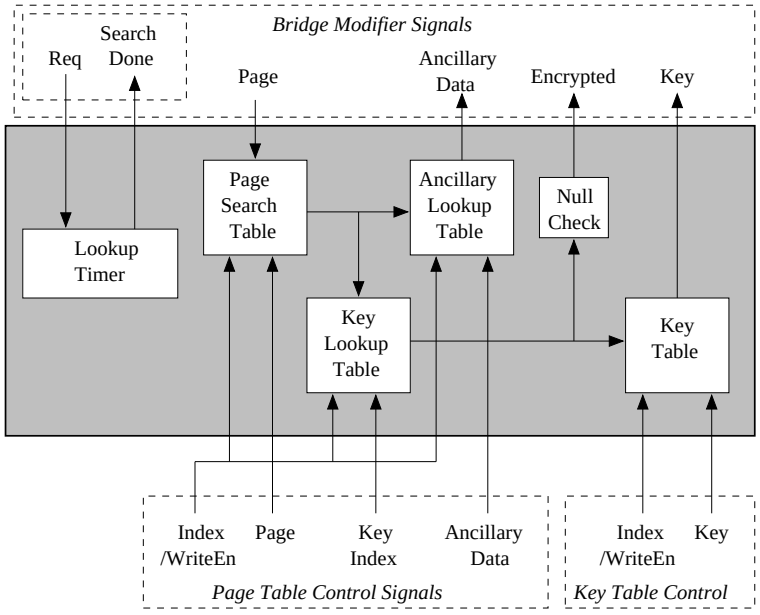

Figure 3.14: Table Unit

The CAM search result is connected to the ancillary data look–up table address and the key slot look–up table address. One requirement of the CAM is that when writing to it, the write signal must be active for only one cycle and the data being written stable for that cycle and the one after it. The synchronizer FSM that communicates over the page table interface holds off the handshaking until both write cycles have completed.

The key slot look–up table allows each page entry to point to any one of the key slots. Without this table, the pages would be directly mapped to the matching slot entry in the key table. Because keys are costly to generate, it is more efficient to have multiple page slots point to a single key slot for pages that share the same key.

Rather than implement the key slot in Xilinx Block RAM, which performs a sequential read access, a very fast asynchronous read, sequential write Xilinx SelectRAM was used. Six 64x1 RAM units were instantiated to create the 64 entry 6–bit look–up table. The combinational output of the CAM, and the fast look–up of the key slot table allow the page slot and key slot to be determined in one cycle.

Both the key and ancillary tables are instantiations of full 128–bit by 64 Xilinx BlockRAM cores. They both read and write sequentially, latching their address on the rising transition of the clock. This relationship between the individual tables is shown in Figure 3.14.

When writing to the Table Unit, the handshaking occurs as described earlier for asynchronous communications. The unit responds to transfer requests from the CPU and SKU with the full handshaking, allowing one cycle to read the information before acknowledging the transaction. Rather than latch the data into registers, the information is loaded directly into the CAM and the tables. All tables require one cycle for a write operation to occur. However, the CAM requires two cycles for loading. Therefore, the page table control interface of the Table Unit adds an additional cycle of delay before acknowledgment.

3.3.3 Control Units

The Page Table Control and Key Table Control units behave in very similar manners. They both provide registers accessible through their generic control interface, they both react when data is written to particular index registers, and they support bi–directional, asynchronous communications with the Table unit. They differ in the register sets they operate with. These similarities and differences are shown in figures 3.15 and 3.16.

The primary functionality necessary for the corresponding bus master devices is writing to the register bank. Each bank consists of eight addressable locations, but easily expandable to accommodate additional registers. When writing, byte–enable addressing is fully supported, although not recommended. Byte–enables allow subsets of a register to be written to natively by the bus.

There are different delays associated with writing to particular registers, such as when writing to the index registers. The handler for the acknowledgment signal is required to assert both *acknowledge* and *busy* for the appropriate duration of the index register writes. This is seen in the diagrams as the *transfer acknowledge logic*. It observes the request and address lines to determine which register is written to, then waits for the unit further upstream to assert its own acknowledge, meanwhile maintaining the *busy* signal. Otherwise, if a regular register is written to or a read operation is requested, *acknowledge* is asserted in the cycle following the bus request.

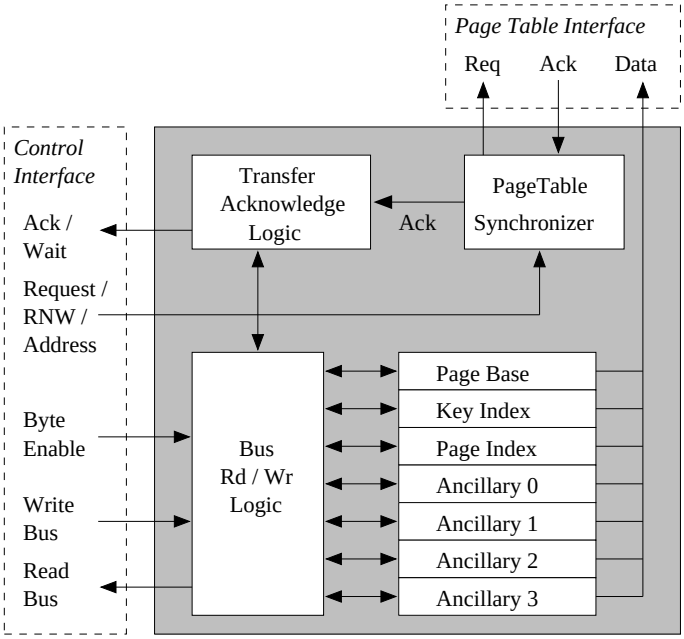

Figure 3.15: Page Table Control Unit

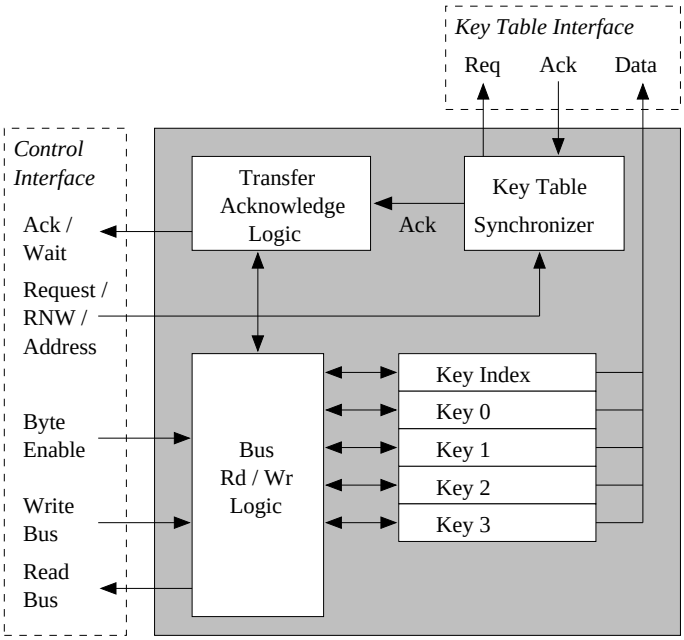

Figure 3.16: Key Table Control Unit

3.3.4 OPB Control Interface Unit

Both the Page Table Control Unit and the Key Table Control Unit connect to separate OPB buses. The control units feature the same generic bus interface, and therefore allow only one Control Unit Interface to be created. To accelerate development time and reduce testing and debugging, the Xilinx OPB IP Interface (IPIF) was used to abstract the lower level OPB protocol into a more generic bus read/write interface.

The unit also translates from bus addresses, which are byte–addressed, to the generic bus addresses, which are word–addressed. This requires a simple shift of the addresses. This unit is not a memory and does not support byte addressing mode, only word–aligned transfers for register writing. The IPIF was configured for use in address mode as it was the most compatible with the control unit interface addressing and protocol used in the EMU. The generic interface the IPIF produces is attached directly to the control unit interface without active or passive logic.

The IPIF was used because it is rapidly implemented within the system. Furthermore, Xilinx provides IPIF compatible interfaces for other bus protocols, such as the PLB bus. As the IPIF signals do not change between buses, the only requirement for switching either control unit to PLB would be to instantiate the PLB IPIF component, adjust the configuration constants, and attach to the appropriate bus in the EDK.

Chapter 4

Modeling

When evaluating the effect on application performance with an Encryption Management Unit, there are two aspects that should be considered. First, the overall delay resulting from using EMU must be modeled. Once this is sufficiently formulated, it can be incorporated into additional models representing the relative effect of the EMU with respect to varying application types and system loads.

4.1 Latency Modeling

To effectively analyze delay added by the EMU, total run time for an application is split into two parts: time stalled on an instruction cache miss and time not stalled on a cache miss. Grouping the two behaviors this way isolates the post–cache memory system and abstracts details including application and system behavior into one term. This is shown in Equation 4.1.

The t_{busy} term reflects any time the system is not stalled on instruction fetching. This includes waiting on system calls, processing data, running instructions from cache, and more. Furthermore, Equation 4.2 expresses the percent of an application run time spent

stalled waiting on instruction fetches to complete. The percent of time stalled will vary with application type and system load.

$$T_{Total} = t_{busy} + t_{stalled} \tag{4.1}$$

$$\%T_{stalled} = \frac{t_{stalled}}{t_{total}} \tag{4.2}$$

Equation 4.3 expresses the total time spent stalled on cache misses in an unencrypted application. The M_{App} term reflects the number of cache misses of an application. L_{Mem} reflects the average latency of the cache miss. It is possible for memory latency to vary between applications. Processor architectures supporting speculative execution and instruction pre–fetching can affect average latency. If the correct branch is taken more than half of the time, then the average latency will decrase. If the correct branch is taken less than half of the time, average latency will incrase as the processor will be stalled not only on the missed instruction, but will also have to wait for the incorrect pre–fetch to complete. The effectiveness of the prediction unit is dependent on application characteritics.

$$t_{Stalled} = M_{App} \times L_{Mem} \tag{4.3}$$

To formulate the time stalled in an encrypted application, Equation 4.3 is modified to account for differences between encrypted and unencrypted fetches and latencies. In the Secure Software architecture, an application can have encrypted and unencrypted pages. Time stalled in an encrypted environment is shown in Equation 4.4. M_{Enc} is the number of encrypted cache misses, while M_{Unenc} is the number of unencrypted cache misses for an application. $M_{Enc}+M_{Unenc}$ will equal M_{App} from 4.3. L_{EMU} is strictly the additional latency of the EMU.

$$t_{StalledEnc} = M_{Enc}(L_{Mem} + L_{EMU}) + M_{Unenc}L_{Mem} \tag{4.4}$$

In a standard implementation, the EMU acts purely as a delay element within the memory stream. There are no acceleration or performance enhancing features within the EMU. Therefore, the EMU directly adds a fixed latency based on decryption mode and algorithm

for encrypted transactions. Inserting Equation 4.4 into 4.1 produces Equation 4.5, which represents the total execution time as a function of EMU latency, memory latency, and application behavior.

$$T_{TotalEnc} = t_{busy} + t_{StalledEnc} \tag{4.5}$$

With the effect of the EMU framed within total execution time, a new model is generated to show the relative change in performance by the EMU with an encrypted application. The relative effect is an important metric for understanding the performance change caused by the EMU. It places the direct effect of the EMU within the context of the application type and system load.

Application types, which are important to this modeling, vary on two primary parameters: system call usage and instruction locality. System calls are issued by an application voluntarily, producing a context switch that results in the application waiting until the system call completes. This includes instances such as device I/O, sleep timers, and more. Instruction locality represents how effective an application is at keeping its instructinos in lower level cache units, which also depends on the hardware cache size and replacement strategy. System load reflects the level of involuntary waiting by the application such as waiting in the process queue on a heavily loaded system.

High code locality, high system call usage, and high system load individually increase the time busy relative to $t_{stalled}$. If $t_{busy} \gg t_{stalled}$ for an unencrypted application, then there will be little difference in performance with increases to $t_{stalled}$ when encrypted. Likewise, if $t_{stalled} \gg t_{busy}$, then small increases to $t_{stalled}$ due to the EMU will have a larger effect on the total execution time.

4.2 Modeling Slow Down

Amdahl's equations for speed–up provides a foundation for evaluating the absolute effect of the EMU relative to application types and system loads. As the EMU adds latency, the

corresponding slow–down variant from Amdahl is used instead, shown in Equation4.6.

$$SlowDown = \frac{T_{new}}{T_{old}} = \frac{T_{TotalEnc}}{T_{Total}} \tag{4.6}$$

Equation 4.6, expanded in Equation 4.7, produces a model supporting encrypted and unencrypted pages. If the application instructions are always encrypted, further reductions are possible. Treating all fetches as encrypted results in Equation 4.8. Substituting the second term in that equation with 4.2 produces the alternative form 4.9. This last equation is also featured in related work [25], which serves to support this model.

$$\begin{aligned}
&SlowDown \\
&= \frac{t_{busy} + M_{Enc}(L_{Mem} + L_{EMU}) + M_{Unenc}L_{Mem}}{t_{busy} + M_{App}L_{Mem}} && (4.7) \\
&= \frac{t_{busy} + M_{Enc}(L_{Mem} + L_{EMU})}{t_{busy} + M_{Enc}L_{Mem}} \text{ when } M_{Unenc} = 0, M_{Enc} = M_{App} \\
&= \frac{t_{busy} + M_{Enc}L_{Mem} + M_{Enc}L_{EMU}}{t_{busy} + M_{Enc}L_{Mem}} \\
&= 1 + \frac{M_{Enc}L_{EMU}}{t_{busy} + M_{Enc}L_{Mem}} \\
&= 1 + \frac{M_{Enc}L_{EMU}}{t_{busy} + M_{Enc}L_{Mem}} \times \frac{L_{Mem}}{L_{Mem}} \\
&= 1 + \frac{L_{EMU}}{L_{Mem}} \times \frac{M_{Enc}L_{Mem}}{t_{busy} + M_{Enc}L_{Mem}} && (4.8) \\
&= 1 + \frac{L_{EMU}}{L_{Mem}} \times \%T_{stalled} \text{ from 4.2} && (4.9)
\end{aligned}$$

Chapter 5

Benchmarking

The benchmark used in this work was created primarily to measure latency on the processor instruction–side bus. It uses similar methods for benchmarking as the LMBench memory latency benchmark. The lmbench's *lat_mem_rd* has already been validated for accuracy. Latencies for cache hits and misses within the instruction and data–side cache on the PowerPC 405 are the same [7]. Both data and instruction interfaces attach to the same processor bus and access the same external memory. Along this line of reasoning, validation of the benchmark created in this work, hereafter referred to as iBench, is gained when the validated results of the *lat_mem_rd* benchmark matches that of iBench.

5.1 Methodology

LMBench uses a circularly linked list, with each list element consisting only of a pointer to the next node. A single C–language command is issued repeatedly to fetch the data contents stored at the location of a pointer, and set the pointer to that value. This command is synthesizable by most compilers and architectures to a single instruction, where data is loaded from the location pointed by a register into that same register. This instruction

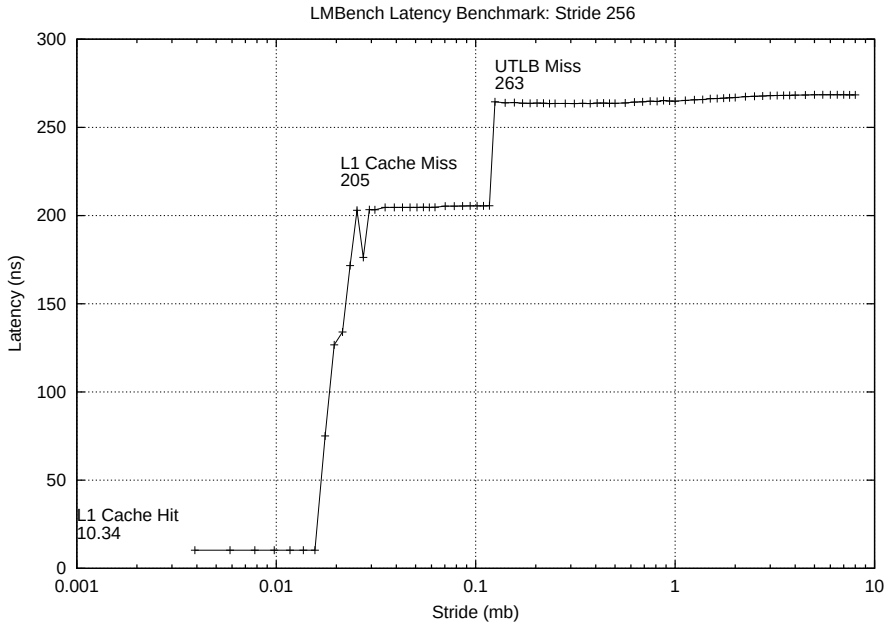

Figure 5.1: LMBench Latency Benchmark: Data memory subsystem

is executed one–hundred times sequentially, inside of a loop that will iterate the desired number of times, which is $\frac{DesiredTimes}{100} + 1$. As the loop is dominated by the one–hundred load instructions, and no branch instructions are between loads within the loop, the instruction pre–fetch unit keeps the instruction pipeline filled. Furthermore, as the 100 instructions require 400 bytes, this loop easily remains inside the instruction cache of size pf 16 kBytes.

The LMBench *lat_mem_rd* benchmark begins by establishing a circular linked list, where each element points backward in memory by a fixed length, referred to as the stride. The element in the lowest addressable area of the memory, the last one pointed to, is set to recirculate and point back to the highest list address. Elements point backward to reduce the possibility of hardware data–side pre–fetching polluting the testing. The test is run for a number of iterations determined by the upper level LMBench framework to ensure the execution time is sufficiently within the resolution of the operating system timer.

The memory area of pointers that are a stride–width apart is increased and the circularly linked list is recreated in the larger space, and the test is rerun. This testing iterates for

increasingly larger memory spaces until a predefined maximum size of memory is met. As the instruction pipeline containing the load instruction is easily kept full, the fetching of data from caches or memory becomes the dominant bottleneck. Granted, when the linked list memory area is small, the linked list is able to fit into data cache. Execution time for the load instruction, without counting the time the CPU is stalled on the data fetch, is a single clock cycle. The total time stalled when fetching for each test is provided in Equation 5.1. Average latency is obtained by dividing the stalled time by the number of instructions executed, as shown in Equation 5.2.

$$StalledTime = ExecutionTime - (NumberOfInstructions \times ClockPeriod) \quad (5.1)$$

$$Latency = \frac{ExecutionTime}{NumberOfInstructions} - (ClockPeriod) \quad (5.2)$$

When walking the linked list for the first iteration, the area that is traversed is relatively small and can fit into the data–side cache. Accounting for instruction execution time, the time spent in these iterations are dominated by fetches from data cache. In the next phase, the number of strides cannot fit into the data cache. The time spent for this phase is dominated by fetches from external memory. Lastly, a final phase consists of the case where the area traversed becomes large enough to not fit in the area mappable by the universal TLB of the MMU, which results in many page miss exceptions in the operating system and increasing average latency.

5.2 iBench Design

As the methodology used in *lat_mem_rd* provides a clean way of testing and calculating latency it was decided that an instruction–side version of the LMBench data–side latency benchmark be created. A direct correlation to walking a linked list is to instruction branch in fixed stride lengths in a given memory area, with branches wrapping around from the end back to the beginning. To setup this branching structure for varying memory widths, iBench would have to allocate and write instructions into memory during run–time for later

execution. While self–modifying code is possible, and has been done with certain iterations of this benchmark, it is not permitted with the Virginia Tech Secure Software architecture. The basic reason for this is that there is no data–side cryptographic unit, and therefore the program cannot create properly encrypted instructions.

The solution to the variable memory width problem was to write a utility that would inject assembly code consisting of the branching table setup for the correct size and stride into a base source file. A file would be generated for each desired width and stride. Furthermore, because of having to establish the table in the executable before run–time, it was necessary to use PowerPC assembly code. Problems that would arise attempting to force high–level languages to produce such an exact table were avoided. Assembly was also used to ensure that *iBench* would easily fit into instruction cache. Heavy–weight libraries, including those needed for system calls, print formatting, and program init/finialization were avoided. Lastly, assembly code was required to ensure proper branching techniques were used.

In *lat_mem_rd* there were two nested loops for each memory width test. As load instructions were performed in blocks of one–hundred, an inner loop counted off how many blocks of one–hundred were necessary to achieve at least $\frac{MemorySize}{MemoryStride}$ strides. The outer loop counted down how many iterations were requested for the particular test. While overhead for these was minimal, the PowerPC offered a superior solution that had no overhead of either loop.

The PowerPC 405 instruction set contains a dedicated 32–bit counting register, *CTR*, which can be initialized to a particular count value. To compliment this register, there are a number of branching mnemonic instructions which will automatically decrement this counter and test if the *CTR* register has, or has not, reached zero. Depending on the result of that test, and the mnemonic used, the absolute or relatively addressed branch may be taken. The complete operation of decrement, test, and set next instruction address occurs within the same cycle.

With iBench, *CTR* is initialized to $\frac{MemorySize}{MemoryStride} \times Iterations$. The overhead of the loops seen in *lat_mem_rd* is packed into the branch instructions. Following the conditional branches

is an unconditional branch to the location stored in the *LR* register. When the test frame established the counter and branched into the jump table, *LR* was set as the return location of the calling function. An unconditional branch to this location, while walking the table, will return to the next instruction after the calling instruction.

The default behavior of the branch instruction used in iBench, when traversing the branches, is to not pre–fetch the next location. All conditional branch instructions to *CTR* are followed immediately by unconditional branches to the calling function in *LR*. Both of these instructions reside in the same cache line fetched from memory. As a result, the hardware instruction pre–fetch unit will pre–fetch instructions after the branch instruction location, instead of the branch target. However, these instructions are already in cache and do not require pre–fetching. Lastly, as the PowerPC 405 will speculatively access up to nineteen instructions down the predicted path, the first instruction is the unconditional return to *LR*, which places the next instruction of the path as coming from the calling function. Because this speculative access occurs on every stride, the entire path remains in cache under the least recently used cache replacement algorithm. This avoids the pre–fetch unit not only from reducing delay on a correctly predicted pre–fetch, but also from adding delay on a miss–predicted pre–fetch.

5.3 iBench Validation

The PowerPC 405 on the Virtex–II Pro FPGA contains a minimal memory subsystem. In the CPU hard processor core is 16 kB of instruction cache and 16 kB of data cache. Each cache unit attaches to the same processor local bus. Fetches from memory are performed similarly, and use the same external memory controller. Fetching instructions or data contained in cache requires one cycle operating at CPU core frequency. If the respective instruction TLB, ITLB, or data TLB, DTLB, do not contain the page entry requested, three CPU cycles are spent automatically retrieving the entry from the universal TLB, UTLB. A software

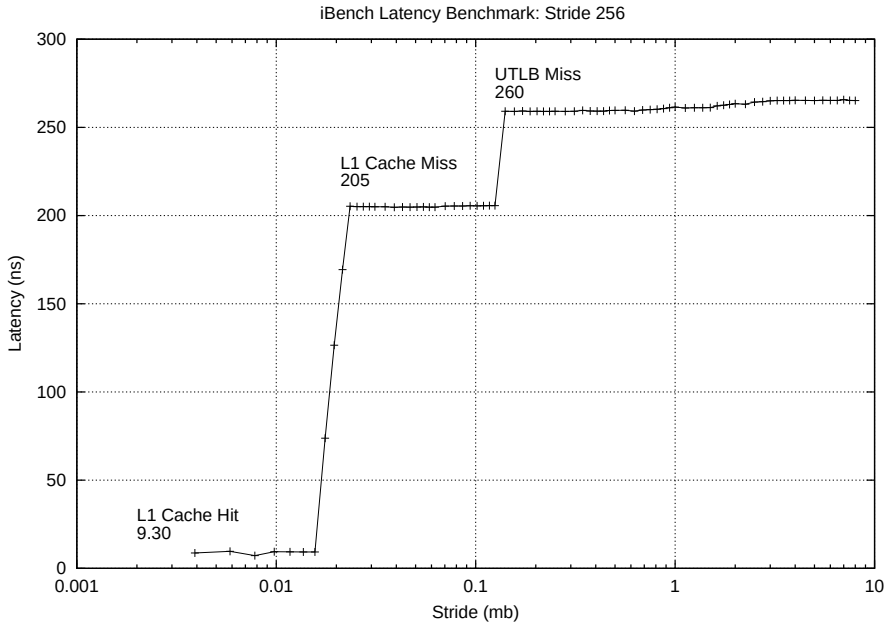

Figure 5.2: iBench Latency Benchmark: Instruction memory subsystem

exception is generated in the event the entry is also not in the UTLB.

From this structure, three distinct plateaus of latency, with respect to area, are expected. The plateaus represent latency for cache hits, memory fetches from cache misses, and TLB misses. This is verified by the results from *lat_mem_rd* in plot 5.1.

The same stride is run using iBench on the same platform, with the bus decryption unit entirely removed to ensure test purity. The results are shown in plot 5.2. Comparing the first plateaus, where latency is primarily from local cache, iBench shows an average latency of 9.3, compared with a LMBench result of 10.3. Future revisions of this benchmark will account for this discrepancy, however it is only 10% variance and cache latency is not of particular interest to this work. Memory latency is the next plateau, and of the most importance to this work. The data points between LMBench and iBench reveal iBench is accurate within 0.25%. Comparing TLB miss latency plateaus shows that iBench has less latency, however it remains within 1.75% of *lat_mem_rd* results.

The processor bus has two primary phases: an address acknowledge phase followed by

Table 5.1: Memory Latency by Stage

Stage	Cycles	Period	Latency
ITLB Miss	3/16	3.33 ns	0.625 ns
Cache Fetch	1	3.33 ns	3.33 ns
Bus Address	3	10 ns	30 ns
Bus Wait	13	10 ns	130 ns
Bus Data	4	10 ns	40 ns
Total			204 ns

a data acknowledge phase. On a cache miss, the L1 cache will request a transaction for an eight–word cache line of the missed address. The external memory controller will acknowledge the address and begin retrieving the data or instructions from memory. Once the external controller–memory fetch is complete, the requested transaction is returned in 64–bit transfer beats to the cache, where it is stored and returned to the processor.

With respect memory bus latency, observations using internal bus analyzers have revealed the timing characteristics for the phases of a bus transaction. First, once the operating system has passed its initial boot stage, it turns the cache and the MMU on. With both the cache and MMU enabled, the processor switches from single beat and four–word line transactions to eight–word cache line fetches from external memory. This simplifies characterization of the data transfer phase as the external memory controller will primarily return four 64–bit data beats. Secondly, when there is no contention on the processor bus, a total of three bus clock cycles are required to acknowledge the transfer request signal.

Cycles, clock periods, and resulting latency for each stage of a memory request is listed in Table 5.1. These numbers support the memory fetch latencies measured by *lat_mem_rd* and iBench. In the plateaus dominated by memory fetch latency in plots 5.1 and 5.2, the shadow TLB (ITLB and DTLB) and universal TLB are not often replaced. This reduces the average

latency impact in updating the shadow TLB from the UTLB as they are not updated for every fetch, but after $\frac{PageSize}{Stride} = \frac{4096}{256} = 16$ fetches. Comparing the analytical memory latency estimate with measured latency further illustrates the validity of both benchmarks.

Chapter 6

Results

The intent of characterizing the EMU is to determine the difference between running encrypted and unencrypted applications on a given platform. To accomplish this, latency and slow down measurements are taken between encrypted and unencrypted executables under varying system load. Although the block and counter modes of the EMU support both encrypted and unencrypted executables, a third configuration connecting the memory bus directly to the CPU provides the base comparison. This base profile still contains the interfaces for CPU and SKU configuration, allowing the kernel and SKU to maintain a constant performance effect of page loading and key management.

Two primary focuses of performance in this work are the direct and relative effects of the EMU. These correspond to the measurable parameters of latency and slow down. Latency describes time required to complete an instruction fetch in various configurations. Slow down describes the relative effect on performance when utilizing the EMU.

The analysis presented in this chapter focuses on five primary modes of operation: the base profile running unencrypted executables, unencrypted and encrypted executables using block mode, and unencrypted and encrypted executables using counter mode. All measurements were taken by the the *iBench* instruction–side memory latency benchmark developed in

Chapter 5.

It should be noted that these measurements do not reflect actual application performance. To achieve precise latency measurement, the benchmark artificially executes instructions in a way that local cache is not effective. It also does not contain any system calls or data operations that would have any significant impact and lessen the performance degradation of the EMU.

6.1 Latency

Latency measurements for the five configurations are shown in Figure 6.1. Immediately visible are three plateaus of data points. The first plateau represents execution purely from L1 cache, and is about 10 ns of latency for each profile. The PowerPC 405 has a single L1 cache, without additional internal or external L2 and L3 caches. Therefore, the second plateau represents external memory latency resulting from cache misses. The final plateau is a result of TLB misses. This benchmark calculates average latency, therefore there are $\frac{PageSize}{Stride} = \frac{4096}{256} = 16$ fetches within a single page. When the system is loaded with page misses, the time for a TLB miss divided among the fetches within each page.

Figure 6.2 depicts the average latency for a memory fetch with profile. The average is computed from the values contained in the second plateau of Figure 6.1.

Analyzing the latency plots demonstrates that EMU latency for unencrypted execution is close expected results. One observation is the slightly higher latency for unencrypted executables in block and counter modes within the EMU system, compared to the base system without an EMU.

In block mode, there is no additional processing incurred when processing unencrypted full cache–line sized transfers. Each 10 ns is a single cycle on a 100 MHz bus. This cycle of delay arises from the lack of support for fetch pipe–lining in block mode. Because the

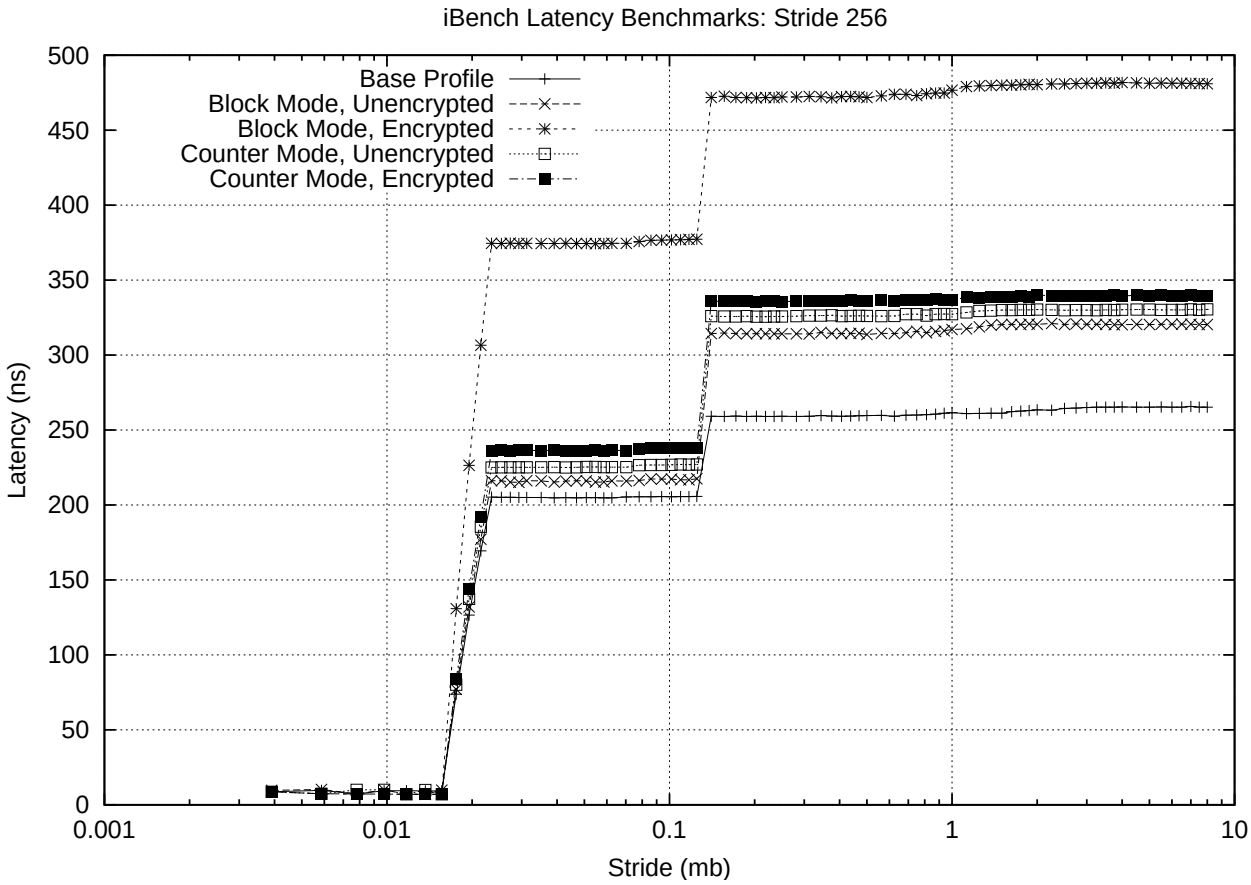

Figure 6.1: Complete iBench latency measurements

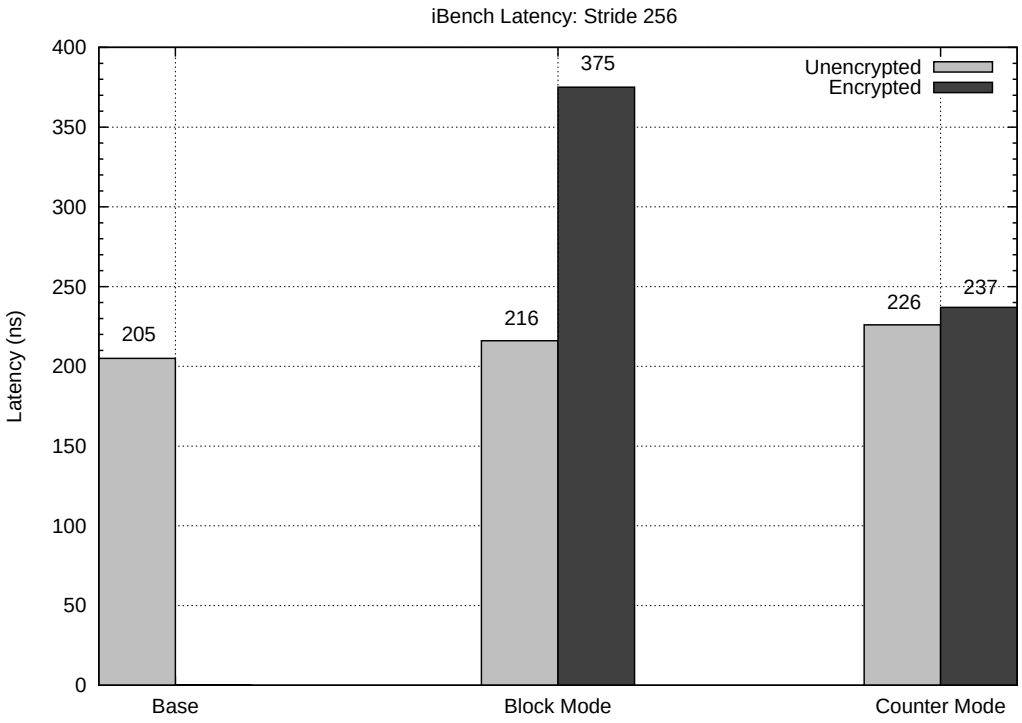

Figure 6.2: iBench memory fetch latency measurements

counter mode also does not support fetch pipe–lining, a cycle of delay is also attributed to this reason. However, counter mode has an additional cycle of delay for unencrypted transfers due to all fetches, encrypted or not, passing through the data FIFO.

The difference in latency between encrypted executables in block mode is as expected. The unit requires four cycles to completely buffer eight–word transactions, and then requires twelve cycles to decrypt with dual cryptographic cores. Once decrypted, the result is immediately transmitted. The total latency for decryption is $4 + 12 = 16 cycles$ or $16 cycles \times \frac{10ns}{cycle} = 160ns$ more than unencrypted execution. This shows the expected and actual latency in block mode encrypted executable processing precisely match.

Counter mode also exhibits expected latency. Latency comes from any additional time spent encrypting the counter, beyond the standard memory access latency. The memory access latency is the time after address acknowledge and before the first data beat arrives. An ideal system would complete counter encryption before any data arrives, however the prototype platform takes slightly longer to encrypt than to fetch.

As shown in Chapter 5, a normal access is 13 cycles long. Before encrypting, however, the table look–up must occur. This completes in two cycles after address acknowledge. This is followed by 12 cycles of encryption, and one final cycle for latching the output into the XOR map. The result is a transaction consisting of a total of 15 cycles. Compared with a normal access of 13 cycles, the counter mode encrypted latency is indeed 2 cycles, or $20ns$. Furthermore, this compares favorably with the 14 cycles of counter mode unencrypted latency, which $13+1FIFO$ cycles correctly relates to $10ns$ of difference from the base profile.

6.2 Slow Down

Figure 6.3 shows the slow down of *iBench* execution time for each profile, and under 1x, 2x, and 4x loads. Each load indicates the number of concurrent instances of the benchmarks running when performing measurements. The 1x load corresponds to the same measurements

used in Figure 6.2. All slow down results calculated from equations based on time or latency are included in Appendix C.

Slow down was calculated using Equation 4.6, where the raw execution time of the benchmark is used. The *old* time parameter used in the equation is the base profile for each load. Each corresponding load for each profile is used as the *new* parameter.

The most important observation from these results is that slow down is reduced as load increases. This is to be expected; as load increases, the applications incur an overhead from context switching and other operating system management. The base profile also suffers this same context switch penalty, which is why difference in slow down is minimal. The encrypted applications show a speed up as the system becomes more heavily loaded, relative to an unloaded system. This correctly follows the reasoning that as an application is stalled less often on instruction fetches, the encrypted slow down will converge to the unencrypted slow down.

The slow down results verify the model derived earlier in Chapter 4. For 1x load, it is a safe assumption that the benchmark is stalled approximately 100% of the time on a memory fetch. L_{Enc} and L_{Mem} of Equation 4.9 are substituted for the external memory fetch latencies in Figure 6.2. Memory latency is defined as time spent fetching from unencrypted external memory. Encryption latency is the time spent fetching and decrypting from external memory, less the time spent on an unencrypted fetch.

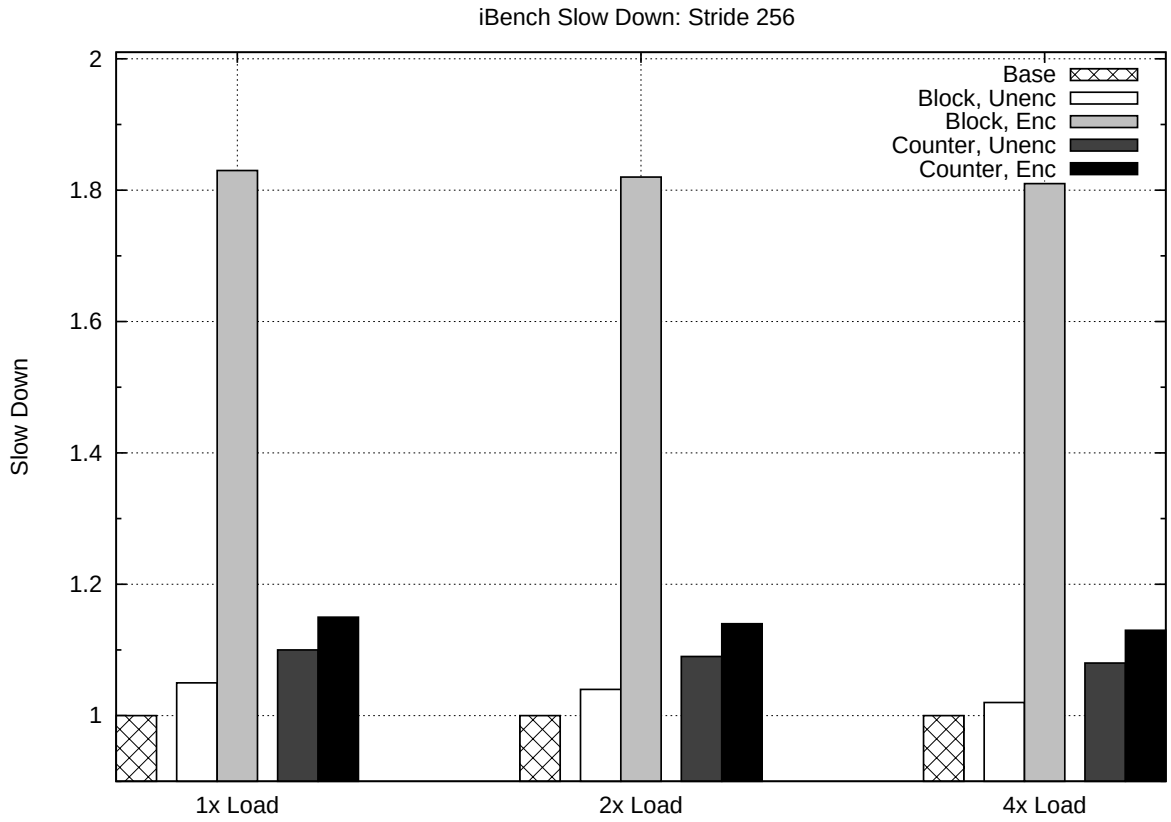

Figure 6.3: Slow down over various loads and profiles

$$BlockUnencSlowDown = 1 + \frac{204ns+1PrefetchLoss}{204ns} = 1.05 \quad (6.1)$$

$$BlockEncSlowDown = 1 + \frac{204ns+12Decrypt+4Buffer+1PrefetchLoss}{204ns} = 1.83 \quad (6.2)$$

$$CounterUnencSlowDown = 1 + \frac{204ns+1FIFO+1PrefetchLoss}{204ns} = 1.10 \quad (6.3)$$

$$CounterEncSlowDown = 1 + \frac{204ns+2Lookup+12Encrypt-13Fetch+1FIFO+1PrefetchLoss}{204ns} = 1.15 \quad (6.4)$$

For block mode, solving Equation 4.9 under 1x load results in a calculated slow down of 1.83 for encrypted execution, and $1.05ns$ for unencrypted execution. In counter mode, encrypted execution slow down is 1.15, and 1.10 for unencrypted execution. These directly correspond to the measured slow down using *iBench*.

For 2x and 4x execution, slow down is slightly mitigated in each profile. The latency of encrypted or unencrypted transactions certainly do not change. Instead, the percent stalled parameter of Equation 4.9 is decreased from the 100% used in 1x loaded benchmarks, as applications spend less of a percent of time stalled on encrypted instructions as load is increased.

Chapter 7

Future Directions

There are a number of interesting possibilities for improvement and extension of the work presented in this thesis. With the EMU there is opportunity in design, performance, and its feature set. The iBench benchmark offers similar opportunities for expansion.

7.1 EMU Design Directions

The most immediate direction of the EMU within the Secure Software Project would likely be the addition of data protections. The Secure Software project is currently exploring a number of options for providing increased protection of program data. Many platform options require the encryption and decryption as data is accessed from external memory.

The instruction–side decryption unit cannot be reused, as the data–side bus interface handles more complex transactions than the instruction–side bus. Additional transaction types such as master–terminated or slave–terminated burst and byte steering modes can be very difficult to keep aligned for certain block cryptographic modes. Furthermore, the data–side bus can access memory mapped I/O (MMIO). Effort will be required to ensure transactions conform to requirements by the slave device. If protections are afforded to

MMIO, additional logic may be required to handle instances where different slave devices using different credentials reside within the same page.

There are several ways for the EMU support data protection. The first method would insert a data–enhanced EMU between the processor data–side interface and the processor local bus, similar to the way the EMU presented in this work is located. The second usage model would partially integrate the instruction–side and data–side encryption management unit. Each unit would remain unique, but they would share state information such as page mappings and keys. The third method would create a small bus and arbiter, and connect the processor instruction–side and data–side bus interfaces to this EMU managed bus. The EMU would be extended to properly support the additional data transaction types. This last option would have similar performance as the first two options, however it would eliminate the need for two cryptographic units and/or two sets of page mapping tables.

A third possible direction for the EMU takes advantage of advances in run–time reconfigurable hardware. With the current state of technology, it is possible to selectively reconfigure logic in specific areas of integrated circuits that support such operations. While the security of such a model would require validation, it is possible to associate a specific cryptographic set with a particular page in physical memory, similar to the way keys and ancillary data are associated with pages. This would allow a secure application to select cryptographic algorithms, cryptographic modes, and possibly obfuscation [40] [41] techniques. Obfuscation would not provide security by itself, but could be used to further increase the difficulty of various snooping and modification attacks on the platform.

A run–time reconfiguration unit would be added to the EMU that could handle individual configuration for cryptographic algorithm and operational mode modules. The cryptographic operational mode module is a reconfigurable version of the bridge modification unit in the EMU. The cryptographic algorithm module is likewise a reconfigurable version of the static cryptographic routine in the EMU. As the versions of the modules in the current EMU are already highly modular and use well defined interfaces, conversion to a run–time reconfigurable

system should not require much modification to the rest of the system.

The last improvement would use the performance monitoring features installed in various locations of the EMU. These signals represent activity for a specific monitor, such as an encrypted fetch or non–encrypted fet, or page table write with zero or non–zero key pointers. All of these strobes are exported outside of the EMU and can be connected to external probes or internal monitoring cores. Regardless of the mechanism, these signals can provide additional insight into the operation and performance of the secure architecture.

7.2 EMU Performance Directions

In addition to functional directions described in the previous section, the EMU implemented in this work is receptive to improvements in performance. These include the use of pre–fetching [42] and stream based [43] processing. In fact, with some secure architectures using pre–decryption, applications exhibiting high locality have even demonstrated an improvement in performance [44].

Another improvement would re–enable address pipe–lining of the processor bus. Presently, decryption units operate one transaction at a time, allowing time for the transaction to complete before servicing the next. Subsequent requests, which could begin processing during the current transaction, are held.

Serializing requests can also result in additional performance loss. If the bus lock is lost by the instruction–side interface of the CPU, then a data request could take over. If that data request is not complete by the end of the decryption cycle, the instruction–side unit has the additional latency penalty for the data transaction to complete. Using additional chip resources it is possible to create a set of ping–ping buffers or FIFO units that would allow transaction requests from the CPU to serviced while the current transaction is being decrypted.

In addition to enabling pipelined access to the bus, improvements in the control mechanism for counter mode are also possible. Bypassing the FIFO for unencrypted transactions will help reduce latency for normal, insecure applications.

7.3 iBench Directions

There are two key directions for *iBench*. The first improvement would increase the usability of the benchmark. Presently, an individual executable is created for each stride and size combination. Depending on the combination of stride and size, there are on the order of one hundred executables created. This is necessary to establish the wrap around functionality for the given memory size without changing instructions during run–time.

A more efficient solution would further emulate what the LMBench latency benchmark does. For each iteration of stride and size, *lat_mem_rd* builds the linked list pointers using the current stride and over the current memory area. A parallel to this within *iBench* would require capabilities similar to self–modifying code. Once a memory space is allocated, *iBench* would write individual branch instructions into memory at a set stride length, and would cap the ends of the memory region with wrap–around branches.

Self modifying code is not typically a problem with standard architectures. However, issues arise when benchmarking the EMU due to the protections offered by the EMU. For *iBench* to measure EMU latency, the memory pages containing the branch table must be flagged as encrypted. However, *iBench* must then write to those memory in a manner that would allow proper decryption of the instructions. Either a "null" encryption could be used such that the "encrypted" output is the same as the input, or the benchmark could possibly pre–determine what the instructions should encrypt as.

The first option would let *iBench* write instructions directly to memory, where they would be "decrypted" to the same value. The second option would be a reasonable solution when using a simple direct block mode encryption where the encrypted result of a block of

instructions does not change with address or counter values.

Another direction for iBench would be conversion from assembly to a higher level language. The current version relies on an assembly program targeted for the PowerPC 405 processor. Using assembly eliminates the overhead of function calls and stacks created by compilers, and ensures the circular branching is constructed appropriately.

Research must be given into high level languages and compilers that would allow the branching table to be synthesized such that it is functionally similar to the current configuration. If programmed appropriately, *iBench* would be transportable to addition processor architectures, in addition to the PowerPC 405. Adding self–modifying code as described above into the high-level version would require additional investigation.

Chapter 8

Conclusion

The Virginia Tech Secure Software Project defines an architecture that increases software protection through extensions of standard hardware and software mechanisms. By maintaining standard computer and software use models, these software protections are easily integrated into modern computing environments. This work presents the successful design and implementation of an Encryption Management Unit to support software protections in hardware for the Secure Software Project.

The EMU continues the VT Secure Software goal of remaining compatible with many modern computing architectures. In addition to demonstrating that the EMU can work with a modern architecture, it was designed from the beginning to allow the actual implementation to be extended and directly reused with other architectures. Defining a cohesive set of modules allows the EMU to tailor specific sets of functionality without requiring modification to the remaining system.

Although performance of the EMU is heavily dependent on the cryptographic routine and operational mode, the EMU itself must still maintains minimal latency. This work provides an efficient solution for handling page mapping look–ups and key retrievals. Encrypted status, along with key and ancillary data, are available on the cycle following the transaction

acknowledgment. Although there is a small penalty for the EMU serializing address pipe–lining, it is still possible to invest additional hardware and logic necessary for pipe–lining support, further increasing performance.

The models created in this work provide methods to calculate the absolute and relative effects of the EMU on a application compared with its unencrypted version. Two equations were derived for calculating slow down of an encrypted executable. The equations provide the ability to determine slow down depending on different sets of information known about the application under test. The same solution was also determined in [25], which gives further credence to these models.

A new benchmark was created to test the EMU and to fill a significant void in benchmarking in the secure software field of research. The benchmark emulates the methods used by proven data–side memory latency benchmarks. The instruction–side benchmark developed, *iBench*, achieves less than 0.25% error in comparison to *lat_mem_rd* for external memory fetch latency. With the accuracy of *iBench* verified, the benchmark provides proper measurement of the EMU through latency measurements of several platform configurations.

The benchmark was used to measure actual EMU latency and slow down of protected applications under varying system load. EMU latency was shown to be as expected. An expected small decrease in performance was revealed for unencrypted applications running with the EMU using block and counter modes. This was attributed to disabling address pipe–lining, which can be easily corrected in future EMU revisions. The measured slow down of an encrypted executable matched the mathematical models within the measurement accuracy of 1%.

As the EMU is a component within a larger developing project, the requirements of the EMU will also change. A number of future directions for the EMU and *iBench* were discussed. Possible paths that the Virginia Tech Secure Software Project could pursue and resulting changes to the EMU were identified. These directions included functional improvements, including data protection, and run–time reconfigurable cryptographic modules. A second

set of directions, attributed directly to the EMU, featured performance improvements in future generations such as allowing address pipe–lining.

Bibliography

[1] J. Edmison, "Secure software platform: Acheiving software security via COTS augmentation," June 2005, to be presented at ERSA 2005.

[2] The Trusted Computing Platform Alliance, 2005, www.trustedcomputinggroup.org.

[3] A. Huang, "Keeping secrets in hardware: the Microsoft Xbox (TM) case study," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, Tech Report AIM-2002-008, May 2002, mITLCS-TR-872.

[4] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 160–171.

[5] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, 2003, pp. 178–192.

[6] *Virtex–II Pro and Virtex–II Pro X Platform FPGAs: Complete Data Sheet. Data Sheet 83*, Xilinx, March 2005.

[7] *PowerPC 405 Processor Block Reference Guide. User Guide 18*, Xilinx, August 2004.

[8] *ML310 User Guide. Virtex–II Pro Embedded Development Platform. User Guide 68*, Xilinx, January 2005.

[9] *Platform Studio User Guide. Embedded Development Kit EDK 7.1. User Guide 113*, Xilinx, Febuary 2005.

[10] R. Best, "Preventing software piracy with crypto–microprocessors," in *Proceedings of the IEEE Spring COMPCON 80*, San Fransisco, CA, Febuary 1980, pp. 466–469.

[11] M. G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp," *IEEE Trans. Comput.*, vol. 47, no. 10, pp. 1153–1157, 1998.

[12] Business Software Alliance, 2005, http://www.bsa.org/.

[13] Business Software Alliance and IDC, "Second annual BSA and IDC global software piracy study," May 2005.

[14] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology.* London, UK: Springer-Verlag, 2001, pp. 1–18.

[15] W. Shi, H.-H. S. Lee, C. Lu, and M. Ghosh, "Towards the issues in architectural support for protection of software execution," *SIGARCH Computer Architure News*, vol. 33, no. 1, pp. 6–15, 2005.

[16] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Hardware mechanisms for memory integrity checking," Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, Tech Report MITLCS-TR-872, November 2002.

[17] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *HPCA '03: Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03).* Washington, DC, USA: IEEE Computer Society, 2003, p. 295.

[18] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal Q1*, 2001.

[19] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal on Research and Development*, vol. 46, no. 1, January 2002.

[20] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, and J. B. Rigaud, "Hardware engines for bus encryption: A survey of existing techniques," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 40–45.

[21] IBM, *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*, International Business Machines (IBM) Corporation, 1998.

[22] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach, "A decompression core for PowerPC," *IBM Journal on Research and Development*, vol. 42, no. 6, pp. 807–812, 1998.

[23] *PowerPC 405 Embedded Processor Core User's Manual, Fifth Edition*, IBM, December 2001.

[24] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method," in *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 93–102.

[25] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM Press, 2000, pp. 168–177.

[26] Standard Performance Evaluation Corporation, "SPEC CPU 2000," 2005, www.spec.org.

[27] M. J. Charney, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM Journal of Research and Development*, vol. 41, no. 3, Febuary 1997.

[28] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 24–36, 1995.

[30] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," *Proceedings of the 1996 USENIX Technical Conference*, pp. 279–295, January 1996.

[31] C. Staelin, "lmbench – an extensible micro-benchmark suite," HP Laboratories, Israel, Tech Report HPL-2004-213, December 2004.

[32] U.S. National Institute of Standards and Technology, "Specification for the advanced encryption standard," *Federal Information Processing Standards Publication 197*, November 2001.

[33] V. Fischer and M. Drutarovsky, "Two methods of rijndael implementation in reconfigurable hardware," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2001, pp. 77–92.

[34] M. McLoone and J. V. McCanny, "High performance single-chip fpga rijndael algorithm implementations," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2001, pp. 65–76.

[35] S. Morioka and A. Satoh, "A 10-gbps full-aes crypto design with a twisted bdd s-box architecture," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 7, pp. 686–691, 2004.

[36] *On–Chip Peripheral Bus Architecture Specifications*, 2nd ed., IBM, April 2001.

[37] M. Stein, "Crossing the abyss: asynchronous signals in a synchronous world," *EDN Magazine*, July 2003.

[38] *64-bit Procesor Local Bus Architecture Specifications*, 3rd ed., IBM, May 2001.

[39] *Content-Addressable Memory v5.1. Data Sheet 253*, Xilinx, November 2004.

[40] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande, "Hardware assisted control flow obfuscation for embedded processors," in *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, NY, USA: ACM Press, 2004, pp. 292–302.

[41] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus," in *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM Press, 2004, pp. 72–84.

[42] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 1992, pp. 51–61.

[43] T. Sherwood, S. Sair, and B. Calder, "Predictor–directed stream buffers," in *MICRO 33: Proceedings of the 33rd anuall ACM/IEEE international symposium on Microarchitecture*. New York, NY, USA: ACM Press, 2000, pp. 42–53.

[44] B. Rogers, Y. Solihin, and M. Prvulovic, "Memory predecryption: hiding the latency overhead of memory encryption," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 27–33, 2005.

Appendix A

Page Table Control Registers

A.1 Page Base Address Register

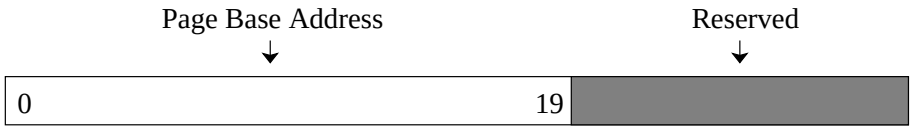

Figure A.1: Page Table Control: Page Base Address Register

Table A.1: Page Base Address Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 19	Page Address	Read/Write	0x00000	Base physical memory address of the page to be associated with the page–key and page–ancillary data pairs
20 - 31				Reserved

A.2 Key Index Register

Figure A.2: Page Table Control: Key Index Register

Table A.2: Key Index Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 25				Reserved
26 - 31	Key Index	Read/Write	0x00	Pointer into key table associated with a page by the page–key pair

A.3 Page Index Register

Figure A.3: Page Table Control: Page Index Register

Table A.3: Page Index Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 25				Reserved
26 - 31	Page Index	Read/Write	0x00	Table location to load page–key and page–ancillary data pairs into. Writing to this register loads the page–key and page–ancillary information into the tables.

A.4 Ancillary Data 0 Register

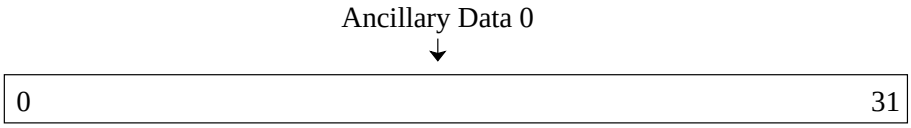

Figure A.4: Page Table Control: Ancillary Data 0 Register

Table A.4: Ancillary Data 0 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (96-127) of four-word data set

A.5 Ancillary Data 1 Register

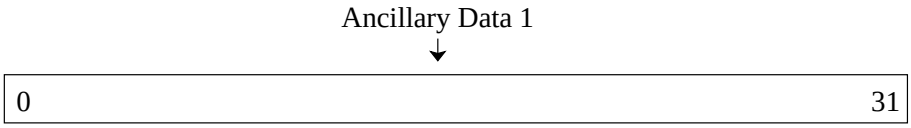

Figure A.5: Page Table Control: Ancillary Data 1 Register

Table A.5: Ancillary Data 1 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (64-95) of four-word data set

A.6 Ancillary Data 2 Register

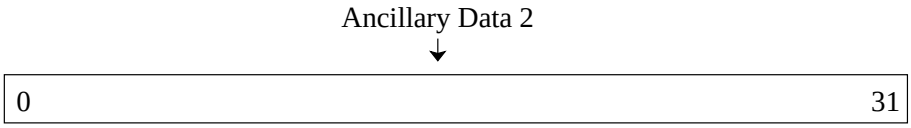

Figure A.6: Page Table Control: Ancillary Data 2 Register

Table A.6: Ancillary Data 2 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (32-63) of four-word data set

A.7 Ancillary Data 3 Register

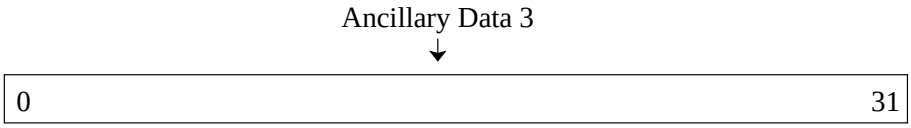

Figure A.7: Page Table Control: Ancillary Data 3 Register

Table A.7: Ancillary Data 3 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (0-31) of four-word data set

Appendix B

Key Table Control Registers

B.1 Key Index Register

Figure B.1: Key Table Control: Key Index Register

Table B.1: Key Index Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 25				Reserved
26 - 31	Key Index	Read/Write	0x00	Pointer to where key will be inserted into key table

B.2 Key Word 0 Register

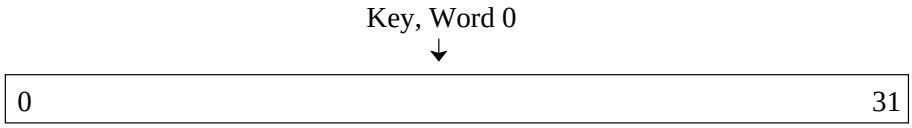

Figure B.2: Key Table Control: Key Word 0 Register

Table B.2: Key Word 0 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (96-127) of four-word key set

B.3 Key Word 1 Register

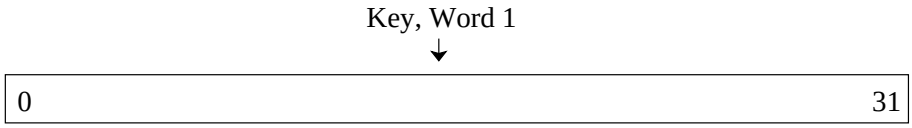

Figure B.3: Key Table Control: Key Word 1 Register

Table B.3: Key Word 1 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (64-95) of four-word key set

B.4 Key Word 2 Register

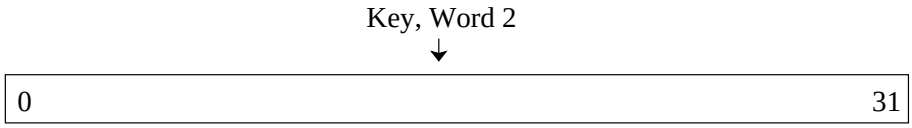

Figure B.4: Key Table Control: Key Word 2 Register

Table B.4: Key Word 2 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (32-63) of four-word key set

B.5 Key Word 3 Register

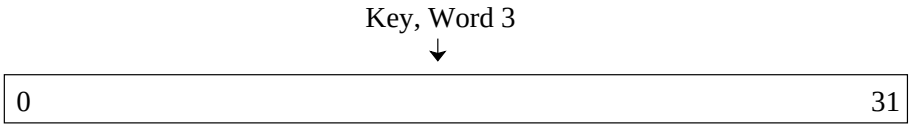

Figure B.5: Key Table Control: Key Word 3 Register

Table B.5: Key Word 3 Register Bit Definitions

Bit(s)	Name	Access	Reset Value	Description
0 - 31	Data	Read/Write	0x00000000	Bits (0-31) of four-word key set

Appendix C

Benchmark Results

C.1 LMBench

256–bit stride, Base platform (no EMU).

```
0.00391 10.310
0.00586 10.311
0.00781 10.311
0.00977 10.312
0.01172 10.310
0.01367 10.313
0.01562 10.311
0.01758 75.063
0.01953 126.741
0.02148 133.986
0.02344 171.629
0.02539 203.001
0.02734 176.270
0.02930 203.353
0.03125 203.293
0.03516 204.635
0.03906 204.636
0.04297 204.642
0.04688 204.678
0.05078 204.676
0.05469 204.683
0.05859 204.669
0.06250 204.699
```

```
0.07031 205.373
0.07812 205.378
0.08594 205.422
0.09375 205.470
0.10156 205.492
0.10938 205.535
0.11719 205.564
0.12500 264.510
0.14062 263.897
0.15625 264.078
0.17188 263.679
0.18750 263.648
0.20312 263.755
0.21875 263.712
0.23438 263.474
0.25000 263.557
0.28125 263.545
0.31250 263.519
0.34375 263.631
0.37500 263.494
0.40625 263.800
0.43750 263.784
0.46875 263.627
0.50000 263.653
0.56250 263.859
0.62500 264.384
0.68750 264.475
0.75000 264.846
0.81250 264.740
0.87500 265.189
0.93750 264.916
1.00000 264.890
1.12500 265.317
1.25000 265.674
1.37500 265.792
1.50000 266.320
1.62500 266.384
1.75000 266.591
1.87500 266.800
2.00000 266.910
2.25000 267.402
2.50000 267.629
```

```
2.75000 267.793
3.00000 268.022
3.25000 268.105
3.50000 268.149
3.75000 268.175
4.00000 268.285
4.50000 268.378
5.00000 268.507
5.50000 268.470
6.00000 268.515
6.50000 268.511
7.00000 268.488
7.50000 268.451
8.00000 268.459
```

C.2 iBench

C.2.1 Complete Latency Measurement

256–bit stride, Base platform (no EMU).

```
256, 0.00391, 409600, 60572000.000, 11744.051, 8.697
256, 0.00586, 273066, 65972000.000, 7207.954, 9.663
256, 0.00781, 204800, 48808000.000, 5172.975, 7.221
256, 0.00977, 163840, 63238000.000, 4026.532, 9.413
256, 0.01172, 136533, 62553000.000, 3293.289, 9.350
256, 0.01367, 117028, 62147000.000, 2784.762, 9.316
256, 0.01562, 102400, 61874000.000, 2411.725, 9.296
256, 0.01758, 91022, 490510000.000, 2126.484, 73.820
256, 0.01953, 81920, 839443000.000, 1901.418, 126.507
256, 0.02148, 74472, 1122672000.000, 1719.284, 169.383
256, 0.02344, 68266, 1359136000.000, 1568.950, 205.251
256, 0.02539, 63015, 1356806000.000, 1442.757, 205.061
256, 0.02734, 58514, 1356077000.000, 1335.316, 205.091
256, 0.02930, 54613, 1354939000.000, 1242.742, 205.040
256, 0.03125, 51200, 1354213000.000, 1162.172, 205.034
256, 0.03516, 45511, 1352585000.000, 1028.721, 204.965
256, 0.03906, 40960, 1350078000.000, 922.747, 204.726
256, 0.04297, 37236, 1350206000.000, 836.534, 204.863
256, 0.04688, 34133, 1349193000.000, 765.057, 204.806
256, 0.05078, 31507, 1348997000.000, 704.811, 204.860
256, 0.05469, 29257, 1348843000.000, 653.392, 204.903
256, 0.05859, 27306, 1347604000.000, 608.921, 204.780
256, 0.06250, 25600, 1347405000.000, 570.163, 204.798
256, 0.07031, 22755, 1350423000.000, 505.708, 205.350
256, 0.07812, 20480, 1350677000.000, 454.383, 205.455
256, 0.08594, 18618, 1350053000.000, 412.490, 205.420
256, 0.09375, 17066, 1350503000.000, 377.652, 205.543
256, 0.10156, 15753, 1349935000.000, 348.247, 205.501
256, 0.10938, 14628, 1350253000.000, 323.107, 205.581
256, 0.11719, 13653, 1350376000.000, 301.354, 205.628
256, 0.12500, 12800, 1350505000.000, 282.351, 205.669
256, 0.14062, 11377, 1700981000.000, 250.675, 259.117
256, 0.15625, 10240, 1700687000.000, 225.444, 259.099
256, 0.17188, 9309, 1701432000.000, 204.801, 259.252
256, 0.18750, 8533, 1700110000.000, 187.612, 259.089
```

```
256, 0.20312, 7876, 1700234000.000, 173.068, 259.154
256, 0.21875, 7314, 1699794000.000, 160.662, 259.089
256, 0.23438, 6826, 1699231000.000, 149.878, 259.038
256, 0.25000, 6400, 1699784000.000, 140.493, 259.113
256, 0.28125, 5688, 1699019000.000, 124.776, 259.065
256, 0.31250, 5120, 1699587000.000, 112.285, 259.134
256, 0.34375, 4654, 1702834000.000, 102.017, 259.678
256, 0.37500, 4266, 1699969000.000, 93.481, 259.266
256, 0.40625, 3938, 1699477000.000, 86.275, 259.194
256, 0.43750, 3657, 1699861000.000, 80.108, 259.244
256, 0.46875, 3413, 1701912000.000, 74.745, 259.581
256, 0.50000, 3200, 1702227000.000, 70.076, 259.612
256, 0.56250, 2844, 1702655000.000, 62.253, 259.732
256, 0.62500, 2560, 1699464000.000, 56.033, 259.216
256, 0.68750, 2327, 1703464000.000, 50.918, 259.866
256, 0.75000, 2133, 1704587000.000, 46.665, 260.055
256, 0.81250, 1969, 1705902000.000, 43.073, 260.252
256, 0.87500, 1828, 1708058000.000, 39.976, 260.638
256, 0.93750, 1706, 1711259000.000, 37.302, 261.151
256, 1.00000, 1600, 1714249000.000, 34.995, 261.510
256, 1.12500, 1422, 1710602000.000, 31.093, 261.001
256, 1.25000, 1280, 1711669000.000, 27.989, 261.129
256, 1.37500, 1163, 1710872000.000, 25.415, 261.155
256, 1.50000, 1066, 1711140000.000, 23.292, 261.220
256, 1.62500, 984, 1717667000.000, 21.499, 262.220
256, 1.75000, 914, 1720600000.000, 19.974, 262.588
256, 1.87500, 853, 1723281000.000, 18.639, 263.020
256, 2.00000, 800, 1726781000.000, 17.487, 263.454
256, 2.25000, 711, 1724578000.000, 15.538, 263.162
256, 2.50000, 640, 1732793000.000, 13.988, 264.377
256, 2.75000, 581, 1731713000.000, 12.680, 264.587
256, 3.00000, 533, 1736049000.000, 11.641, 265.044
256, 3.25000, 492, 1736926000.000, 10.745, 265.180
256, 3.50000, 457, 1737473000.000, 9.984, 265.182
256, 3.75000, 426, 1735619000.000, 9.295, 265.232
256, 4.00000, 400, 1739295000.000, 8.741, 265.379
256, 4.50000, 355, 1735783000.000, 7.745, 265.260
256, 5.00000, 320, 1738163000.000, 6.992, 265.210
256, 5.50000, 290, 1733629000.000, 6.317, 265.348
256, 6.00000, 266, 1734165000.000, 5.798, 265.265
256, 6.50000, 246, 1737340000.000, 5.372, 265.253
256, 7.00000, 228, 1737288000.000, 4.969, 265.744
```

```
256, 7.50000, 213, 1735714000.000, 4.647, 265.255
256, 8.00000, 200, 1738310000.000, 4.370, 265.237
```

C.2.2 EMU Memory Fetch Latency

Table C.1: Base Profile Memory Fetch Latency

Size (kB)	1x	2x	4x
$2.40E+01$	$2.05E+02$	$4.21E+02$	$8.62E+02$
$2.60E+01$	$2.05E+02$	$4.10E+02$	$8.53E+02$
$2.80E+01$	$2.05E+02$	$4.10E+02$	$8.53E+02$
$3.00E+01$	$2.05E+02$	$4.22E+02$	$8.24E+02$
$3.20E+01$	$2.05E+02$	$4.27E+02$	$8.31E+02$
$3.60E+01$	$2.05E+02$	$4.10E+02$	$8.60E+02$
$4.00E+01$	$2.05E+02$	$4.22E+02$	$8.56E+02$
$4.40E+01$	$2.05E+02$	$4.21E+02$	$8.43E+02$
$4.80E+01$	$2.05E+02$	$4.11E+02$	$8.43E+02$
$5.20E+01$	$2.05E+02$	$4.10E+02$	$8.33E+02$
$5.60E+01$	$2.05E+02$	$4.10E+02$	$8.43E+02$
$6.00E+01$	$2.05E+02$	$4.10E+02$	$8.80E+02$
$6.40E+01$	$2.05E+02$	$4.22E+02$	$8.49E+02$
$7.20E+01$	$2.05E+02$	$4.16E+02$	$8.33E+02$
$8.00E+01$	$2.06E+02$	$4.13E+02$	$8.35E+02$
$8.80E+01$	$2.06E+02$	$4.24E+02$	$8.43E+02$
$9.60E+01$	$2.06E+02$	$4.23E+02$	$8.35E+02$
$1.04E+02$	$2.06E+02$	$4.12E+02$	$8.42E+02$
$1.12E+02$	$2.06E+02$	$4.12E+02$	$8.40E+02$
$1.20E+02$	$2.06E+02$	$4.12E+02$	$8.72E+02$
Avg (ns)	$2.05E+02$	$4.16E+02$	$8.46E+02$

Table C.2: Block Mode Profile Memory Fetch Latency

Size (kB)	Unenc 1x	Unenc 2x	Unenc 4x	Enc 1x	Enc 2x	Enc 4x
$2.40E+01$	$2.16E+02$	$4.43E+02$	$8.84E+02$	$3.75E+02$	$7.59E+02$	$1.54E+03$
$2.60E+01$	$2.16E+02$	$4.32E+02$	$8.85E+02$	$3.74E+02$	$7.61E+02$	$1.54E+03$
$2.80E+01$	$2.15E+02$	$4.38E+02$	$8.60E+02$	$3.75E+02$	$7.61E+02$	$1.53E+03$
$3.00E+01$	$2.15E+02$	$4.20E+02$	$8.63E+02$	$3.75E+02$	$7.49E+02$	$1.52E+03$
$3.20E+01$	$2.16E+02$	$4.22E+02$	$8.45E+02$	$3.75E+02$	$7.61E+02$	$1.54E+03$
$3.60E+01$	$2.16E+02$	$4.43E+02$	$8.66E+02$	$3.75E+02$	$7.61E+02$	$1.53E+03$
$4.00E+01$	$2.15E+02$	$4.32E+02$	$8.67E+02$	$3.74E+02$	$7.63E+02$	$1.50E+03$
$4.40E+01$	$2.16E+02$	$4.32E+02$	$8.64E+02$	$3.74E+02$	$7.61E+02$	$1.52E+03$
$4.80E+01$	$2.16E+02$	$4.44E+02$	$8.73E+02$	$3.75E+02$	$7.49E+02$	$1.51E+03$
$5.20E+01$	$2.16E+02$	$4.33E+02$	$8.51E+02$	$3.75E+02$	$7.60E+02$	$1.50E+03$
$5.60E+01$	$2.15E+02$	$4.32E+02$	$8.51E+02$	$3.75E+02$	$7.49E+02$	$1.51E+03$
$6.00E+01$	$2.15E+02$	$4.20E+02$	$8.54E+02$	$3.75E+02$	$7.50E+02$	$1.50E+03$
$6.40E+01$	$2.16E+02$	$4.22E+02$	$8.51E+02$	$3.75E+02$	$7.50E+02$	$1.50E+03$
$7.20E+01$	$2.16E+02$	$4.27E+02$	$8.61E+02$	$3.74E+02$	$7.49E+02$	$1.50E+03$
$8.00E+01$	$2.17E+02$	$4.34E+02$	$8.65E+02$	$3.77E+02$	$7.65E+02$	$1.54E+03$
$8.80E+01$	$2.17E+02$	$4.32E+02$	$8.72E+02$	$3.77E+02$	$7.65E+02$	$1.54E+03$
$9.60E+01$	$2.17E+02$	$4.46E+02$	$8.66E+02$	$3.77E+02$	$7.66E+02$	$1.54E+03$
$1.04E+02$	$2.17E+02$	$4.34E+02$	$8.54E+02$	$3.77E+02$	$7.65E+02$	$1.55E+03$
$1.12E+02$	$2.17E+02$	$4.23E+02$	$8.53E+02$	$3.77E+02$	$7.66E+02$	$1.58E+03$
$1.20E+02$	$2.17E+02$	$4.22E+02$	$8.86E+02$	$3.77E+02$	$7.66E+02$	$1.56E+03$
$Avg(ns)$	$2.16E+02$	$4.32E+02$	$8.64E+02$	$3.75E+02$	$7.59E+02$	$1.53E+03$
Slow Down	1.05	1.04	1.02	1.83	1.82	1.81

Table C.3: Counter Mode Profile Memory Fetch Latency

Size (kB)	Unenc 1x	Unenc 2x	Unenc 4x	Enc 1x	Enc 2x	Enc 4x
$2.40E+01$	$2.25E+02$	$4.57E+02$	$9.36E+02$	$2.36E+02$	$4.75E+02$	$9.64E+02$
$2.60E+01$	$2.25E+02$	$4.50E+02$	$9.09E+02$	$2.36E+02$	$4.74E+02$	$9.62E+02$
$2.80E+01$	$2.25E+02$	$4.53E+02$	$9.03E+02$	$2.36E+02$	$4.75E+02$	$9.73E+02$
$3.00E+01$	$2.25E+02$	$4.52E+02$	$9.14E+02$	$2.36E+02$	$4.74E+02$	$9.58E+02$
$3.20E+01$	$2.25E+02$	$4.51E+02$	$9.05E+02$	$2.36E+02$	$4.74E+02$	$9.48E+02$
$3.60E+01$	$2.25E+02$	$4.52E+02$	$9.10E+02$	$2.36E+02$	$4.74E+02$	$9.54E+02$
$4.00E+01$	$2.25E+02$	$4.57E+02$	$9.10E+02$	$2.36E+02$	$4.71E+02$	$9.58E+02$
$4.40E+01$	$2.25E+02$	$4.47E+02$	$9.14E+02$	$2.36E+02$	$4.74E+02$	$9.48E+02$
$4.80E+01$	$2.25E+02$	$4.42E+02$	$9.10E+02$	$2.36E+02$	$4.74E+02$	$9.46E+02$
$5.20E+01$	$2.25E+02$	$4.53E+02$	$9.01E+02$	$2.37E+02$	$4.74E+02$	$9.47E+02$
$5.60E+01$	$2.25E+02$	$4.59E+02$	$8.98E+02$	$2.36E+02$	$4.73E+02$	$9.59E+02$
$6.00E+01$	$2.25E+02$	$4.46E+02$	$9.43E+02$	$2.36E+02$	$4.73E+02$	$9.70E+02$
$6.40E+01$	$2.25E+02$	$4.59E+02$	$9.08E+02$	$2.36E+02$	$4.74E+02$	$9.50E+02$
$7.20E+01$	$2.25E+02$	$4.54E+02$	$9.13E+02$	$2.36E+02$	$4.75E+02$	$9.50E+02$
$8.00E+01$	$2.27E+02$	$4.55E+02$	$9.11E+02$	$2.38E+02$	$4.76E+02$	$9.68E+02$
$8.80E+01$	$2.27E+02$	$4.54E+02$	$9.09E+02$	$2.38E+02$	$4.74E+02$	$9.62E+02$
$9.60E+01$	$2.27E+02$	$4.55E+02$	$9.09E+02$	$2.38E+02$	$4.76E+02$	$9.49E+02$
$1.04E+02$	$2.27E+02$	$4.60E+02$	$9.25E+02$	$2.38E+02$	$4.77E+02$	$9.55E+02$
$1.12E+02$	$2.27E+02$	$4.56E+02$	$9.30E+02$	$2.38E+02$	$4.88E+02$	$9.53E+02$
$1.20E+02$	$2.27E+02$	$4.67E+02$	$9.36E+02$	$2.38E+02$	$4.77E+02$	$9.53E+02$
$Avg(ns)$	$2.26E+02$	$4.54E+02$	$9.15E+02$	$2.37E+02$	$4.75E+02$	$9.56E+02$
Slow Down	1.10	1.09	1.08	1.15	1.14	1.13

C.2.3 EMU Execution Time

Table C.4: Base Profile Execution Time

Size (kB)	1x	2x	4x
$2.40E+01$	$1.36E+09$	$2.79E+09$	$5.71E+09$
$2.60E+01$	$1.36E+09$	$2.71E+09$	$5.64E+09$
$2.80E+01$	$1.36E+09$	$2.71E+09$	$5.64E+09$
$3.00E+01$	$1.35E+09$	$2.79E+09$	$5.45E+09$
$3.20E+01$	$1.35E+09$	$2.82E+09$	$5.49E+09$
$3.60E+01$	$1.35E+09$	$2.71E+09$	$5.67E+09$
$4.00E+01$	$1.35E+09$	$2.78E+09$	$5.65E+09$
$4.40E+01$	$1.35E+09$	$2.78E+09$	$5.56E+09$
$4.80E+01$	$1.35E+09$	$2.71E+09$	$5.55E+09$
$5.20E+01$	$1.35E+09$	$2.70E+09$	$5.48E+09$
$5.60E+01$	$1.35E+09$	$2.70E+09$	$5.55E+09$
$6.00E+01$	$1.35E+09$	$2.70E+09$	$5.79E+09$
$6.40E+01$	$1.35E+09$	$2.77E+09$	$5.59E+09$
$7.20E+01$	$1.35E+09$	$2.74E+09$	$5.48E+09$
$8.00E+01$	$1.35E+09$	$2.71E+09$	$5.49E+09$
$8.80E+01$	$1.35E+09$	$2.78E+09$	$5.54E+09$
$9.60E+01$	$1.35E+09$	$2.78E+09$	$5.48E+09$
$1.04E+02$	$1.35E+09$	$2.71E+09$	$5.53E+09$
$1.12E+02$	$1.35E+09$	$2.71E+09$	$5.52E+09$
$1.20E+02$	$1.35E+09$	$2.70E+09$	$5.72E+09$
$Avg(ns)$	$1.35E+09$	$2.74E+09$	$5.58E+09$

Table C.5: Block Mode Profile Execution Time

Size (kB)	Unenc 1x	Unenc 2x	Unenc 4x	Enc 1x	Enc 2x	Enc 4x
$2.40E+01$	$1.43E+09$	$2.93E+09$	$5.85E+09$	$2.48E+09$	$5.03E+09$	$1.02E+10$
$2.60E+01$	$1.43E+09$	$2.86E+09$	$5.85E+09$	$2.48E+09$	$5.04E+09$	$1.02E+10$
$2.80E+01$	$1.42E+09$	$2.89E+09$	$5.69E+09$	$2.48E+09$	$5.03E+09$	$1.01E+10$
$3.00E+01$	$1.42E+09$	$2.78E+09$	$5.70E+09$	$2.47E+09$	$4.95E+09$	$1.01E+10$
$3.20E+01$	$1.43E+09$	$2.79E+09$	$5.58E+09$	$2.47E+09$	$5.03E+09$	$1.02E+10$
$3.60E+01$	$1.43E+09$	$2.93E+09$	$5.72E+09$	$2.47E+09$	$5.02E+09$	$1.01E+10$
$4.00E+01$	$1.42E+09$	$2.85E+09$	$5.72E+09$	$2.47E+09$	$5.03E+09$	$9.91E+09$
$4.40E+01$	$1.42E+09$	$2.85E+09$	$5.69E+09$	$2.47E+09$	$5.02E+09$	$1.00E+10$
$4.80E+01$	$1.42E+09$	$2.92E+09$	$5.75E+09$	$2.47E+09$	$4.93E+09$	$9.94E+09$
$5.20E+01$	$1.42E+09$	$2.85E+09$	$5.61E+09$	$2.47E+09$	$5.01E+09$	$9.88E+09$
$5.60E+01$	$1.42E+09$	$2.84E+09$	$5.60E+09$	$2.47E+09$	$4.93E+09$	$9.97E+09$
$6.00E+01$	$1.42E+09$	$2.76E+09$	$5.62E+09$	$2.47E+09$	$4.94E+09$	$9.87E+09$
$6.40E+01$	$1.42E+09$	$2.78E+09$	$5.60E+09$	$2.46E+09$	$4.93E+09$	$9.87E+09$
$7.20E+01$	$1.42E+09$	$2.81E+09$	$5.66E+09$	$2.46E+09$	$4.93E+09$	$9.85E+09$
$8.00E+01$	$1.42E+09$	$2.85E+09$	$5.69E+09$	$2.48E+09$	$5.03E+09$	$1.01E+10$
$8.80E+01$	$1.43E+09$	$2.84E+09$	$5.73E+09$	$2.47E+09$	$5.03E+09$	$1.01E+10$
$9.60E+01$	$1.43E+09$	$2.93E+09$	$5.69E+09$	$2.48E+09$	$5.03E+09$	$1.01E+10$
$1.04E+02$	$1.43E+09$	$2.85E+09$	$5.61E+09$	$2.47E+09$	$5.03E+09$	$1.02E+10$
$1.12E+02$	$1.42E+09$	$2.78E+09$	$5.60E+09$	$2.48E+09$	$5.03E+09$	$1.04E+10$
$1.20E+02$	$1.42E+09$	$2.77E+09$	$5.82E+09$	$2.48E+09$	$5.03E+09$	$1.03E+10$
$Avg(ns)$	$1.42E+09$	$2.84E+09$	$5.69E+09$	$2.47E+09$	$5.00E+09$	$1.01E+10$
Slow down	1.05	1.04	1.02	1.83	1.82	1.81

Table C.6: Counter Mode Profile Execution Time

Size (kB)	Unenc 1x	Unenc 2x	Unenc 4x	Enc 1x	Enc 2x	Enc 4x
$2.40E+01$	$1.49E+09$	$3.03E+09$	$6.20E+09$	$1.56E+09$	$3.15E+09$	$6.38E+09$
$2.60E+01$	$1.49E+09$	$2.98E+09$	$6.01E+09$	$1.56E+09$	$3.14E+09$	$6.37E+09$
$2.80E+01$	$1.49E+09$	$2.99E+09$	$5.97E+09$	$1.56E+09$	$3.14E+09$	$6.44E+09$
$3.00E+01$	$1.49E+09$	$2.99E+09$	$6.04E+09$	$1.56E+09$	$3.13E+09$	$6.33E+09$
$3.20E+01$	$1.49E+09$	$2.98E+09$	$5.98E+09$	$1.56E+09$	$3.13E+09$	$6.26E+09$
$3.60E+01$	$1.49E+09$	$2.98E+09$	$6.01E+09$	$1.56E+09$	$3.13E+09$	$6.29E+09$
$4.00E+01$	$1.49E+09$	$3.02E+09$	$6.00E+09$	$1.56E+09$	$3.10E+09$	$6.32E+09$
$4.40E+01$	$1.48E+09$	$2.95E+09$	$6.02E+09$	$1.56E+09$	$3.12E+09$	$6.25E+09$
$4.80E+01$	$1.48E+09$	$2.91E+09$	$5.99E+09$	$1.56E+09$	$3.12E+09$	$6.23E+09$
$5.20E+01$	$1.48E+09$	$2.98E+09$	$5.94E+09$	$1.56E+09$	$3.12E+09$	$6.24E+09$
$5.60E+01$	$1.48E+09$	$3.02E+09$	$5.91E+09$	$1.56E+09$	$3.11E+09$	$6.31E+09$
$6.00E+01$	$1.48E+09$	$2.93E+09$	$6.20E+09$	$1.55E+09$	$3.11E+09$	$6.39E+09$
$6.40E+01$	$1.48E+09$	$3.02E+09$	$5.98E+09$	$1.55E+09$	$3.12E+09$	$6.25E+09$
$7.20E+01$	$1.48E+09$	$2.98E+09$	$6.01E+09$	$1.55E+09$	$3.12E+09$	$6.24E+09$
$8.00E+01$	$1.49E+09$	$2.99E+09$	$5.99E+09$	$1.56E+09$	$3.13E+09$	$6.36E+09$
$8.80E+01$	$1.49E+09$	$2.98E+09$	$5.98E+09$	$1.56E+09$	$3.11E+09$	$6.32E+09$
$9.60E+01$	$1.49E+09$	$2.99E+09$	$5.97E+09$	$1.56E+09$	$3.13E+09$	$6.24E+09$
$1.04E+02$	$1.49E+09$	$3.02E+09$	$6.08E+09$	$1.56E+09$	$3.13E+09$	$6.27E+09$
$1.12E+02$	$1.49E+09$	$2.99E+09$	$6.11E+09$	$1.56E+09$	$3.21E+09$	$6.26E+09$
$1.20E+02$	$1.49E+09$	$3.07E+09$	$6.15E+09$	$1.56E+09$	$3.13E+09$	$6.26E+09$
$Avg(ns)$	$1.49E+09$	$2.99E+09$	$6.03E+09$	$1.56E+09$	$3.13E+09$	$6.30E+09$
Slow down	1.10	1.09	1.08	1.15	1.14	1.13

Appendix D

Source Listings

D.1 Encryption Management Unit HDL

D.1.1 Bridge.vhd

```
-------------------------------------------------------------------------------
-- Title      : Encrypted Memory Bus Bridge
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : Bridge.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-03-21
-- Last update: 2005/07/19
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: High level module to bridge to interface between lower level
--              PPC405 cache and upper level memory bus, including any
--              intermediate modules such as decryption units, caches, etc.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-05-05  amahar  Added "decryption" module
-- 2005-05-02  amahar  Switched from the old Req/Ack/Addr/Data to pseudo-PLB
-- 2005-04-11  amahar  Removed logic from this module and instantiated
--                     the CPU/Memory interface modules
-- 2005-03-25  amahar  Switched from complete passthrough to selective XOR
--                     decryption
-- 2005-03-21  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

library plb_emu_v1_00_a;
```

```
use plb_emu_v1_00_a.all;

entity Bridge is

------------------------------------------------------------------------------
-- Entity: generics
------------------------------------------------------------------------------
  generic (
    C_CPUBUS_NUM_MASTERS : integer := 1;
    C_CPUBUS_MID_WIDTH   : integer := 1;
    C_CPUBUS_AWIDTH      : integer := 32;
    C_CPUBUS_DWIDTH      : integer := 64;
    C_MEMBUS_AWIDTH      : integer := 32;
    C_MEMBUS_DWIDTH      : integer := 64;
    C_PAGE_AWIDTH        : integer := 20;
    C_KEYSIZE            : integer := 128;
    C_ANCILLARY_DWIDTH   : integer := 128);

------------------------------------------------------------------------------
-- Entity: ports
------------------------------------------------------------------------------
  port (
    MemClk                    : in  std_logic;
    MemReset                  : in  std_logic;
-- Master side PLB bus signals
    -- Request Qualifiers
    iMemBus_PLB_MAddrAck      : in  std_logic;
    iMemBus_PLB_MBusy         : in  std_logic;
    iMemBus_PLB_MErr          : in  std_logic;
    iMemBus_PLB_MRearbitrate  : in  std_logic;
    iMemBus_PLB_MSSize        : in  std_logic_vector(0 to 1);
    oMemBus_M_abort           : out std_logic;
    oMemBus_M_ABus            : out std_logic_vector(0 to C_MEMBUS_AWIDTH-1);
    oMemBus_M_BE              : out std_logic_vector(0 to ((C_MEMBUS_DWIDTH/8))-1);
    oMemBus_M_busLock         : out std_logic;
    oMemBus_M_compress        : out std_logic;
    oMemBus_M_guarded         : out std_logic;
    oMemBus_M_lockErr         : out std_logic;
    oMemBus_M_MSize           : out std_logic_vector(0 to 1);
    oMemBus_M_ordered         : out std_logic;
    oMemBus_M_priority        : out std_logic_vector(0 to 1);
    oMemBus_M_request         : out std_logic;
    oMemBus_M_RNW             : out std_logic;
    oMemBus_M_size            : out std_logic_vector(0 to 3);
    oMemBus_M_type            : out std_logic_vector(0 to 2);
    iMemBus_PLB_SMBusy        : in  std_logic;
    iMemBus_PLB_SMErr         : in  std_logic;
    -- Write Data Bus
    oMemBus_M_wrBurst         : out std_logic;
    oMemBus_M_wrDBus          : out std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
    iMemBus_PLB_MWrBTerm      : in  std_logic;
    iMemBus_PLB_MWrDAck       : in  std_logic;
    -- Read Data Bus
    oMemBus_M_rdBurst         : out std_logic;
    iMemBus_PLB_MRdBTerm      : in  std_logic;
    iMemBus_PLB_MRdDAck       : in  std_logic;
    iMemBus_PLB_MRdDBus       : in  std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
    iMemBus_PLB_MRdWdAddr     : in  std_logic_vector(0 to 3);
    -- Unused Master side signals
    --iMemBus_PLB_pendReq      : in  std_logic;
    --iMemBus_PLB_pendPri      : in  std_logic_vector(0 to 1);
    --iMemBus_PLB_reqPri       : in  std_logic_vector(0 to 1);

-- Slave side PLB bus signals
```

```
    -- Request Qualifiers
    oTransBus_Sl_addrAck      : out std_logic;
    oTransBus_Sl_MBusy        : out std_logic_vector(0 to C_CPUBUS_NUM_MASTERS-1);
    oTransBus_Sl_MErr         : out std_logic_vector(0 to C_CPUBUS_NUM_MASTERS-1);
    oTransBus_Sl_rearbitrate  : out std_logic;
    oTransBus_Sl_SSize        : out std_logic_vector(0 to 1);
    iTransBus_PLB_abort       : in  std_logic;
    iTransBus_PLB_ABus        : in  std_logic_vector(0 to C_CPUBUS_AWIDTH-1);
    iTransBus_PLB_BE          : in  std_logic_vector(0 to (C_CPUBUS_DWIDTH/8)-1);
    iTransBus_PLB_busLock     : in  std_logic;
    iTransBus_PLB_compress    : in  std_logic;
    iTransBus_PLB_guarded     : in  std_logic;
    iTransBus_PLB_lockErr     : in  std_logic;
    iTransBus_PLB_MSize       : in  std_logic_vector(0 to 1);
    iTransBus_PLB_ordered     : in  std_logic;
    iTransBus_PLB_pendPri     : in  std_logic_vector(0 to 1);
    iTransBus_PLB_pendReq     : in  std_logic;
    iTransBus_PLB_RNW         : in  std_logic;
    iTransBus_PLB_size        : in  std_logic_vector(0 to 3);
    iTransBus_PLB_type        : in  std_logic_vector(0 to 2);
    oTransBus_Sl_rdComp       : out std_logic;
    oTransBus_Sl_wrComp       : out std_logic;
    -- Write Data Bus
    iTransBus_PLB_wrBurst     : in  std_logic;
    iTransBus_PLB_wrDBus      : in  std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
    oTransBus_Sl_wrBTerm      : out std_logic;
    oTransBus_Sl_wrDAck       : out std_logic;
    -- Read Data Bus
    iTransBus_PLB_rdBurst     : in  std_logic;
    oTransBus_Sl_rdBTerm      : out std_logic;
    oTransBus_Sl_rdDAck       : out std_logic;
    oTransBus_Sl_rdDBus       : out std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
    oTransBus_Sl_rdWdAddr     : out std_logic_vector(0 to 3);
    -- Unused Slave side signals
    --iTransBus_PLB_masterID   : in  std_logic_vector(0 to C_PLB_MID_WIDTH-1));
    --iTransBus_PLB_PAValid    : out std_logic;
    --iTransBus_PLB_SAValid    : out std_logic;
    --iTransBus_PLB_reqPri     : out std_logic_vector(0 to 1);
    --iTransBus_PLB_rdPrim     : out std_logic;
    --iTransBus_PLB_wrPrim     : out std_logic;
    --oTransBus_Sl_wait        : out std_logic;

-- Table Search Interface
    oTableReq             : out std_logic;
    iTableSearchDone      : in  std_logic;
    oPageAddress          : out std_logic_vector(0 to C_PAGE_AWIDTH-1);
    iPageEncrypted        : in  std_logic;
    iPageKey              : in  std_logic_vector(0 to C_KEYSIZE-1);
    iPageAncillary        : in  std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
    oPerformance_Fetch    : out std_logic;
    oPerformance_EncFetch : out std_logic);

end Bridge;

-------------------------------------------------------------------------------
-- Architecture Section
-------------------------------------------------------------------------------

architecture arch of Bridge is

  ---------------------------------------------------------------------------
  -- Internal signal declarations
  ---------------------------------------------------------------------------
  signal s1_M_ABus          : std_logic_vector(0 to C_CPUBUS_AWIDTH-1);
  signal s1_M_request       : std_logic;
```

```
  signal s1_M_size         : std_logic_vector(0 to 1);
  signal s1_M_abort        : std_logic;
  signal s1_M_BE           : std_logic_vector(0 to (C_CPUBUS_DWIDTH/8)-1);
  signal s1_PLB_MAddrAck   : std_logic;
  signal s1_PLB_MBusy      : std_logic;
  signal s1_PLB_MRdDAck    : std_logic;
  signal s1_PLB_MRdDBus    : std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
  signal s1_PLB_MRdWdAddr  : std_logic_vector(0 to 3);

  signal s2_M_ABus         : std_logic_vector(0 to C_MEMBUS_AWIDTH-1);
  signal s2_M_request      : std_logic;
  signal s2_M_size         : std_logic_vector(0 to 1);
  signal s2_M_abort        : std_logic;
  signal s2_M_BE           : std_logic_vector(0 to (C_CPUBUS_DWIDTH/8)-1);
  signal s2_PLB_MAddrAck   : std_logic;
  signal s2_PLB_MBusy      : std_logic;
  signal s2_PLB_MRdDAck    : std_logic;
  signal s2_PLB_MRdDBus    : std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
  signal s2_PLB_MRdWdAddr  : std_logic_vector(0 to 3);

---------------------------------------------------------------------------
-- Begin architecture
---------------------------------------------------------------------------
begin

  ---------------------------------------------------------------------------
  -- Sanity check
  ---------------------------------------------------------------------------
  assert (C_MEMBUS_AWIDTH = C_CPUBUS_AWIDTH) report "ERROR: CPU and Memory Address Busses
  width differ!" severity failure;
  assert (C_MEMBUS_DWIDTH = C_CPUBUS_DWIDTH) report "ERROR: CPU and Memory Data Busses width
  differ!" severity failure;
  assert (C_MEMBUS_AWIDTH = 32) report "ERROR: Address bus not 32-bits" severity failure;
  assert (C_MEMBUS_DWIDTH = 64) report "ERROR: Data bus not 64-bits" severity failure;

  ---------------------------------------------------------------------------
  -- Component instantiations
  ---------------------------------------------------------------------------
  Bridge_CPU_Interface_1 : entity plb_emu_v1_00_a.Bridge_CPU_Interface
    generic map (
      C_CPUBUS_AWIDTH => C_CPUBUS_AWIDTH,
      C_CPUBUS_DWIDTH => C_CPUBUS_DWIDTH)
    port map (
      oMemBus_PLB_MAddrAck      => oTransBus_Sl_AddrAck,
      oMemBus_PLB_MBusy         => oTransBus_Sl_MBusy(0),
      oMemBus_PLB_MErr          => oTransBus_Sl_MErr(0),
      oMemBus_PLB_MRearbitrate  => oTransBus_Sl_rearbitrate,
      oMemBus_PLB_MSSize        => oTransBus_Sl_SSize,
      iMemBus_M_abort           => iTransBus_PLB_abort,
      iMemBus_M_ABus            => iTransBus_PLB_ABus,
      iMemBus_M_BE              => iTransBus_PLB_BE,
      iMemBus_M_busLock         => iTransBus_PLB_busLock,
      iMemBus_M_compress        => iTransBus_PLB_compress,
      iMemBus_M_guarded         => iTransBus_PLB_guarded,
      iMemBus_M_lockErr         => iTransBus_PLB_lockErr,
      iMemBus_M_MSize           => iTransBus_PLB_MSize,
      iMemBus_M_ordered         => iTransBus_PLB_ordered,
      iMemBus_M_priority        => iTransBus_PLB_pendPri,
      iMemBus_M_request         => iTransBus_PLB_pendReq,
      iMemBus_M_RNW             => iTransBus_PLB_RNW,
      iMemBus_M_size            => iTransBus_PLB_size,
      iMemBus_M_type            => iTransBus_PLB_type,
      oMemBus_PLB_SMBusy        => oTransBus_Sl_rdComp,
      oMemBus_PLB_SMErr         => oTransBus_Sl_wrComp,
```

```
    iMemBus_M_wrBurst          => iTransBus_PLB_wrBurst,
    iMemBus_M_wrDBus           => iTransBus_PLB_wrDBus,
    oMemBus_PLB_MWrBTerm       => oTransBus_Sl_wrBTerm,
    oMemBus_PLB_MWrDAck        => oTransBus_Sl_wrDAck,
    iMemBus_M_rdBurst          => iTransBus_PLB_rdBurst,
    oMemBus_PLB_MRdBTerm       => oTransBus_Sl_rdBTerm,
    oMemBus_PLB_MRdDAck        => oTransBus_Sl_rdDAck,
    oMemBus_PLB_MRdDBus        => oTransBus_Sl_rdDBus,
    oMemBus_PLB_MRdWdAddr      => oTransBus_Sl_rdWdAddr,
    oMem_M_ABus                => s1_M_ABus,
    oMem_M_request             => s1_M_request,
    oMem_M_size                => s1_M_size,
    oMem_M_abort               => s1_M_abort,
    oMem_M_BE                  => s1_M_BE,
    iMem_PLB_MAddrAck          => s1_PLB_MAddrAck,
    iMem_PLB_MBusy             => s1_PLB_MBusy,
    iMem_PLB_MRdDAck           => s1_PLB_MRdDAck,
    iMem_PLB_MRdDBus           => s1_PLB_MRdDBus,
    iMem_PLB_MRdWdAddr         => s1_PLB_MRdWdAddr);

Bridge_Decrypt_1 : entity plb_emu_v1_00_a.Bridge_CounterModeDecrypt
  port map (
    Clk                        => MemClk,
    Reset                      => MemReset,
    iCPU_M_ABus                => s1_M_ABus,
    iCPU_M_request             => s1_M_request,
    iCPU_M_size                => s1_M_size,
    iCPU_M_abort               => s1_M_abort,
    iCPU_M_BE                  => s1_M_BE,
    oCPU_PLB_MAddrAck          => s1_PLB_MAddrAck,
    oCPU_PLB_MBusy             => s1_PLB_MBusy,
    oCPU_PLB_MRdDAck           => s1_PLB_MRdDAck,
    oCPU_PLB_MRdDBus           => s1_PLB_MRdDBus,
    oCPU_PLB_MRdWdAddr         => s1_PLB_MRdWdAddr,
    oMem_M_ABus                => s2_M_ABus,
    oMem_M_request             => s2_M_request,
    oMem_M_size                => s2_M_size,
    oMem_M_abort               => s2_M_abort,
    oMem_M_BE                  => s2_M_BE,
    iMem_PLB_MAddrAck          => s2_PLB_MAddrAck,
    iMem_PLB_MBusy             => s2_PLB_MBusy,
    iMem_PLB_MRdDAck           => s2_PLB_MRdDAck,
    iMem_PLB_MRdDBus           => s2_PLB_MRdDBus,
    iMem_PLB_MRdWdAddr         => s2_PLB_MRdWdAddr,
    oTableReq                  => oTableReq,
    iTableSearchDone           => iTableSearchDone,
    oPageAddress               => oPageAddress,
    iPageEncrypted             => iPageEncrypted,
    iPageKey                   => iPageKey,
    iPageAncillary             => iPageAncillary,
    oPerformance_EncFetch      => oPerformance_EncFetch,
    oPerformance_Fetch         => oPerformance_Fetch);

Bridge_Memory_Interface_1 : entity plb_emu_v1_00_a.Bridge_Memory_Interface
  generic map (
    C_MEMBUS_AWIDTH => C_MEMBUS_AWIDTH,
    C_MEMBUS_DWIDTH => C_MEMBUS_DWIDTH)
  port map (
    iMemBus_PLB_MAddrAck       => iMemBus_PLB_MAddrAck,
    iMemBus_PLB_MBusy          => iMemBus_PLB_MBusy,
    iMemBus_PLB_MErr           => iMemBus_PLB_MErr,
    iMemBus_PLB_MRearbitrate   => iMemBus_PLB_MRearbitrate,
    iMemBus_PLB_MSSize         => iMemBus_PLB_MSSize,
    oMemBus_M_abort            => oMemBus_M_abort,
```

```
        oMemBus_M_ABus          => oMemBus_M_ABus,
        oMemBus_M_BE            => oMemBus_M_BE,
        oMemBus_M_busLock       => oMemBus_M_busLock,
        oMemBus_M_compress      => oMemBus_M_compress,
        oMemBus_M_guarded       => oMemBus_M_guarded,
        oMemBus_M_lockErr       => oMemBus_M_lockErr,
        oMemBus_M_MSize         => oMemBus_M_MSize,
        oMemBus_M_ordered       => oMemBus_M_ordered,
        oMemBus_M_priority      => oMemBus_M_priority,
        oMemBus_M_request       => oMemBus_M_request,
        oMemBus_M_RNW           => oMemBus_M_RNW,
        oMemBus_M_size          => oMemBus_M_size,
        oMemBus_M_type          => oMemBus_M_type,
        iMemBus_PLB_SMBusy      => iMemBus_PLB_SMBusy,
        iMemBus_PLB_SMErr       => iMemBus_PLB_SMErr,
        oMemBus_M_wrBurst       => oMemBus_M_wrBurst,
        oMemBus_M_wrDBus        => oMemBus_M_wrDBus,
        iMemBus_PLB_MWrBTerm    => iMemBus_PLB_MWrBTerm,
        iMemBus_PLB_MWrDAck     => iMemBus_PLB_MWrDAck,
        oMemBus_M_rdBurst       => oMemBus_M_rdBurst,
        iMemBus_PLB_MRdBTerm    => iMemBus_PLB_MRdBTerm,
        iMemBus_PLB_MRdDAck     => iMemBus_PLB_MRdDAck,
        iMemBus_PLB_MRdDBus     => iMemBus_PLB_MRdDBus,
        iMemBus_PLB_MRdWdAddr   => iMemBus_PLB_MRdWdAddr,
        iCPU_M_ABus             => s2_M_ABus,
        iCPU_M_request          => s2_M_request,
        iCPU_M_size             => s2_M_size,
        iCPU_M_abort            => s2_M_abort,
        ICPU_M_BE               => s2_M_BE,
        oCPU_PLB_MAddrAck       => s2_PLB_MAddrAck,
        oCPU_PLB_MBusy          => s2_PLB_MBusy,
        oCPU_PLB_MRdDAck        => s2_PLB_MRdDAck,
        oCPU_PLB_MRdDBus        => s2_PLB_MRdDBus,
        oCPU_PLB_MRdWdAddr      => s2_PLB_MRdWdAddr);

end arch;
```

D.1.2 Bridge_CPU_Interface.vhd

```
-------------------------------------------------------------------------------
-- Title      : PLB CPU side interface module
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : Bridge_CPU_Interface.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configruable Computing Lab
-- Created    : 2005-04-11
-- Last update: 2005-07-10
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Reduces the full PLB bus specification to the complete subset
--              of signals and transaction types used by the PPC405 Instruction
--              side bus, in addition to being a read only interface
-------------------------------------------------------------------------------
-- Assumptions: Read only interface
--              Ignores all modes but single beat, 4-wd cache, and 8-wd cache
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-05-02  amahar  Switched from old style Req/Addr/Ack/Data requesting
--                      interface to pseudo-PLB interface to allow 0-latency
--                      optional transfers
-- 2005-04-11  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------------------------------------

entity Bridge_CPU_Interface is

  -----------------------------------------------------------------------------
  -- Entity: generics
  -----------------------------------------------------------------------------
  generic (
    C_CPUBUS_AWIDTH : integer := 32;
    C_CPUBUS_DWIDTH : integer := 64);

  -----------------------------------------------------------------------------
  -- Entity: ports
  -----------------------------------------------------------------------------
  port (
    -- Master side PLB bus signals
    -- Request Qualifiers
    oMemBus_PLB_MAddrAck      : out std_logic;
    oMemBus_PLB_MBusy         : out std_logic;
    oMemBus_PLB_MErr          : out std_logic;
    oMemBus_PLB_MRearbitrate  : out std_logic;
    oMemBus_PLB_MSSize        : out std_logic_vector(0 to 1);
    iMemBus_M_abort           : in  std_logic;
    iMemBus_M_ABus            : in  std_logic_vector(0 to C_CPUBUS_AWIDTH-1);
    iMemBus_M_BE              : in  std_logic_vector(0 to ((C_CPUBUS_DWIDTH/8))-1);
    iMemBus_M_busLock         : in  std_logic;
    iMemBus_M_compress        : in  std_logic;
    iMemBus_M_guarded         : in  std_logic;
    iMemBus_M_lockErr         : in  std_logic;
    iMemBus_M_MSize           : in  std_logic_vector(0 to 1);
```

```
    iMemBus_M_ordered          : in  std_logic;
    iMemBus_M_priority         : in  std_logic_vector(0 to 1);
    iMemBus_M_request          : in  std_logic;
    iMemBus_M_RNW              : in  std_logic;
    iMemBus_M_size             : in  std_logic_vector(0 to 3);
    iMemBus_M_type             : in  std_logic_vector(0 to 2);
    oMemBus_PLB_SMBusy         : out std_logic;
    oMemBus_PLB_SMErr          : out std_logic;
    -- Write Data Bus
    iMemBus_M_wrBurst          : in  std_logic;
    iMemBus_M_wrDBus           : in  std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
    oMemBus_PLB_MWrBTerm       : out std_logic;
    oMemBus_PLB_MWrDAck        : out std_logic;
    -- Read Data Bus
    iMemBus_M_rdBurst          : in  std_logic;
    oMemBus_PLB_MRdBTerm       : out std_logic;
    oMemBus_PLB_MRdDAck        : out std_logic;
    oMemBus_PLB_MRdDBus        : out std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
    oMemBus_PLB_MRdWdAddr      : out std_logic_vector(0 to 3);
    -- Unused Master side signals
    --iMemBus_PLB_pendReq       : in  std_logic;
    --iMemBus_PLB_pendPri       : in  std_logic_vector(0 to 1);
    --iMemBus_PLB_reqPri        : in  std_logic_vector(0 to 1);

-- Standard bridge plug-in interface
    oMem_M_ABus          : out std_logic_vector(0 to C_CPUBUS_AWIDTH-1);
    oMem_M_request       : out std_logic;
    oMem_M_size          : out std_logic_vector(0 to 1);
    oMem_M_abort         : out std_logic;
    oMem_M_BE            : out std_logic_vector(0 to ((C_CPUBUS_DWIDTH/8))-1);
    iMem_PLB_MAddrAck    : in  std_logic;
    iMem_PLB_MBusy       : in  std_logic;
    iMem_PLB_MRdDAck     : in  std_logic;
    iMem_PLB_MRdDBus     : in  std_logic_vector(0 to C_CPUBUS_DWIDTH-1);
    iMem_PLB_MRdWdAddr   : in  std_logic_vector(0 to 3));

end entity Bridge_CPU_Interface;


------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture arch of Bridge_CPU_Interface is

------------------------------------------------------------------------------
-- Begin architecture
------------------------------------------------------------------------------

begin

  ------------------------------------------------------------------------
  -- Constant Logic
  ------------------------------------------------------------------------
  oMemBus_PLB_MErr          <= '0';
  oMemBus_PLB_MRdBTerm      <= '0';
  oMemBus_PLB_MRearbitrate  <= '0';
  oMemBus_PLB_MWrBTerm      <= '0';
  oMemBus_PLB_MWrDAck       <= '0';
  oMemBus_PLB_SMBusy        <= '0';
  oMemBus_PLB_SMErr         <= '0';
  oMemBus_PLB_MSSize        <= "01";

  ------------------------------------------------------------------------
  -- Forwarded Logic
  ------------------------------------------------------------------------
```

```
  oMem_M_request       <= iMemBus_M_request;
  oMem_M_abort         <= iMemBus_M_abort;
  oMem_M_ABus          <= iMemBus_M_ABus;
  oMem_M_size(0 to 1) <= iMemBus_M_size(2 to 3);
  oMem_M_BE            <= iMemBus_M_BE;

  oMemBus_PLB_MAddrAck  <= iMem_PLB_MAddrAck;
  oMemBus_PLB_MBusy     <= iMem_PLB_MBusy;
  oMemBus_PLB_MRdDAck   <= iMem_PLB_MRdDAck;
  oMemBus_PLB_MRdDBus   <= iMem_PLB_MRdDBus;
  oMemBus_PLB_MRdWdAddr <= iMem_PLB_MRdWdAddr;

end architecture arch;
```

D.1.3 Bridge_Memory_Interface.vhd

```
-------------------------------------------------------------------------------
-- Title      : PLB memory side interface module
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : Bridge_Memory_Interface.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-04-12
-- Last update: 2005-07-16
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Reduces the full PLB bus specification to the complete subset
--              of signals and transaction types used by the PPC405 Instruction
--              side bus, in addition to being a read only interface
-------------------------------------------------------------------------------
-- Assumptions: Read only interface
--              Ignores all modes but single beat, 4-wd cache, and 8-wd cache
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-05-02  amahar  Switched from old style Req/Addr/Ack/Data requesting
--                      interface to pseudo-PLB interface to allow 0-latency
--                      optional transfers
-- 2005-04-12  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------------------------------------

entity Bridge_Memory_Interface is

  -----------------------------------------------------------------------------
  -- Entity: generics
  -----------------------------------------------------------------------------
  generic (
    C_MEMBUS_AWIDTH : integer := 32;
    C_MEMBUS_DWIDTH : integer := 64);

  -----------------------------------------------------------------------------
  -- Entity: ports
  -----------------------------------------------------------------------------
  port (
    -- Master side PLB bus signals
    -- Request Qualifiers
    iMemBus_PLB_MAddrAck      : in  std_logic;
    iMemBus_PLB_MBusy         : in  std_logic;
    iMemBus_PLB_MErr          : in  std_logic;
    iMemBus_PLB_MRearbitrate  : in  std_logic;
    iMemBus_PLB_MSSize        : in  std_logic_vector(0 to 1);
    oMemBus_M_abort           : out std_logic;
    oMemBus_M_ABus            : out std_logic_vector(0 to C_MEMBUS_AWIDTH-1);
    oMemBus_M_BE              : out std_logic_vector(0 to ((C_MEMBUS_DWIDTH/8))-1);
    oMemBus_M_busLock         : out std_logic;
    oMemBus_M_compress        : out std_logic;
    oMemBus_M_guarded         : out std_logic;
    oMemBus_M_lockErr         : out std_logic;
    oMemBus_M_MSize           : out std_logic_vector(0 to 1);
```

```
    oMemBus_M_ordered          : out std_logic;
    oMemBus_M_priority         : out std_logic_vector(0 to 1);
    oMemBus_M_request          : out std_logic;
    oMemBus_M_RNW              : out std_logic;
    oMemBus_M_size             : out std_logic_vector(0 to 3);
    oMemBus_M_type             : out std_logic_vector(0 to 2);
    iMemBus_PLB_SMBusy         : in  std_logic;
    iMemBus_PLB_SMErr          : in  std_logic;
    -- Write Data Bus
    oMemBus_M_wrBurst          : out std_logic;
    oMemBus_M_wrDBus           : out std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
    iMemBus_PLB_MWrBTerm       : in  std_logic;
    iMemBus_PLB_MWrDAck        : in  std_logic;
    -- Read Data Bus
    oMemBus_M_rdBurst          : out std_logic;
    iMemBus_PLB_MRdBTerm       : in  std_logic;
    iMemBus_PLB_MRdDAck        : in  std_logic;
    iMemBus_PLB_MRdDBus        : in  std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
    iMemBus_PLB_MRdWdAddr      : in  std_logic_vector(0 to 3);
    -- Unused Master side signals
    --iMemBus_PLB_pendReq       : in  std_logic;
    --iMemBus_PLB_pendPri       : in  std_logic_vector(0 to 1);
    --iMemBus_PLB_reqPri        : in  std_logic_vector(0 to 1);

-- Standard Bridge plug-in interface
    iCPU_M_ABus          : in  std_logic_vector(0 to C_MEMBUS_AWIDTH-1);
    iCPU_M_request       : in  std_logic;
    iCPU_M_size          : in  std_logic_vector(0 to 1);
    iCPU_M_abort         : in  std_logic;
    iCPU_M_BE            : in  std_logic_vector(0 to ((C_MEMBUS_DWIDTH/8))-1);
    oCPU_PLB_MAddrAck    : out std_logic;
    oCPU_PLB_MBusy       : out std_logic;
    oCPU_PLB_MRdDAck     : out std_logic;
    oCPU_PLB_MRdDBus     : out std_logic_vector(0 to C_MEMBUS_DWIDTH-1);
    oCPU_PLB_MRdWdAddr   : out std_logic_vector(0 to 3));

end entity Bridge_Memory_Interface;

-------------------------------------------------------------------------------
-- Architecture section
-------------------------------------------------------------------------------
architecture arch of Bridge_Memory_Interface is

-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------
begin

  ---------------------------------------------------------------------------
  -- Constant Logic
  ---------------------------------------------------------------------------
  oMemBus_M_wrDBus         <= (others => '0');
  oMemBus_M_wrBurst        <= '0';
  oMemBus_M_RNW            <= '1';
  oMemBus_M_type           <= "000";
  oMemBus_M_guarded        <= '0';
  oMemBus_M_ordered        <= '0';
  oMemBus_M_compress       <= '0';
  oMemBus_M_busLock        <= '0';
  oMemBus_M_lockErr        <= '0';
  oMemBus_M_priority       <= "11";
  oMemBus_M_rdBurst        <= '0';
  oMemBus_M_size(0 to 1)   <= "00";
  oMemBus_M_MSize          <= "01";
```

```
  ------------------------------------------------------------------------------
  -- Forwarded Logic
  ------------------------------------------------------------------------------
  oMemBus_M_request       <= iCPU_M_request;
  oMemBus_M_abort         <= iCPU_M_abort;
  oMemBus_M_ABus          <= iCPU_M_ABus;
  oMemBus_M_size(2 to 3) <= iCPU_M_size(0 to 1);
  oMemBus_M_BE            <= iCPU_M_BE;

  oCPU_PLB_MAddrAck  <= iMemBus_PLB_MAddrAck;
  oCPU_PLB_MBusy     <= iMemBus_PLB_MBusy;
  oCPU_PLB_MRdDAck   <= iMemBus_PLB_MRdDAck;
  oCPU_PLB_MRdDBus   <= iMemBus_PLB_MRdDBus;
  oCPU_PLB_MRdWdAddr <= iMemBus_PLB_MRdWdAddr;

end architecture arch;
```

D.1.4 Bridge_BlockModeDecrypt.vhd

```
-------------------------------------------------------------------------------
-- Title      : Block Mode Decryption Unit
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : Bridge_BlockModeDecrypt.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-05-05
-- Last update: 2005-07-17
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Decryption plug-in module for the EMU memory bridge.
--              Uses a direct block mode policy, and requires entire
--              transaction to be buffered before block decrypting n*BlockSize
--              chunks.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-07-09  amahar  Code cleanup, commenting
-- 2005-06-25  amahar  Added Ancillary input for plug-in compatibility, even
--                     though not used
-- 2005-05-05  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library plb_emu_v1_00_a;
use plb_emu_v1_00_a.all;

-------------------------------------------------------------------------------

entity Bridge_BlockModeDecrypt is

  -----------------------------------------------------------------------------
  -- Entity: generics
  -----------------------------------------------------------------------------
  generic (
    C_AWIDTH            : integer := 32;
    C_DWIDTH            : integer := 64;
    C_PAGE_AWIDTH       : integer := 20;
    C_KEYSIZE           : integer := 128;
    C_ANCILLARY_DWIDTH : integer := 128);

  -----------------------------------------------------------------------------
  -- Entity: ports
  -----------------------------------------------------------------------------
  port (
    Clk    : in std_logic;
    Reset : in std_logic;

    -- Plug-in CPU Interface
    iCPU_M_ABus          : in  std_logic_vector(0 to C_AWIDTH-1);
    iCPU_M_request       : in  std_logic;
    iCPU_M_size          : in  std_logic_vector(0 to 1);
    iCPU_M_abort         : in  std_logic;
    iCPU_M_BE            : in  std_logic_vector(0 to ((C_DWIDTH/8))-1);
    oCPU_PLB_MAddrAck   : out std_logic;
```

```
    oCPU_PLB_MBusy      : out std_logic;
    oCPU_PLB_MRdDAck    : out std_logic;
    oCPU_PLB_MRdDBus    : out std_logic_vector(0 to C_DWIDTH-1);
    oCPU_PLB_MRdWdAddr  : out std_logic_vector(0 to 3);

    -- Plug-in Memory Interface
    oMem_M_ABus         : out std_logic_vector(0 to C_AWIDTH-1);
    oMem_M_request      : out std_logic;
    oMem_M_size         : out std_logic_vector(0 to 1);
    oMem_M_abort        : out std_logic;
    oMem_M_BE           : out std_logic_vector(0 to ((C_DWIDTH/8))-1);
    iMem_PLB_MAddrAck   : in  std_logic;
    iMem_PLB_MBusy      : in  std_logic;
    iMem_PLB_MRdDAck    : in  std_logic;
    iMem_PLB_MRdDBus    : in  std_logic_vector(0 to C_DWIDTH-1);
    iMem_PLB_MRdWdAddr  : in  std_logic_vector(0 to 3);

    -- Table Interface
    oTableReq           : out std_logic;
    iTableSearchDone    : in  std_logic;
    oPageAddress        : out std_logic_vector(0 to C_PAGE_AWIDTH-1);
    iPageEncrypted      : in  std_logic;
    iPageKey            : in  std_logic_vector(0 to C_KEYSIZE-1);
    iPageAncillary      : in  std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
    oPerformance_Fetch  : out std_logic;
    oPerformance_EncFetch : out std_logic);

end entity Bridge_BlockModeDecrypt;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture arch of Bridge_BlockModeDecrypt is

  ------------------------------------------------------------------------------
  -- Internal constant declarations
  ------------------------------------------------------------------------------
  type StateType is (ST_WaitAddrAck, ST_Read, ST_Decrypt, ST_Write);

  ------------------------------------------------------------------------------
  -- Internal signal declarations
  ------------------------------------------------------------------------------
  signal sClk_CurrState : StateType := ST_WaitAddrAck;
  signal sNextState     : StateType := ST_WaitAddrAck;

  signal sPLB_MAddrAck  : std_logic;
  signal sPLB_MRdDBus   : std_logic_vector(0 to C_DWIDTH-1);
  signal sPLB_MRdDBus_0 : std_logic_vector(0 to C_DWIDTH-1);
  signal sM_request     : std_logic;
  signal sM_abort       : std_logic;
  signal sM_size        : std_logic_vector(0 to 1);
  signal sM_BE          : std_logic_vector(0 to ((C_DWIDTH/8))-1);

  signal sClk_DataIn        : std_logic_vector(0 to 255);
  signal sClk_DataOut       : std_logic_vector(0 to 255);
  signal sClk_PLB_MBusy     : std_logic;
  signal sClk_PLB_MRdDAck   : std_logic;
  signal sClk_PLB_MRdWdAddr : std_logic_vector(0 to 2);

  signal sClk_MemReadCount     : std_logic_vector(0 to 1);
  signal sClk_CPUReadCount     : std_logic_vector(0 to 1);
  signal sClk_MaxMemReadCount  : std_logic_vector(0 to 1);
  signal sClk_MaxCPUReadCount  : std_logic_vector(0 to 1);
```

```
  signal sClk_Encrypted : std_logic;
  signal sClk_Key       : std_logic_vector(0 to C_KEYSIZE-1);

  signal sDecryptDataOut : std_logic_vector(0 to 255);
  signal sDecryptAck     : std_logic;
  signal sDecryptDone    : std_logic;
  signal sClk_DecryptReq : std_logic;

------------------------------------------------------------------------
-- Begin architecture
------------------------------------------------------------------------
begin

  ----------------------------------------------------------------------
  -- Crypto core instantiations
  ----------------------------------------------------------------------
  crypto_module_1 : entity plb_emu_v1_00_a.crypto_xor
    port map (
      Clk           => Clk,
      Reset         => Reset,
      iRequest      => sClk_DecryptReq,
      iData         => sClk_DataIn(0 to 127),
      iKey          => sClk_Key,
      oAcknowledge => sDecryptAck,
      oDone         => sDecryptDone,
      oData         => sDecryptDataOut(0 to 127));

  crypto_module_2 : entity plb_emu_v1_00_a.crypto_xor
    port map (
      Clk           => Clk,
      Reset         => Reset,
      iRequest      => sClk_DecryptReq,
      iData         => sClk_DataIn(128 to 255),
      iKey          => sClk_Key,
      oAcknowledge => open,
      oDone         => open,
      oData         => sDecryptDataOut(128 to 255));

  ----------------------------------------------------------------------
  -- Address Transaction Selection
  ----------------------------------------------------------------------
  TransReq : process (iCPU_M_abort, iCPU_M_request, iMem_PLB_MAddrAck,
                      sClk_CurrState) is
  begin
    if (sClk_CurrState = ST_WaitAddrAck) then
      -- Pass through all signals when not involved in a transaction
      sPLB_MAddrAck <= iMem_PLB_MAddrAck;
      sM_request    <= iCPU_M_request;
      sM_abort      <= iCPU_M_abort;
      sM_BE         <= iCPU_M_BE;
    else
      -- Loop-back abort signal to acknowledge per PLB spec
      -- disable request/abort to memory bus
      sPLB_MAddrAck <= iCPU_M_abort;
      sM_request    <= '0';
      sM_abort      <= '0';
      sM_BE         <= (others => '0');
    end if;
  end process TransReq;

  -- Up-convert 64-bit transactions to 128-bit
  sM_size <= "01" when iCPU_M_size = "00" else iCPU_M_size;

  -- Forward any muxed signals
```

```
oMem_M_ABus         <= iCPU_M_ABus;
oMem_M_size         <= sM_size;
oMem_M_abort        <= sM_abort;
oMem_M_request      <= sM_request;
oMem_M_BE           <= sM_BE;
oCPU_PLB_MAddrAck <= sPLB_MAddrAck;

-------------------------------------------------------------------------------
-- Data Transaction Selection
-------------------------------------------------------------------------------
-- If not encrypted, or not dealing with a multiple of 128-bit block, forward
-- signals from memory side to CPU side, otherwise use the delayed, decrypted
-- signals
TransAck : process (iMem_PLB_MBusy, iMem_PLB_MRdDAck, iMem_PLB_MRdDBus,
                    iMem_PLB_MRdWdAddr, sClk_Encrypted, sClk_MaxCPUReadCount,
                    sClk_PLB_MBusy, sClk_PLB_MRdDAck, sClk_PLB_MRdWdAddr,
                    sPLB_MRdDBus)
begin
  if (sClk_Encrypted = '0' and sClk_MaxCPUReadCount /= "00") then
    oCPU_PLB_MRdDBus   <= iMem_PLB_MRdDBus;
    oCPU_PLB_MBusy     <= iMem_PLB_MBusy;
    oCPU_PLB_MRdDAck   <= iMem_PLB_MRdDAck;
    oCPU_PLB_MRdWdAddr <= iMem_PLB_MRdWdAddr;
  else
    oCPU_PLB_MRdDBus   <= sPLB_MRdDBus;
    oCPU_PLB_MBusy     <= sClk_PLB_MBusy;
    oCPU_PLB_MRdDAck   <= sClk_PLB_MRdDAck;
    oCPU_PLB_MRdWdAddr <= sClk_PLB_MRdWdAddr & '0';
  end if;
end process TransAck;

-------------------------------------------------------------------------------
-- Table Search Interface
-------------------------------------------------------------------------------
-- A request occurs on every official address transaction acknowledge
oTableReq    <= iCPU_M_request and (not iCPU_M_abort) and iMem_PLB_MAddrAck;
oPageAddress <= iCPU_M_ABus(0 to C_PAGE_AWIDTH-1);

EncLatch : process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_Encrypted <= '0';
    sClk_Key       <= (others => '0');
  elsif rising_edge(Clk) then
    if (iTableSearchDone = '1') then
      sClk_Encrypted <= iPageEncrypted;
      sClk_Key       <= iPageKey;
    end if;
  end if;
end process EncLatch;

-------------------------------------------------------------------------------
-- Input and Output buffering / selection
-------------------------------------------------------------------------------

-- On data acknowledge, select the appropraite 64-bit data beat from the
-- output buffer
process (sClk_DataOut, sClk_PLB_MRdWdAddr) is
begin
  case sClk_PLB_MRdWdAddr(1 to 2) is
    when "00"   => sPLB_MRdDBus_0 <= sClk_DataOut(0 to 63);
    when "01"   => sPLB_MRdDBus_0 <= sClk_DataOut(64 to 127);
    when "10"   => sPLB_MRdDBus_0 <= sClk_DataOut(128 to 191);
    when others => sPLB_MRdDBus_0 <= sClk_DataOut(192 to 255);
```

```
  end case;
end process;
sPLB_MRdDBus <= sPLB_MRdDBus_0 when sClk_PLB_MRdDAck = '1' else (others => '0');

-- Store incoming read data bus from Memory into buffer.  Buffer is simply
-- not used if transaction is not encrypted
process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_DataIn <= (others => '0');
  elsif rising_edge(Clk) then
    if (iMem_PLB_MRdDAck = '1') then
      case iMem_PLB_MRdWdAddr(1 to 2) is
        when "00"   => sClk_DataIn(0 to 63)   <= iMem_PLB_MRdDBus;
        when "01"   => sClk_DataIn(64 to 127)  <= iMem_PLB_MRdDBus;
        when "10"   => sClk_DataIn(128 to 191) <= iMem_PLB_MRdDBus;
        when "11"   => sClk_DataIn(192 to 255) <= iMem_PLB_MRdDBus;
        when others => null;
      end case;
    end if;
  end if;
end process;


-------------------------------------------------------------------------------
-- Block Decryption Finite State Machine
-------------------------------------------------------------------------------

-- Combinational FSM process
FSM_Cmb : process (iCPU_M_abort, iCPU_M_request, iMem_PLB_MAddrAck,
                   iMem_PLB_MRdDAck, sClk_CPUReadCount, sClk_CurrState,
                   sClk_Encrypted, sClk_MaxCPUReadCount,
                   sClk_MaxMemReadCount, sClk_MemReadCount, sDecryptDone) is
begin
  case sClk_CurrState is

    when ST_WaitAddrAck =>
      -- Transition to reading after official address transaction acknowledge
      if (iCPU_M_request = '1' and iCPU_M_abort = '0' and iMem_PLB_MAddrAck = '1') then
        sNextState <= ST_Read;
      else
        sNextState <= ST_WaitAddrAck;
      end if;

    when ST_Read =>
      -- If last read data beat has been achieved, either progress to decryption,
      -- or return to waiting for next transaction.  If last beat has not
      -- been reached, continue to wait
      if (sClk_MemReadCount = sClk_MaxMemReadCount and iMem_PLB_MRdDAck = '1') then
        if (sClk_Encrypted = '1' or sClk_MaxCPUReadCount = "00") then
          sNextState <= ST_Decrypt;
        else
          sNextState <= ST_WaitAddrAck;
        end if;
      else
        sNextState <= ST_Read;
      end if;

    when ST_Decrypt =>
      -- If in this state, but not encrypted, then move data from input buffer
      -- directly to output buffer, and progress to write state.  Otherwise,
      -- wait for data blocks to be decrypted
      if (sClk_Encrypted = '0') then
        sNextState <= ST_Write;
```

```
        elsif (sDecryptDone = '1') then
          sNextState <= ST_Write;
        else
          sNextState <= ST_Decrypt;
        end if;

      when ST_Write =>
        -- Write correct transaction words and size to the CPU
        if (sClk_CPUReadCount = sClk_MaxCPUReadCount) then
          sNextState <= ST_WaitAddrAck;
        else
          sNextState <= ST_Write;
        end if;

      when others =>
        sNextState <= ST_WaitAddrAck;
    end case;
  end process FSM_Cmb;


  -- Sequential FSM process
  FSM_Seq : process (Clk, Reset) is
  begin
    if Reset = '1' then
      sClk_CurrState        <= ST_WaitAddrAck;
      sClk_PLB_MBusy        <= '0';
      sClk_PLB_MRdDAck      <= '0';
      sClk_PLB_MRdWdAddr    <= "000";
      sClk_MemReadCount     <= "00";
      sClk_CPUReadCount     <= "00";
      sClk_MaxMemReadCount  <= "00";
      sClk_MaxCPUReadCount  <= "00";
      sClk_DataOut          <= (others => '0');
      sClk_DecryptReq       <= '0';
    elsif rising_edge(Clk) then
      sClk_CurrState <= sNextState;

      case sClk_CurrState is

        when ST_WaitAddrAck =>
          -- Latch in busy when address phase is complete
          sClk_PLB_MBusy <= iCPU_M_request and (not iCPU_M_abort) and iMem_PLB_MAddrAck;

          -- Establish correct memory and cpu transaction sizes.  They differ
          -- only on a 64-bit request
          case iCPU_M_size is
            when "01" =>
              sClk_MaxMemReadCount <= "01";
              sClk_MaxCPUReadCount <= "01";
            when "10" =>
              sClk_MaxMemReadCount <= "11";
              sClk_MaxCPUReadCount <= "11";
            when others =>
              sClk_MaxMemReadCount <= "01";
              sClk_MaxCPUReadCount <= "00";
          end case;

          -- Store the target first word
          sClk_PLB_MRdWdAddr <= iCPU_M_ABus(C_AWIDTH-6 to C_AWIDTH-4);
          -- Disable data acknowledge to the CPU
          sClk_PLB_MRdDAck   <= '0';
          -- Zero out Memory, CPU transaction counters and decryption
          sClk_MemReadCount  <= "00";
          sClk_CPUReadCount  <= "00";
```

```
      when ST_Read =>
        -- Increment Memory read counter when memory data ack occurs
        if (iMem_PLB_MRdDAck = '1') then
          sClk_MemReadCount <= sClk_MemReadCount + '1';
        end if;
        if (sClk_MemReadCount = sClk_MaxMemReadCount and iMem_PLB_MRdDAck = '1' and
        sClk_Encrypted = '1') then
          sClk_DecryptReq <= '1';
        end if;

      when ST_Decrypt =>
        if (sClk_Encrypted = '0') then
          -- If not encrypted, but in this state, a single beat
          -- transaction has occured.  Shift input directly to output.
          -- as this is a single data beat, enable CPU data output
          sClk_DataOut     <= sClk_DataIn;
          sClk_PLB_MRdDAck <= '1';
        elsif (sDecryptDone = '1') then
          -- If encrypted and decryption complete, store in output buffer,
          -- and enable output
          sClk_DataOut     <= sDecryptDataOut;
          sClk_PLB_MRdDAck <= '1';
        end if;

        -- Disable request once crypto module has acknowledged receipt
        if sDecryptAck = '1' then
          sClk_DecryptReq <= '0';
        end if;

      when ST_Write =>
        -- Note, the first cycle of this state handles the single beat case,
        -- the first single beat is always initiated from the previous state
        if (sClk_CPUReadCount = sClk_MaxCPUReadCount) then
          -- When complete disable busy and data ack.
          sClk_PLB_MBusy   <= '0';
          sClk_PLB_MRdDAck <= '0';
        else
          -- Increment word pointers, and wrap at the appropriate boundary
          -- depending on transaction size
          if (sClk_MaxCPUReadCount = "01") then
            sClk_PLB_MRdWdAddr <= sClk_PLB_MRdWdAddr(0 to 1) & not sClk_PLB_MRdWdAddr(2);
          else
            sClk_PLB_MRdWdAddr <= sClk_PLB_MRdWdAddr(0) & (sClk_PLB_MRdWdAddr(1 to 2) +
            '1');
          end if;
        end if;

        -- Increment the read count
        sClk_CPUReadCount <= sClk_CPUReadCount + '1';

      when others => null;
    end case;
  end if;
end process FSM_Seq;

-------------------------------------------------------------------------------
-- Performance Counter Handler
-------------------------------------------------------------------------------

-- Create a strobe for every encrypted transaction
process (Clk, Reset) is
begin
  if (Reset = '1') then
```

```
      oPerformance_Fetch <= '0';
      oPerformance_EncFetch <= '0';
    elsif (rising_edge(Clk)) then
      if iTableSearchDone = '1' then
        oPerformance_Fetch <= '1';
      else
        oPerformance_Fetch <= '0';
      end if;
      if iTableSearchDone = '1' and iPageEncrypted = '1' then
        oPerformance_EncFetch <= '1';
      else
        oPerformance_EncFetch <= '0';
      end if;
    end if;
  end process;

end architecture arch;
```

D.1.5 Bridge_CounterModeDecrypt.vhd

```
-------------------------------------------------------------------------------
-- Title      : Counter Mode Decryption Unit
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : Bridge_CounterModeDecrypt.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-06-25
-- Last update: 2005/07/28
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Counter-mode decryption unit for encrypted / unencrypted memory
--              transactions within the Secure Software architecture.  Follows
--              plug-in module interface for the SecSoft bridge unit.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-06-25  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library plb_emu_v1_00_a;
use plb_emu_v1_00_a.all;

-------------------------------------------------------------------------------

entity Bridge_CounterModeDecrypt is

  generic (
    C_AWIDTH            : integer := 32;
    C_DWIDTH            : integer := 64;
    C_PAGE_AWIDTH       : integer := 20;
    C_KEYSIZE           : integer := 128;
    C_ANCILLARY_DWIDTH : integer := 128);

  port (
    Clk                 : in  std_logic;
    Reset               : in  std_logic;
    iCPU_M_ABus         : in  std_logic_vector(0 to C_AWIDTH-1);
    iCPU_M_request      : in  std_logic;
    iCPU_M_size         : in  std_logic_vector(0 to 1);
    iCPU_M_abort        : in  std_logic;
    iCPU_M_BE           : in  std_logic_vector(0 to ((C_DWIDTH/8))-1);
    oCPU_PLB_MAddrAck   : out std_logic;
    oCPU_PLB_MBusy      : out std_logic;
    oCPU_PLB_MRdDAck    : out std_logic;
    oCPU_PLB_MRdDBus    : out std_logic_vector(0 to C_DWIDTH-1);
    oCPU_PLB_MRdWdAddr : out std_logic_vector(0 to 3);

    oMem_M_ABus     : out std_logic_vector(0 to C_AWIDTH-1);
    oMem_M_request  : out std_logic;
    oMem_M_size     : out std_logic_vector(0 to 1);
    oMem_M_abort    : out std_logic;
    oMem_M_BE       : out std_logic_vector(0 to ((C_DWIDTH/8))-1);

    iMem_PLB_MAddrAck  : in std_logic;
```

```
    iMem_PLB_MBusy      : in std_logic;
    iMem_PLB_MRdDAck    : in std_logic;
    iMem_PLB_MRdDBus    : in std_logic_vector(0 to C_DWIDTH-1);
    iMem_PLB_MRdWdAddr  : in std_logic_vector(0 to 3);

    oTableReq           : out std_logic;
    iTableSearchDone    : in  std_logic;
    oPageAddress        : out std_logic_vector(0 to C_PAGE_AWIDTH-1);
    iPageEncrypted      : in  std_logic;
    iPageKey            : in  std_logic_vector(0 to C_KEYSIZE-1);
    iPageAncillary      : in  std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
    oPerformance_EncFetch : out std_logic;
    oPerformance_Fetch  : out std_logic);

end entity Bridge_CounterModeDecrypt;

-------------------------------------------------------------------------------

architecture arch of Bridge_CounterModeDecrypt is

  type StateType is (ST_Wait, ST_Encrypt, ST_Write);

  -----------------------------------------------------------------------------
  -- Internal signal declarations
  -----------------------------------------------------------------------------

  signal sClk_CurrState, sNextState : StateType := ST_Wait;

  -- Crypto Signals
  signal sClk_CryptoRequest : std_logic;
  signal sClk_Key           : std_logic_vector(0 to 127);
  signal sClk_Ancillary     : std_logic_vector(0 to 127);

  -- FIFO Signals
  signal sFIFO_Push, sFIFO_Pop, sFIFO_Empty : std_logic;
  signal sDataFIFO_DataIn                   : std_logic_vector(0 to 255);
  signal sDataFIFO_DataOut                  : std_logic_vector(0 to 63);
  signal sRdWdAddrFIFO_DataIn               : std_logic_vector(0 to 3);
  signal sRdWdAddrFIFO_DataOut              : std_logic_vector(0 to 3);

  -- Crypto Signals
  signal sCrypto_Acknowledge, sCrypto_Done : std_logic;
  signal sCounter_1, sCounter_0            : std_logic_vector(0 to 127);
  signal sCryptoOut                        : std_logic_vector(0 to 255);
  signal sClk_CounterEncrypted             : std_logic_vector(0 to 255);
  signal sXOR                              : std_logic_vector(0 to 63);

  -- Transaction signals
  signal sClk_WordAddress                    : std_logic_vector(0 to 3);
  signal sClk_LineAddress                    : std_logic_vector(0 to C_AWIDTH -
  C_PAGE_AWIDTH - 5 - 1);
  signal sClk_PLB_MBusy                      : std_logic;
  signal sTransactionStart                   : std_logic;
  signal sClk_BeatCounter                    : std_logic_vector(0 to 1);
  signal sClk_MaxBeatCounter                 : std_logic_vector(0 to 1);
  signal sM_request, sM_abort, sPLB_MAddrAck : std_logic;
  signal sM_BE                               : std_logic_vector(0 to ((C_DWIDTH/8))-1);
  signal sCPU_PLB_MRdDAck                    : std_logic;

begin

  -- Performance outputs
  oPerformance_Fetch    <= sTransactionStart;
  oPerformance_EncFetch <= iTableSearchDone and iPageEncrypted;
```

```
-------------------------------------------------------------------------------
-- Crypto Core Interface
-------------------------------------------------------------------------------

sCounter_1 <= sClk_Ancillary(0 to C_ANCILLARY_DWIDTH - 8 - 1) & sClk_LineAddress & "1";
sCounter_0 <= sClk_Ancillary(0 to C_ANCILLARY_DWIDTH - 8 - 1) & sClk_LineAddress & "0";

crypto_1 : entity plb_emu_v1_00_a.crypto_xor
  port map (
    Clk          => Clk,
    Reset        => Reset,
    iRequest     => sClk_CryptoRequest,
    iData        => sCounter_1,
    iKey         => sClk_Key,
    oAcknowledge => sCrypto_Acknowledge,
    oDone        => sCrypto_Done,
    oData        => sCryptoOut(0 to 127));

crypto_0 : entity plb_emu_v1_00_a.crypto_xor
  port map (
    Clk          => Clk,
    Reset        => Reset,
    iRequest     => sClk_CryptoRequest,
    iData        => sCounter_0,
    iKey         => sClk_Key,
    oAcknowledge => open,
    oDone        => open,
    oData        => sCryptoOut(128 to 255));

-------------------------------------------------------------------------------
-- FIFO Interface
-------------------------------------------------------------------------------
sFIFO_Push             <= iMem_PLB_MRdDAck;
sFIFO_Pop              <= sCPU_PLB_MRdDAck;
sRdWdAddrFIFO_DataIn <= sClk_WordAddress when sClk_MaxBeatCounter = "00" else
iMem_PLB_MRdWdAddr;

sXOR <= sClk_CounterEncrypted(0 to 63) when sRdWdAddrFIFO_DataOut(1 to 2) = "10" else
        sClk_CounterEncrypted(64 to 127)  when sRdWdAddrFIFO_DataOut(1 to 2) = "11" else
        sClk_CounterEncrypted(128 to 191) when sRdWdAddrFIFO_DataOut(1 to 2) = "00" else
        sClk_CounterEncrypted(192 to 255);

sfifo_data : entity plb_emu_v1_00_a.sfifo
  generic map (
    C_WIDTH => 64)
  port map (
    Clk    => Clk,
    Reset  => Reset,
    iPush  => sFIFO_Push,
    iData  => iMem_PLB_MRdDBus,
    iPop   => sFIFO_Pop,
    oData  => sDataFIFO_DataOut,
    oEmpty => sFIFO_Empty,
    oFull  => open);

sfifo_rdwdaddr : entity plb_emu_v1_00_a.sfifo
  generic map (
    C_WIDTH => 4)
  port map (
    Clk    => Clk,
    Reset  => Reset,
    iPush  => sFIFO_Push,
    iData  => sRdWdAddrFIFO_DataIn,
```

```
    iPop   => sFIFO_Pop,
    oData  => sRdWdAddrFIFO_DataOut,
    oEmpty => open,
    oFull  => open);

-------------------------------------------------------------------------------
-- Table Search Interface
-------------------------------------------------------------------------------
-- A request occurs on every official address transaction acknowledge
oTableReq    <= sTransactionStart;
oPageAddress <= iCPU_M_ABus(0 to C_PAGE_AWIDTH-1);

TableLatch : process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_Key       <= (others => '0');
    sClk_Ancillary <= (others => '0');
  elsif rising_edge(Clk) then
    if (iTableSearchDone = '1') then
      sClk_Key       <= iPageKey;
      sClk_Ancillary <= iPageAncillary;
    end if;
  end if;
end process TableLatch;

-------------------------------------------------------------------------------
-- Address Transaction Selection
-------------------------------------------------------------------------------
sTransactionStart <= iCPU_M_request and (not iCPU_M_abort) and iMem_PLB_MAddrAck;
sCPU_PLB_MRdDAck  <= '1' when sFIFO_Empty = '0' and sClk_CurrState = ST_Write else '0';

TransReq : process (iCPU_M_BE, iCPU_M_abort, iCPU_M_request,
                    iMem_PLB_MAddrAck, sClk_CurrState) is
begin
  if (sClk_CurrState = ST_Wait) then
    -- Pass through all signals when not involved in a transaction
    sPLB_MAddrAck <= iMem_PLB_MAddrAck;
    sM_request    <= iCPU_M_request;
    sM_abort      <= iCPU_M_abort;
    sM_BE         <= iCPU_M_BE;
  else
    -- Loop-back abort signal to acknowledge per PLB spec
    -- disable request/abort to memory bus
    sPLB_MAddrAck <= iCPU_M_abort;
    sM_request    <= '0';
    sM_abort      <= '0';
    sM_BE         <= (others => '0');
  end if;
end process TransReq;

-- Forward any muxed signals
oMem_M_ABus         <= iCPU_M_ABus;
oMem_M_size         <= iCPU_M_size;
oMem_M_abort        <= sM_abort;
oMem_M_request      <= sM_request;
oMem_M_BE           <= sM_BE;
oCPU_PLB_MAddrAck   <= sPLB_MAddrAck;
oCPU_PLB_MRdDAck    <= sCPU_PLB_MRdDAck;
oCPU_PLB_MRdWdAddr <= sRdWdAddrFIFO_DataOut       when sCPU_PLB_MRdDAck = '1' else (others
=> '0');
oCPU_PLB_MRdDBus    <= sDataFIFO_DataOut xor sXOR when sCPU_PLB_MRdDAck = '1' else (others
=> '0');
oCPU_PLB_MBusy      <= sClk_PLB_MBusy;
```

```
-------------------------------------------------------------------------------
-- Counter-Mode Finite State machine
-------------------------------------------------------------------------------

-- Sequential process
FSM_Seq : process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_CurrState        <= ST_Wait;
    sClk_LineAddress      <= (others => '0');
    sClk_CryptoRequest    <= '0';
    sClk_CounterEncrypted <= (others => '0');
    sClk_BeatCounter      <= (others => '0');
    sClk_MaxBeatCounter   <= (others => '0');

  elsif rising_edge(Clk) then
    sClk_CurrState <= sNextState;

    case sClk_CurrState is
      when ST_Wait =>
        -- Store the requested address for counter mode
        sClk_LineAddress <= iCPU_M_ABus(C_PAGE_AWIDTH to C_AWIDTH - 5 - 1);

        -- Begin the busy period when the transaction starts
        sClk_PLB_MBusy <= sTransactionStart;

        -- Store the number of beats for the transaction
        case iCPU_M_size is
          when "01" =>
            sClk_MaxBeatCounter <= "01";
          when "10" =>
            sClk_MaxBeatCounter <= "11";
          when others =>
            sClk_MaxBeatCounter <= "00";
        end case;

        -- Reset the nubmer of beats transferred
        sClk_BeatCounter <= "00";

        -- Word address (needed later for single beat trans)
        sClk_WordAddress <= iCPU_M_ABus(C_AWIDTH-6 to C_AWIDTH-3);

      when ST_Encrypt =>

        -- Assert request to the crypto cores to process if the transaction
        -- is encrypted, deassert when crypto process complete
        if iTableSearchDone = '1' and iPageEncrypted = '1' then
          sClk_CryptoRequest <= '1';
        elsif sCrypto_Acknowledge = '1' then
          sClk_CryptoRequest <= '0';
        end if;

        -- If the transaction is not encrypted, clear out the encrypted
        -- counter buffer to xor against 0
        if iTableSearchDone = '1' and iPageEncrypted = '0' then
          sClk_CounterEncrypted <= (others => '0');
        else
          sClk_CounterEncrypted <= sCryptoOut;
        end if;

      when ST_Write =>
        -- Increment counted beats on internally generated Address ack
        if sCPU_PLB_MRdDAck = '1' then
          sClk_BeatCounter <= sClk_BeatCounter + '1';
```

```
          end if;

          -- Deassert busy after all reads have been completed
          if sClk_BeatCounter = sClk_MaxBeatCounter and sCPU_PLB_MRdDAck = '1' then
            sClk_PLB_MBusy <= '0';
          end if;
        when others => null;
      end case;
    end if;
  end process FSM_Seq;

  -- Combinational process
  FSM_Cmb : process (iPageEncrypted, iTableSearchDone, sCPU_PLB_MRdDAck,
                     sClk_BeatCounter, sClk_CurrState, sClk_MaxBeatCounter,
                     sCrypto_Done, sTransactionStart) is
  begin
    case sClk_CurrState is
      when ST_Wait =>
        if sTransactionStart = '1' then
          sNextState <= ST_Encrypt;
        else
          sNextState <= ST_Wait;
        end if;
      when ST_Encrypt =>
        if (iTableSearchDone = '1' and iPageEncrypted = '0') or (sCrypto_Done = '1') then
          sNextState <= ST_Write;
        else
          sNextState <= ST_Encrypt;
        end if;
      when ST_Write =>
        if sClk_BeatCounter = sClk_MaxBeatCounter and sCPU_PLB_MRdDAck = '1' then
          sNextState <= ST_Wait;
        else
          sNextState <= ST_Write;
        end if;
      when others =>
        sNextState <= ST_Wait;
    end case;
  end process FSM_Cmb;

end architecture arch;
```

D.1.6 Tables.vhd

```
--------------------------------------------------------------------------------
-- Title      : Page/Key Lookup Tables
-- Project    : Secure Software
--------------------------------------------------------------------------------
-- File       : Tables.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-03-10
-- Last update: 2005-07-17
-- Platform   :
-- Standard   : VHDL'93
--------------------------------------------------------------------------------
-- Description: Container for the page look up, key address look up, key look
--              up, and ancillary data look up.  Includes both a search/find
--              interface, and table writing interfaces.
--------------------------------------------------------------------------------
-- Copyright (c) 2005
--------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-07-09  amahar  General code clean-up and commenting.  Switched to
--                     StateType FSM encoding rather than direct encoding.
-- 2005-06-25  amahar  Added ancillary data to table lookups
-- 2005-05-07  amahar  Removed unecessary intermediate registers for the
--                     cross clock domain synchronizers, fixed up state
--                     machines to proper format recognizable by XST,
--                     removed the start/done feature which is no longer
--                     needed
-- 2005-04-01  amahar  Added clock synchronizers
-- 2005-03-10  amahar  Created
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library unisim;
use unisim.vcomponents.all;

--------------------------------------------------------------------------------

entity Tables is

--------------------------------------------------------------------------------
-- Entity: generics
--------------------------------------------------------------------------------
  generic (
    C_PAGE_AWIDTH       : integer := 20;
    C_PAGETABLE_AWIDTH  : integer := 6;
    C_KEYTABLE_AWIDTH   : integer := 6;
    C_KEYSIZE           : integer := 128;
    C_ANCILLARY_DWIDTH  : integer := 128);

--------------------------------------------------------------------------------
-- Entity: ports
--------------------------------------------------------------------------------
  port (
    Clk                         : in  std_logic;
    Reset                       : in  std_logic;
    iPTCtoPT_Req                : in  std_logic;
    oPTCtoPT_Ack                : out std_logic;
    iPTCtoPT_PageBase           : in  std_logic_vector(0 to C_PAGE_AWIDTH-1);
```

```
    iPTCtoPT_KeyIndex          : in  std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
    iPTCtoPT_PageIndex         : in  std_logic_vector(0 to C_PAGETABLE_AWIDTH-1);
    iPTCtoPT_Ancillary         : in  std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
    iKTCtoKT_Req               : in  std_logic;
    oKTCtoKT_Ack               : out std_logic;
    iKTCtoKT_KeyIndex          : in  std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
    iKTCtoKT_Key               : in  std_logic_vector(0 to C_KEYSIZE-1);
    iTableReq                  : in  std_logic;
    oTableSearchDone           : out std_logic;
    iPageAddress               : in  std_logic_vector(0 to C_PAGE_AWIDTH-1);
    oPageEncrypted             : out std_logic;
    oPageKey                   : out std_logic_vector(0 to C_KEYSIZE-1);
    oPageAncillary             : out std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
    oPerformance_TableWrite    : out std_logic;
    oPerformance_EncTableWrite : out std_logic);

end Tables;

-------------------------------------------------------------------------------
-- Architecture section
-------------------------------------------------------------------------------

architecture arch of Tables is

  -----------------------------------------------------------------------------
  -- Component declarations
  -----------------------------------------------------------------------------
  component cam20bx64
    port (
      clk        : in  std_logic;
      cmp_din    : in  std_logic_vector(19 downto 0);
      din        : in  std_logic_vector(19 downto 0);
      we         : in  std_logic;
      wr_addr    : in  std_logic_vector(5 downto 0);
      busy       : out std_logic;
      match      : out std_logic;
      match_addr : out std_logic_vector(5 downto 0));
  end component;

  component ram128bx64
    port (
      addra : in  std_logic_vector(5 downto 0);
      addrb : in  std_logic_vector(5 downto 0);
      clka  : in  std_logic;
      clkb  : in  std_logic;
      dinb  : in  std_logic_vector(127 downto 0);
      douta : out std_logic_vector(127 downto 0);
      web   : in  std_logic);
  end component;

  -----------------------------------------------------------------------------
  -- Type declarations
  -----------------------------------------------------------------------------
  type PTCtoPT_StateType is (ST_PTCtoPT_Write, ST_PTCtoPT_WaitReqActive,
  ST_PTCtoPT_WaitReqInactive);
  type KTCtoKT_StateType is (ST_KTCtoKT_Write, ST_KTCtoKT_WaitReqActive,
  ST_KTCtoKT_WaitReqInactive);

  -----------------------------------------------------------------------------
  -- Internal constant declarations
  -----------------------------------------------------------------------------
  constant NULLADDR : std_logic_vector(0 to C_KEYTABLE_AWIDTH-1) := (others => '0');

  -----------------------------------------------------------------------------
  -- Internal signal declarations
```

```
  ---------------------------------------------------------------------------
  -- Signals: PageTable handshake signals
  signal sClk_PTCtoPT_CurrState : PTCtoPT_StateType := ST_PTCtoPT_WaitReqActive;
  signal sPTCtoPT_NextState     : PTCtoPT_StateType := ST_PTCtoPT_WaitReqActive;
  signal sClk_PTCtoPT_Req_d1    : std_logic;
  signal sClk_PTCtoPT_Req_d2    : std_logic;
  signal sClk_PTCtoPT_Ack       : std_logic;

  -- Signals: KeyTable handshake signals
  signal sClk_KTCtoKT_CurrState : KTCtoKT_StateType := ST_KTCtoKT_WaitReqActive;
  signal sKTCtoKT_NextState     : KTCtoKT_StateType := ST_KTCtoKT_WaitReqActive;
  signal sClk_KTCtoKT_Req_d1    : std_logic;
  signal sClk_KTCtoKT_Req_d2    : std_logic;
  signal sClk_KTCtoKT_Ack       : std_logic;

  -- Table control signals
  signal sClk_PTWrite : std_logic;
  signal sClk_KTWrite : std_logic;

  signal sMatchAddr : std_logic_vector(0 to C_PAGETABLE_AWIDTH-1);
  signal sKeyAddr   : std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
  signal sKey       : std_logic_vector(0 to C_KEYSIZE-1);
  signal sAncillary : std_logic_vector(0 to C_ANCILLARY_DWIDTH-1);
  signal sClk_Timer : std_logic_vector(0 to 1);
  signal sClk_Enc   : std_logic;

---------------------------------------------------------------------------
-- Begin architecture
---------------------------------------------------------------------------
begin


  ---------------------------------------------------------------------------
  -- Component instantiations
  ---------------------------------------------------------------------------

  -- Content Addressable Memory: 64x128-bit
  cam20bx64_1 : cam20bx64
    port map (
      clk       => Clk,
      cmp_din   => iPageAddress,
      din       => iPTCtoPT_PageBase,
      we        => sClk_PTWrite,
      wr_addr   => iPTCtoPT_PageIndex,
      busy      => open,
      match     => open,
      match_addr => sMatchAddr);

  -- Key Address Table: 64x6-bit
  ram6bx64_1 : for i in 0 to 5 generate
  begin
    ram64x1d_inst : ram64x1d
      port map (
        DPO   => sKeyAddr(i),
        SPO   => open,
        A0    => iPTCtoPT_PageIndex(0),
        A1    => iPTCtoPT_PageIndex(1),
        A2    => iPTCtoPT_PageIndex(2),
        A3    => iPTCtoPT_PageIndex(3),
        A4    => iPTCtoPT_PageIndex(4),
        A5    => iPTCtoPT_PageIndex(5),
        D     => iPTCtoPT_KeyIndex(i),
        DPRA0 => sMatchAddr(0),
        DPRA1 => sMatchAddr(1),
```

```
      DPRA2 => sMatchAddr(2),
      DPRA3 => sMatchAddr(3),
      DPRA4 => sMatchAddr(4),
      DPRA5 => sMatchAddr(5),
      WCLK  => Clk,
      WE    => sClk_PTWrite
      );
end generate;

-- Key Table: 64x128-bit
ram128bx64_1 : ram128bx64
  port map (
    addra => sKeyAddr,
    addrb => iKTCtoKT_KeyIndex,
    clka  => Clk,
    clkb  => Clk,
    dinb  => iKTCtoKT_Key,
    douta => sKey,
    web   => sClk_KTWrite);

-- Ancillary Data Table: 64x128-bit
ram128bx64_2 : ram128bx64
  port map (
    addra => sMatchAddr,
    addrb => iPTCtoPT_PageIndex,
    clka  => Clk,
    clkb  => Clk,
    dinb  => iPTCtoPT_Ancillary,
    douta => sAncillary,
    web   => sClk_PTWrite);

-------------------------------------------------------------------------------
-- Look-up timers
-------------------------------------------------------------------------------
TimerProc : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_Timer   <= (others => '0');
    sClk_Enc     <= '0';
  elsif rising_edge(Clk) then
    sClk_Timer(0) <= iTableReq;
    sClk_Timer(1) <= sClk_Timer(0);
    if (sKeyAddr /= NULLADDR and sClk_Timer(0) = '1') then
      sClk_Enc <= '1';
    else
      sClk_Enc <= '0';
    end if;
  end if;
end process TimerProc;

-- Map internal signals to timing interface
oTableSearchDone <= sClk_Timer(1);    -- Search takes 1 clock cycle
oPageEncrypted   <= sClk_Enc;
oPageKey         <= sKey;
oPageAncillary   <= sAncillary;


-------------------------------------------------------------------------------
-- PageTable Interface handshaking (receive)
-------------------------------------------------------------------------------
-- Stabalize Acknowledgement signal from PTC unit
PTC_Meta_Ack : process (Clk, Reset) is
begin
  if Reset = '1' then
```

```
    sClk_PTCtoPT_Req_d1 <= '0';
    sClk_PTCtoPT_Req_d2 <= '0';
  elsif rising_edge(Clk) then
    sClk_PTCtoPT_Req_d1 <= iPTCtoPT_Req;
    sClk_PTCtoPT_Req_d2 <= sClk_PTCtoPT_Req_d1;
  end if;
end process PTC_Meta_Ack;

-- Protocol FSM Sequential process
PTCtoPT_Seq_Ifc : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_PTCtoPT_CurrState     <= ST_PTCtoPT_WaitReqActive;
    sClk_PTCtoPT_Ack           <= '0';
    sClk_PTWrite               <= '0';
    oPerformance_TableWrite    <= '0';
    oPerformance_EncTableWrite <= '0';
  elsif rising_edge(Clk) then
    sClk_PTCtoPT_CurrState <= sPTCtoPT_NextState;
    case sClk_PTCtoPT_CurrState is
      when ST_PTCtoPT_WaitReqActive =>
        oPerformance_TableWrite    <= '0';
        oPerformance_EncTableWrite <= '0';
        sClk_PTCtoPT_Ack           <= '0';
        sClk_PTWrite               <= sClk_PTCtoPT_Req_d2;
      when ST_PTCtoPT_Write =>
        if iPTCtoPT_KeyIndex /= "000000" then
          oPerformance_EncTableWrite <= '1';
        end if;
        oPerformance_TableWrite <= '1';
        sClk_PTCtoPT_Ack        <= '0';
        sClk_PTWrite            <= '1';
      when ST_PTCtoPT_WaitReqInactive =>
        oPerformance_TableWrite    <= '0';
        oPerformance_EncTableWrite <= '0';
        sClk_PTCtoPT_Ack           <= '1';
        sClk_PTWrite               <= '0';
      when others =>
        oPerformance_TableWrite    <= '0';
        oPerformance_EncTableWrite <= '0';
        sClk_PTCtoPT_Ack           <= '0';
        sClk_PTWrite               <= '0';
    end case;
  end if;
end process PTCtoPT_Seq_Ifc;

-- Protocol FSM Combinational process
PTCtoPT_Cmb_Ifc : process (sClk_PTCtoPT_CurrState, sClk_PTCtoPT_Req_d2) is
begin
  case sClk_PTCtoPT_CurrState is
    when ST_PTCtoPT_WaitReqActive =>
      if (sClk_PTCtoPT_Req_d2 = '1') then
        sPTCtoPT_NextState <= ST_PTCtoPT_Write;
      else
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitReqActive;
      end if;
    when ST_PTCtoPT_Write =>
      sPTCtoPT_NextState <= ST_PTCtoPT_WaitReqInactive;
    when ST_PTCtoPT_WaitReqInactive =>
      if (sClk_PTCtoPT_Req_d2 = '0') then
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitReqActive;
      else
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitReqInactive;
      end if;
```

```
    when others =>
      sPTCtoPT_NextState <= ST_PTCtoPT_WaitReqActive;
  end case;
end process PTCtoPT_Cmb_Ifc;

-- Map internal signals to PTC-PT Interface
oPTCtoPT_Ack <= sClk_PTCtoPT_Ack;

-------------------------------------------------------------------------------
-- KeyTable Interface handshaking (receive)
-------------------------------------------------------------------------------

-- Stabalize Acknowledgement signal from KTC unit
KTC_Meta_Ack : process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_KTCtoKT_Req_d1 <= '0';
    sClk_KTCtoKT_Req_d2 <= '0';
  elsif rising_edge(Clk) then
    sClk_KTCtoKT_Req_d1 <= iKTCtoKT_Req;
    sClk_KTCtoKT_Req_d2 <= sClk_KTCtoKT_Req_d1;
  end if;
end process KTC_Meta_Ack;

-- Protocol FSM Sequential process
KTCtoKT_Seq_Ifc : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_KTCtoKT_CurrState <= ST_KTCtoKT_WaitReqActive;
    sClk_KTCtoKT_Ack       <= '0';
    sClk_KTWrite           <= '0';
  elsif rising_edge(Clk) then
    sClk_KTCtoKT_CurrState <= sKTCtoKT_NextState;
    case sClk_KTCtoKT_CurrState is
      when ST_KTCtoKT_WaitReqActive =>
        sClk_KTCtoKT_Ack <= '0';
        sClk_KTWrite     <= sClk_KTCtoKT_Req_d2;
      when ST_KTCtoKT_Write =>
        sClk_KTCtoKT_Ack <= '0';
        sClk_KTWrite     <= '1';
      when ST_KTCtoKT_WaitReqInactive =>
        sClk_KTCtoKT_Ack <= '1';
        sClk_KTWrite     <= '0';
      when others =>
        sClk_KTCtoKT_Ack <= '0';
        sClk_KTWrite     <= '0';
    end case;
  end if;
end process KTCtoKT_Seq_Ifc;

-- Protocol FSM Combinational process
KTCtoKT_Cmb_Ifc : process (sClk_KTCtoKT_CurrState, sClk_KTCtoKT_Req_d2) is
begin
  case sClk_KTCtoKT_CurrState is
    when ST_KTCtoKT_WaitReqActive =>
      if (sClk_KTCtoKT_Req_d2 = '1') then
        sKTCtoKT_NextState <= ST_KTCtoKT_Write;
      else
        sKTCtoKT_NextState <= ST_KTCtoKT_WaitReqActive;
      end if;
    when ST_KTCtoKT_Write =>
      sKTCtoKT_NextState <= ST_KTCtoKT_WaitReqInactive;
    when ST_KTCtoKT_WaitReqInactive =>
      if (sClk_KTCtoKT_Req_d2 = '0') then
```

```
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitReqActive;
        else
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitReqInactive;
        end if;
      when others =>
        sKTCtoKT_NextState <= ST_KTCtoKT_WaitReqActive;
    end case;
  end process KTCtoKT_Cmb_Ifc;

  -- Map internal signals to KTC-kT Interface
  oKTCtoKT_Ack <= sClk_KTCtoKT_Ack;

end arch;
```

D.1.7 PageTableControl.vhd

```
-------------------------------------------------------------------------------
-- Title      : Page Table Control Unit
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : PageTableControl.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-03-07
-- Last update: 2005-07-15
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Unit for controlling the Page tables in the Table unit.
--              This unit is controled by a standard Control Interface.
--              Clock domain synchronization is used when writing to Table
--              unit.
-- Assumptions: Bus interface control unit is in the same clock domain.
--              Table unit is in a different clock domain.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author   Description
-- 2005-07-09  amahar   Switched state machine for better inference with synth.
--                      Used StateType declaration instead of hard coding
-- 2005-06-25  amahar   Removed KTC/PTC Communication. not cohesive.
-- 2005-03-31  amahar   Finalized KTCtoPTC and PTCtoKTC cross clock
--                      domain synchronization
-- 2005-03-25  amahar   Relocated the PageTable write-enable signal
--                      to its own dedicated process
-- 2005-03-25  amahar   Added comments, switched to Zero bus padding
--                      on reads
-- 2005-03-15  amahar   Fixed high/low location of sku control/status
--                      control now in upper 16-bits
-- 2005-03-09  amahar   Switched to address/write_enable interface
-- 2005-03-08  amahar   Added core functionality
-- 2005-03-07  amahar   Created
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity PageTableControl is

-------------------------------------------------------------------------------
-- Entity: generics
-------------------------------------------------------------------------------
  generic (
    CONTROL_DWIDTH      : integer := 32;
    CONTROL_AWIDTH      : integer := 3;
    C_PAGE_AWIDTH       : integer := 20;
    C_PAGETABLE_AWIDTH  : integer := 6;
    C_KEYTABLE_AWIDTH   : integer := 6;
    C_ANCILLARY_DWIDTH  : integer := 128);

-------------------------------------------------------------------------------
-- Entity: ports
-------------------------------------------------------------------------------
  port (
    Clk                 : in  std_logic;
    Reset               : in  std_logic;
    iControl_Addr       : in  std_logic_vector(0 to CONTROL_AWIDTH-1);
```

```
    iControl_Req        : in  std_logic;
    oControl_Ack        : out std_logic;
    oControl_Wait       : out std_logic;
    iControl_RNW        : in  std_logic;
    iControl_BE         : in  std_logic_vector(0 to (CONTROL_DWIDTH/8)-1);
    iControl_WrBus      : in  std_logic_vector(0 to CONTROL_DWIDTH-1);
    oControl_RdBus      : out std_logic_vector(0 to CONTROL_DWIDTH-1);
    oPTCtoPT_Req        : out std_logic;
    iPTCtoPT_Ack        : in  std_logic;
    oPTCtoPT_PageBase   : out std_logic_vector(0 to C_PAGE_AWIDTH-1);
    oPTCtoPT_KeyIndex   : out std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
    oPTCtoPT_PageIndex  : out std_logic_vector(0 to C_PAGETABLE_AWIDTH-1);
    oPTCtoPT_Ancillary  : out std_logic_vector(0 to C_ANCILLARY_DWIDTH-1));

end PageTableControl;

-------------------------------------------------------------------------------
-- Architecture Section
-------------------------------------------------------------------------------

architecture arch of PageTableControl is

  type PTCtoPT_StateType is (ST_PTCtoPT_WaitWrite, ST_PTCtoPT_WaitAckActive,
  ST_PTCtoPT_WaitAckInactive);

  -----------------------------------------------------------------------------
  -- Internal constant declarations
  -----------------------------------------------------------------------------
  -- Constants: Zero Pads
  constant PAGE_ZEROS      : std_logic_vector(0 to CONTROL_DWIDTH-C_PAGE_AWIDTH-1)      := (
  others => '0');
  constant KEYINDEX_ZEROS  : std_logic_vector(0 to CONTROL_DWIDTH-C_KEYTABLE_AWIDTH-1)  := (
  others => '0');
  constant PAGEINDEX_ZEROS : std_logic_vector(0 to CONTROL_DWIDTH-C_PAGETABLE_AWIDTH-1) := (
  others => '0');

  -----------------------------------------------------------------------------
  -- Internal signal declarations
  -----------------------------------------------------------------------------
  -- Signals: Control Unit Registers
  signal sClk_PageBase    : std_logic_vector(0 to C_PAGE_AWIDTH-1);
  signal sClk_KeyIndex    : std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
  signal sClk_PageIndex   : std_logic_vector(0 to C_PAGETABLE_AWIDTH-1);
  signal sClk_Ancillary0  : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Ancillary1  : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Ancillary2  : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Ancillary3  : std_logic_vector(0 to CONTROL_DWIDTH-1);

  -- Signals: PageTable handshake signals
  signal sClk_PTCtoPT_CurrState : PTCtoPT_StateType := ST_PTCtoPT_WaitWrite;
  signal sPTCtoPT_NextState     : PTCtoPT_StateType := ST_PTCtoPT_WaitWrite;
  signal sClk_PTCtoPT_Req       : std_logic;
  signal sClk_PTCtoPT_Ack_d1    : std_logic;
  signal sClk_PTCtoPT_Ack_d2    : std_logic;

  -- Signals: Control Interface handshake signals
  signal sClk_Control_PTCtoPT_Ack : std_logic;
  signal sClk_Control_Ack         : std_logic;
  signal sClk_Control_Wait        : std_logic;


-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------

begin
```

```
-------------------------------------------------------------------------------
-- PageTable Interface handshaking (transmit)
-------------------------------------------------------------------------------

-- Stabalize Acknowledgement signal from Table unit
Meta_Ack : process (Clk, Reset) is
begin
  if Reset = '1' then
    sClk_PTCtoPT_Ack_d1 <= '0';
    sClk_PTCtoPT_Ack_d2 <= '0';
  elsif rising_edge(Clk) then
    sClk_PTCtoPT_Ack_d1 <= iPTCtoPT_Ack;
    sClk_PTCtoPT_Ack_d2 <= sClk_PTCtoPT_Ack_d1;
  end if;
end process Meta_Ack;

-- Protocol FSM Sequential process
PTCtoPT_Seq_Ifc : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_PTCtoPT_CurrState   <= ST_PTCtoPT_WaitWrite;
    sClk_PTCtoPT_Req         <= '0';
    sClk_Control_PTCtoPT_Ack <= '0';
  elsif rising_edge(Clk) then
    sClk_PTCtoPT_CurrState <= sPTCtoPT_NextState;
    case sClk_PTCtoPT_CurrState is
      when ST_PTCtoPT_WaitWrite =>
        sClk_PTCtoPT_Req         <= '0';
        sClk_Control_PTCtoPT_Ack <= '0';
      when ST_PTCtoPT_WaitAckActive =>
        sClk_PTCtoPT_Req         <= '1';
        sClk_Control_PTCtoPT_Ack <= '0';
      when ST_PTCtoPT_WaitAckInactive =>
        sClk_PTCtoPT_Req         <= '0';
        sClk_Control_PTCtoPT_Ack <= '1';
      when others =>
        sClk_PTCtoPT_Req         <= '0';
        sClk_Control_PTCtoPT_Ack <= '0';
    end case;
  end if;
end process PTCtoPT_Seq_Ifc;

-- Protocol FSM Combinational process
PTCtoPT_Cmb_Ifc : process (iControl_Addr, iControl_RNW, iControl_Req,
                           sClk_PTCtoPT_Ack_d2, sClk_PTCtoPT_CurrState) is
begin
  case sClk_PTCtoPT_CurrState is
    when ST_PTCtoPT_WaitWrite =>
      if (iControl_RNW = '0' and iControl_Addr = "010" and iControl_Req = '1') then
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitAckActive;
      else
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitWrite;
      end if;
    when ST_PTCtoPT_WaitAckActive =>
      if (sClk_PTCtoPT_Ack_d2 = '1') then
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitAckInactive;
      else
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitAckActive;
      end if;
    when ST_PTCtoPT_WaitAckInactive =>
      if (sClk_PTCtoPT_Ack_d2 = '0') then
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitWrite;
      else
```

```
        sPTCtoPT_NextState <= ST_PTCtoPT_WaitAckInactive;
      end if;
    when others =>
      sPTCtoPT_NextState <= ST_PTCtoPT_WaitWrite;
  end case;
end process PTCtoPT_Cmb_Ifc;

-- Map internal signals to PTC-PT Interface
oPTCtoPT_Req       <= sClk_PTCtoPT_Req;
oPTCtoPT_PageBase  <= sClk_PageBase;
oPTCtoPT_PageIndex <= sClk_PageIndex;
oPTCtoPT_KeyIndex  <= sClk_KeyIndex;
oPTCtoPT_Ancillary <= sClk_Ancillary3 & sClk_Ancillary2 & sClk_Ancillary1 &
sClk_Ancillary0;


-------------------------------------------------------------------------------
-- Control Interface handshaking
-------------------------------------------------------------------------------

ControlAck : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_Control_Ack  <= '0';
    sClk_Control_Wait <= '0';
  elsif rising_edge(Clk) then
    if (iControl_Req = '1' and iControl_RNW = '1') then      -- Read request
      sClk_Control_Ack  <= '1';
      sClk_Control_Wait <= '0';
    elsif (iControl_Req = '1' and iControl_RNW = '0') then  -- Write request
      if (iControl_Addr = "010") then  -- PageTable Write
        sClk_Control_Ack  <= sClk_Control_PTCtoPT_Ack;
        sClk_Control_Wait <= '1';
      else                             -- Regular register write
        sClk_Control_Ack  <= '1';
        sClk_Control_Wait <= '0';
      end if;
    else                               -- No request
      sClk_Control_Ack  <= '0';
      sClk_Control_Wait <= '0';
    end if;
  end if;
end process ControlAck;

-- Map internal signals to PTC-Control Interface
oControl_Ack  <= sClk_Control_Ack;
oControl_Wait <= sClk_Control_Wait;

-------------------------------------------------------------------------------
-- Register handling
-------------------------------------------------------------------------------

-- Write interface
RegisterWrite : process (Clk, Reset)
begin
  if (Reset = '1') then
    sClk_PageBase   <= (others => '0');
    sClk_KeyIndex   <= (others => '0');
    sClk_PageIndex  <= (others => '0');
    sClk_Ancillary0 <= (others => '0');
    sClk_Ancillary1 <= (others => '0');
    sClk_Ancillary2 <= (others => '0');
    sClk_Ancillary3 <= (others => '0');
  elsif rising_edge(Clk) then
```

```
    if (iControl_RNW = '0' and iControl_Req = '1') then
      case iControl_Addr(0 to 2) is
        when "000" =>
          if (iControl_BE(0) = '1') then
            sClk_PageBase(0 to 7) <= iControl_WrBus(0 to 7);
          end if;
          if (iControl_BE(1) = '1') then
            sClk_PageBase(8 to 15) <= iControl_WrBus(8 to 15);
          end if;
          if (iControl_BE(2) = '1') then
            sClk_PageBase(16 to C_PAGE_AWIDTH-1) <= iControl_WrBus(16 to C_PAGE_AWIDTH-1);
          end if;
        when "001" =>
          if (iControl_BE(3) = '1') then
            sClk_KeyIndex <= iControl_WrBus(CONTROL_DWIDTH-C_KEYTABLE_AWIDTH to
            CONTROL_DWIDTH-1);
          end if;
        when "010" =>
          if (iControl_BE(3) = '1') then
            sClk_PageIndex <= iControl_WrBus(CONTROL_DWIDTH-C_PAGETABLE_AWIDTH to
            CONTROL_DWIDTH-1);
          end if;
        when "100" =>
          for i in 0 to (CONTROL_DWIDTH/8)-1 loop
            if (iControl_BE(i) = '1') then
              sClk_Ancillary0(i*8 to i*8+7) <= iControl_WrBus(i*8 to i*8+7);
            end if;
          end loop;
        when "101" =>
          for i in 0 to (CONTROL_DWIDTH/8)-1 loop
            if (iControl_BE(i) = '1') then
              sClk_Ancillary1(i*8 to i*8+7) <= iControl_WrBus(i*8 to i*8+7);
            end if;
          end loop;
        when "110" =>
          for i in 0 to (CONTROL_DWIDTH/8)-1 loop
            if (iControl_BE(i) = '1') then
              sClk_Ancillary2(i*8 to i*8+7) <= iControl_WrBus(i*8 to i*8+7);
            end if;
          end loop;
        when "111" =>
          for i in 0 to (CONTROL_DWIDTH/8)-1 loop
            if (iControl_BE(i) = '1') then
              sClk_Ancillary3(i*8 to i*8+7) <= iControl_WrBus(i*8 to i*8+7);
            end if;
          end loop;
        when others => null;
      end case;
    end if;
  end if;
end process RegisterWrite;

-- Read interface
RegisterRead : process (iControl_Addr, sClk_Ancillary0,
                        sClk_Ancillary1, sClk_Ancillary2, sClk_Ancillary3,
                        sClk_KeyIndex, sClk_PageBase, sClk_PageIndex)
begin
  case iControl_Addr(0 to 2) is
    when "000"  => oControl_RdBus <= sClk_PageBase & PAGE_ZEROS;
    when "001"  => oControl_RdBus <= KEYINDEX_ZEROS & sClk_KeyIndex;
    when "010"  => oControl_RdBus <= PAGEINDEX_ZEROS & sClk_PageIndex;
    when "100"  => oControl_RdBus <= sClk_Ancillary0;
    when "101"  => oControl_RdBus <= sClk_Ancillary1;
    when "110"  => oControl_RdBus <= sClk_Ancillary2;
```

```
      when "111"  => oControl_RdBus <= sClk_Ancillary3;
      when others => oControl_RdBus <= (others => '0');
    end case;
  end process RegisterRead;

end arch;
```

D.1.8 KeyTableControl.vhd

```
-------------------------------------------------------------------------------
-- Title      : Key Table Control Unit
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : KeyTableControl.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-03-07
-- Last update: 2005-07-10
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Unit for controlling the Key tables of the Table unit.
--              This unit is controled by a standard Control Interface.
--              Clock domain synchronization is used when writing to Table
--              unit.
-- Assumptions: Bus interface control unit is in the same clock domain.
--              Table unit is in a different clock domain.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-07-09  amahar  Switched state machine for better inference with synth.
--                     Used StateType declaration instead of hard coding
-- 2005-06-25  amahar  Removed PTC/KTC handshaking.  not cohesive.
-- 2005-03-31  amahar  Finalized KTCtoPTC and PTCtoKTC cross clock
--                     domain synchronization
-- 2005-03-25  amahar  Relocated the PageTable write-enable signal
--                     to its own dedicated process
-- 2005-03-25  amahar  Added comments, switched to Zero bus padding
--                     on reads
-- 2005-03-15  amahar  Fixed high/low location of sku control/status
--                     control now in upper 16-bits
-- 2005-03-09  amahar  Switched to address/write_enable interface
-- 2005-03-08  amahar  Added core functionality
-- 2005-03-07  amahar  Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity KeyTableControl is

-------------------------------------------------------------------------------
-- Entity: generics
-------------------------------------------------------------------------------
  generic (
    CONTROL_DWIDTH     : integer := 32;
    CONTROL_AWIDTH     : integer := 8;
    C_KEYTABLE_AWIDTH : integer := 6;
    C_KEYSIZE          : integer := 128);

-------------------------------------------------------------------------------
-- Entity: ports
-------------------------------------------------------------------------------
  port (
    Clk                : in  std_logic;
    Reset              : in  std_logic;
    iControl_Addr      : in  std_logic_vector(0 to 2);
    iControl_Req       : in  std_logic;
    oControl_Ack       : out std_logic;
```

```
    oControl_Wait      : out std_logic;
    iControl_RNW       : in  std_logic;
    iControl_BE        : in  std_logic_vector(0 to (CONTROL_DWIDTH/8)-1);
    iControl_WrBus     : in  std_logic_vector(0 to CONTROL_DWIDTH-1);
    oControl_RdBus     : out std_logic_vector(0 to CONTROL_DWIDTH-1);
    oKTCtoKT_Req       : out std_logic;
    iKTCtoKT_Ack       : in  std_logic;
    oKTCtoKT_KeyIndex  : out std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
    oKTCtoKT_Key       : out std_logic_vector(0 to C_KEYSIZE-1));

end entity KeyTableControl;

-------------------------------------------------------------------------------
-- Architecture section
-------------------------------------------------------------------------------

architecture arch of KeyTableControl is

  -----------------------------------------------------------------------------
  -- Type declarations
  -----------------------------------------------------------------------------
  type KTCtoKT_StateType is (ST_KTCtoKT_WaitWrite, ST_KTCtoKT_WaitAckActive,
  ST_KTCtoKT_WaitAckInactive);

  -----------------------------------------------------------------------------
  -- Internal constant declarations
  -----------------------------------------------------------------------------
  -- Constants: Zero Pads
  constant ZEROS_KeyIndex : std_logic_vector(0 to CONTROL_DWIDTH-C_KEYTABLE_AWIDTH-1) := (
  others => '0');

  -----------------------------------------------------------------------------
  -- Internal signal declarations
  -----------------------------------------------------------------------------
  -- Signals: Control Unit Registers
  signal sClk_KeyIndex : std_logic_vector(0 to C_KEYTABLE_AWIDTH-1);
  signal sClk_Key0     : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Key1     : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Key2     : std_logic_vector(0 to CONTROL_DWIDTH-1);
  signal sClk_Key3     : std_logic_vector(0 to CONTROL_DWIDTH-1);

  -- Signals: KeyTable handshake signals
  signal sClk_KTCtoKT_CurrState : KTCtoKT_StateType := ST_KTCtoKT_WaitWrite;
  signal sKTCtoKT_NextState     : KTCtoKT_StateType := ST_KTCtoKT_WaitWrite;
  signal sClk_KTCtoKT_Req       : std_logic;
  signal sClk_KTCtoKT_Ack_d1    : std_logic;
  signal sClk_KTCtoKT_Ack_d2    : std_logic;

  -- Signals: Control Interface handshake signals
  signal sClk_Control_KTCtoKT_Ack : std_logic;
  signal sClk_Control_Ack         : std_logic;
  signal sClk_Control_Wait        : std_logic;

-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------

begin

  -----------------------------------------------------------------------------
  -- KeyTable Interface handshaking
  -----------------------------------------------------------------------------

  -- Stabalize Acknowledgement signal from Table unit
  Meta_Ack : process (Clk, Reset) is
  begin
```

```
    if Reset = '1' then
      sClk_KTCtoKT_Ack_d1 <= '0';
      sClk_KTCtoKT_Ack_d2 <= '0';
    elsif rising_edge(Clk) then
      sClk_KTCtoKT_Ack_d1 <= iKTCtoKT_Ack;
      sClk_KTCtoKT_Ack_d2 <= sClk_KTCtoKT_Ack_d1;
    end if;
  end process Meta_Ack;

  -- Protocol FSM Sequential process
  KTCtoKT_Seq_Ifc : process (Clk, Reset) is
  begin
    if (Reset = '1') then
      sClk_KTCtoKT_CurrState   <= ST_KTCtoKT_WaitWrite;
      sClk_KTCtoKT_Req         <= '0';
      sClk_Control_KTCtoKT_Ack <= '0';
    elsif rising_edge(Clk) then
      sClk_KTCtoKT_CurrState <= sKTCtoKT_NextState;
      case sClk_KTCtoKT_CurrState is
        when ST_KTCtoKT_WaitWrite =>
          sClk_KTCtoKT_Req         <= '0';
          sClk_Control_KTCtoKT_Ack <= '0';
        when ST_KTCtoKT_WaitAckActive =>
          sClk_KTCtoKT_Req         <= '1';
          sClk_Control_KTCtoKT_Ack <= '0';
        when ST_KTCtoKT_WaitAckInactive =>
          sClk_KTCtoKT_Req         <= '0';
          sClk_Control_KTCtoKT_Ack <= '1';
        when others =>
          sClk_KTCtoKT_Req         <= '0';
          sClk_Control_KTCtoKT_Ack <= '0';
      end case;
    end if;
  end process KTCtoKT_Seq_Ifc;

  -- Protocol FSM Combinational process
  KTCtoKT_Cmb_Ifc : process (iControl_Addr, iControl_RNW, iControl_Req,
                             sClk_KTCtoKT_Ack_d2, sClk_KTCtoKT_CurrState) is
  begin
    case sClk_KTCtoKT_CurrState is
      when ST_KTCtoKT_WaitWrite =>
        if (iControl_RNW = '0' and iControl_Addr = "000" and iControl_Req = '1') then
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitAckActive;
        else
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitWrite;
        end if;
      when ST_KTCtoKT_WaitAckActive =>
        if (sClk_KTCtoKT_Ack_d2 = '1') then
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitAckInactive;
        else
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitAckActive;
        end if;
      when ST_KTCtoKT_WaitAckInactive =>
        if (sClk_KTCtoKT_Ack_d2 = '0') then
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitWrite;
        else
          sKTCtoKT_NextState <= ST_KTCtoKT_WaitAckInactive;
        end if;
      when others =>
        sKTCtoKT_NextState <= ST_KTCtoKT_WaitWrite;
    end case;
  end process KTCtoKT_Cmb_Ifc;

  -- Map internal signals to KTC-KT Interface
```

```
oKTCtoKT_Req      <= sClk_KTCtoKT_Req;
oKTCtoKT_KeyIndex <= sClk_KeyIndex;
oKTCtoKT_Key      <= sClk_Key3 & sClk_Key2 & sClk_Key1 & sClk_Key0;

-------------------------------------------------------------------------------
-- Control Interface handshaking
-------------------------------------------------------------------------------

-- purpose: Generate the acknowledge signal back to the control interface
-- type:    sequential
-- comments:
ControlAck : process (Clk, Reset) is
begin
  if (Reset = '1') then
    sClk_Control_Ack  <= '0';
    sClk_Control_Wait <= '0';
  elsif rising_edge(Clk) then
    if (iControl_Req = '1' and iControl_RNW = '1') then
      -- Read request
      sClk_Control_Ack  <= '1';
      sClk_Control_Wait <= '0';
    elsif (iControl_Req = '1' and iControl_RNW = '0') then
      -- Write Request
      if (iControl_Addr = "000") then  -- KeyTable Write
        sClk_Control_Ack  <= sClk_Control_KTCtoKT_Ack;
        sClk_Control_Wait <= '1';
      else
        sClk_Control_Ack  <= '1';
        sClk_Control_Wait <= '0';
      end if;
    else
      -- No request
      sClk_Control_Ack  <= '0';
      sClk_Control_Wait <= '0';
    end if;
  end if;
end process ControlAck;

-- Map internal signals to KTC-Control Interface
oControl_Ack  <= sClk_Control_Ack;
oControl_Wait <= sClk_Control_Wait;

-------------------------------------------------------------------------------
-- Register handling
-------------------------------------------------------------------------------

-- Write interface
RegisterWrite : process(Clk, Reset) is
begin
  if Reset = '1' then
    sClk_KeyIndex <= (others => '0');
    sClk_Key0     <= (others => '0');
    sClk_Key1     <= (others => '0');
    sClk_Key2     <= (others => '0');
    sClk_Key3     <= (others => '0');
  elsif rising_edge(Clk) then
    if (iControl_RNW = '0' and iControl_Req = '1') then
      case iControl_Addr is
        when "000" =>
          if (iControl_BE(3) = '1') then
            sClk_KeyIndex <= iControl_WrBus(CONTROL_DWIDTH-C_KEYTABLE_AWIDTH to
            CONTROL_DWIDTH-1);
          end if;
        when "100" =>
```

```
            for byte_index in 0 to (CONTROL_DWIDTH/8)-1 loop
              if (iControl_BE(byte_index) = '1') then
                sClk_Key0(byte_index*8 to byte_index*8+7) <= iControl_WrBus(byte_index*8 to
                byte_index*8+7);
              end if;
            end loop;
          when "101" =>
            for byte_index in 0 to (CONTROL_DWIDTH/8)-1 loop
              if (iControl_BE(byte_index) = '1') then
                sClk_Key1(byte_index*8 to byte_index*8+7) <= iControl_WrBus(byte_index*8 to
                byte_index*8+7);
              end if;
            end loop;
          when "110" =>
            for byte_index in 0 to (CONTROL_DWIDTH/8)-1 loop
              if (iControl_BE(byte_index) = '1') then
                sClk_Key2(byte_index*8 to byte_index*8+7) <= iControl_WrBus(byte_index*8 to
                byte_index*8+7);
              end if;
            end loop;
          when "111" =>
            for byte_index in 0 to (CONTROL_DWIDTH/8)-1 loop
              if (iControl_BE(byte_index) = '1') then
                sClk_Key3(byte_index*8 to byte_index*8+7) <= iControl_WrBus(byte_index*8 to
                byte_index*8+7);
              end if;
            end loop;
          when others => null;
        end case;
      end if;
    end if;
  end process RegisterWrite;

  -- Read interface
  RegisterRead : process(iControl_Addr, sClk_Key0, sClk_Key1, sClk_Key2, sClk_Key3,
                         sClk_KeyIndex) is
  begin
    case iControl_Addr is
      when "000"  => oControl_RdBus <= ZEROS_KeyIndex & sClk_KeyIndex;
      when "100"  => oControl_RdBus <= sClk_Key0;
      when "101"  => oControl_RdBus <= sClk_Key1;
      when "110"  => oControl_RdBus <= sClk_Key2;
      when "111"  => oControl_RdBus <= sClk_Key3;
      when others => oControl_RdBus <= (others => '0');
    end case;
  end process RegisterRead;

end arch;
```

D.1.9 OPB_ControlInterface.vhd

```
-------------------------------------------------------------------------------
-- Title      : OPB Control Interface
-- Project    : Secure Software
-------------------------------------------------------------------------------
-- File       : OPB_ControlInterface.vhd
-- Author     : Anthony Mahar <amahar@vt.edu>
-- Company    : Virginia Tech Configurable Computing Lab
-- Created    : 2005-03-15
-- Last update: 2005-07-15
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Translates OPB bus transaction requests to the generic internal
--              request/ack and RNW bus requests used by the internal
--              control units.
-------------------------------------------------------------------------------
-- Copyright (c) 2005
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Author  Description
-- 2005-03-23  amahar   removing "Write enable" signal and replaced with a
--                      request and RNW signal
-- 2005-03-22  amahar   Modified unit individual register selects to
--                      real 3-bit addresses.
-- 2005-03-15  amahar   Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
use proc_common_v2_00_a.ipif_pkg.all;

library opb_ipif_v3_01_b;
use opb_ipif_v3_01_b.all;

-------------------------------------------------------------------------------

entity OPB_ControlInterface is

    generic (
        C_OPB_AWIDTH    : integer          := 32;
        C_OPB_DWIDTH    : integer          := 32;
        C_BASEADDR      : std_logic_vector := X"FFFFFFFF";
        C_HIGHADDR      : std_logic_vector := X"00000000";
        CONTROL_AWIDTH : integer          := 3;
        CONTROL_DWIDTH : integer          := 32;
        C_FAMILY        : string           := "virtex2p");

    port (
        Clk             : in  std_logic;
        Reset           : in  std_logic;
        oSl_DBus        : out std_logic_vector(0 to C_OPB_DWIDTH-1);
        oSl_errAck      : out std_logic;
        oSl_retry       : out std_logic;
        oSl_toutSup     : out std_logic;
        oSl_xferAck     : out std_logic;
        iOPB_ABus       : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
        iOPB_BE         : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
        iOPB_DBus       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
        iOPB_RNW        : in  std_logic;
```

```
        iOPB_select     : in  std_logic;
        iOPB_seqAddr    : in  std_logic;
        oControl_Clk    : out std_logic;
        oControl_Reset  : out std_logic;
        oControl_Addr   : out std_logic_vector(0 to CONTROL_AWIDTH-1);
        oControl_Req    : out std_logic;
        iControl_Wait   : in  std_logic;
        iControl_Ack    : in  std_logic;
        oControl_RNW    : out std_logic;
        oControl_BE     : out std_logic_vector(0 to (CONTROL_DWIDTH/8)-1);
        oControl_WrBus  : out std_logic_vector(0 to CONTROL_DWIDTH-1);
        iControl_RdBus  : in  std_logic_vector(0 to CONTROL_DWIDTH-1));

end entity OPB_ControlInterface;

------------------------------------------------------------------------------

architecture arch of OPB_ControlInterface is

    ------------------------------------------------------------------------
    -- Internal constant declarations
    ------------------------------------------------------------------------
    constant ZERO_ADDR_PAD      : std_logic_vector(0 to 64-C_OPB_AWIDTH-1) := (others => '0')
    ;
    constant USER_NUM_ADDR_RNG : integer                                    := 1;

    constant ARD_ID_ARRAY : INTEGER_ARRAY_TYPE :=
        (
            0 => USER_01
            );

    constant ARD_ADDR_RANGE_ARRAY : SLV64_ARRAY_TYPE :=
        (
            ZERO_ADDR_PAD & C_BASEADDR,
            ZERO_ADDR_PAD & C_HIGHADDR
            );

    -- User address range 0 - Data Width
    constant ARD_DWIDTH_ARRAY : INTEGER_ARRAY_TYPE :=
        (
            0 => CONTROL_DWIDTH
            );
    -- User address range 0 - 1 CE
    constant ARD_NUM_CE_ARRAY : INTEGER_ARRAY_TYPE :=
        (
            0 => 1
            );
    -- Array of unique properties for each address range (none)
    constant ARD_DEPENDENT_PROPS_ARRAY : DEPENDENT_PROPS_ARRAY_TYPE :=
        (
            0 => (others => 0)
            );
    -- Array of IP interrupt modes (none)
    constant IP_INTR_MODE_ARRAY : INTEGER_ARRAY_TYPE :=
        (
            0 => 0
            );

    constant PIPELINE_MODEL     : integer := 5;
    constant DEV_BLK_ID         : integer := 0;
    constant DEV_MIR_ENABLE     : integer := 0;
    constant DEV_BURST_ENABLE   : integer := 0;
    constant INCLUDE_ADDR_CNTR  : integer := 0;
    constant INCLUDE_WR_BUF     : integer := 0;
```

```
    constant USER01_CS_INDEX    : integer := get_id_index(ARD_ID_ARRAY, USER_01);
    constant USER01_CE_INDEX    : integer := calc_start_ce_index(ARD_NUM_CE_ARRAY,
    USER01_CS_INDEX);

    ------------------------------------------------------------------------------
    -- Internal signal declarations
    ------------------------------------------------------------------------------
    signal ZERO_IP2Bus_Data         : std_logic_vector(0 to C_OPB_DWIDTH-1)              := (
    others => '0');
    signal ZERO_IP2Bus_PostedWrInh : std_logic_vector(0 to ARD_ID_ARRAY'length-1)       := (
    others => '1');
    signal ZERO_IP2RFIFO_Data       : std_logic_vector(0 to 31)                          := (
    others => '0');
    signal ZERO_WFIFO2IP_Data       : std_logic_vector(0 to 31)                          := (
    others => '0');
    signal ZERO_IP2Bus_IntrEvent    : std_logic_vector(0 to IP_INTR_MODE_ARRAY'length-1) := (
    others => '0');

    signal sBus2IP_Clk      : std_logic;
    signal sBus2IP_Reset    : std_logic;
    signal sBus2IP_Addr     : std_logic_vector(0 to C_OPB_AWIDTH-1);
    signal sBus2IP_Data     : std_logic_vector(0 to C_OPB_DWIDTH-1);
    signal sBus2IP_RNW      : std_logic;
    signal sBus2IP_BE       : std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    signal sBus2IP_CS       : std_logic_vector(0 to ARD_ID_ARRAY'length-1);
    signal sIP2Bus_Ack      : std_logic;
    signal sIP2Bus_toutSup  : std_logic;
    signal sIP2Bus_Data     : std_logic_vector(0 to C_OPB_DWIDTH-1);

begin

    ------------------------------------------------------------------------------
    -- Sanity check
    ------------------------------------------------------------------------------
    assert (C_OPB_DWIDTH = CONTROL_DWIDTH) report "ERROR: OPB Control Unit data width and
    Control Unit data width differ" severity warning;

    ------------------------------------------------------------------------------
    -- Constant Logic
    ------------------------------------------------------------------------------
    ZERO_IP2Bus_Data        <= (others => '0');
    ZERO_IP2Bus_PostedWrInh <= (others => '1');
    ZERO_IP2RFIFO_Data      <= (others => '0');
    ZERO_WFIFO2IP_Data      <= (others => '0');
    ZERO_IP2Bus_IntrEvent   <= (others => '0');

    ------------------------------------------------------------------------------
    -- Component instantiations
    ------------------------------------------------------------------------------
    OPB_IPIF_1 : entity opb_ipif_v3_01_b.opb_ipif
        generic map
        (
            C_ARD_ID_ARRAY              => ARD_ID_ARRAY,
            C_ARD_ADDR_RANGE_ARRAY      => ARD_ADDR_RANGE_ARRAY,
            C_ARD_DWIDTH_ARRAY          => ARD_DWIDTH_ARRAY,
            C_ARD_NUM_CE_ARRAY          => ARD_NUM_CE_ARRAY,
            C_ARD_DEPENDENT_PROPS_ARRAY => ARD_DEPENDENT_PROPS_ARRAY,
            C_PIPELINE_MODEL            => PIPELINE_MODEL,
            C_DEV_BLK_ID                => DEV_BLK_ID,
            C_DEV_MIR_ENABLE            => DEV_MIR_ENABLE,
            C_OPB_AWIDTH                => C_OPB_AWIDTH,
            C_OPB_DWIDTH                => C_OPB_DWIDTH,
            C_FAMILY                    => C_FAMILY,
            C_IP_INTR_MODE_ARRAY        => IP_INTR_MODE_ARRAY,
```

```
    C_DEV_BURST_ENABLE          => DEV_BURST_ENABLE,
    C_INCLUDE_ADDR_CNTR         => INCLUDE_ADDR_CNTR,
    C_INCLUDE_WR_BUF            => INCLUDE_WR_BUF)
port map
(
    OPB_select              => iOPB_select,
    OPB_DBus                => iOPB_DBus,
    OPB_ABus                => iOPB_ABus,
    OPB_BE                  => iOPB_BE,
    OPB_RNW                 => iOPB_RNW,
    OPB_seqAddr             => iOPB_seqAddr,
    Sln_DBus                => oSl_DBus,
    Sln_xferAck             => oSl_xferAck,
    Sln_errAck              => oSl_errAck,
    Sln_retry               => oSl_retry,
    Sln_toutSup             => oSl_toutSup,
    Bus2IP_CS               => sBus2IP_CS,
    Bus2IP_CE               => open,
    Bus2IP_RdCE             => open,
    Bus2IP_WrCE             => open,
    Bus2IP_Data             => sBus2IP_Data,
    Bus2IP_Addr             => sBus2IP_Addr,
    Bus2IP_AddrValid        => open,
    Bus2IP_BE               => sBus2IP_BE,
    Bus2IP_RNW              => sBus2IP_RNW,
    Bus2IP_Burst            => open,
    IP2Bus_Data             => sIP2Bus_Data,
    IP2Bus_Ack              => sIP2Bus_Ack,
    IP2Bus_AddrAck          => '0',
    IP2Bus_Error            => '0',
    IP2Bus_Retry            => '0',
    IP2Bus_ToutSup          => sIP2Bus_toutSup,
    IP2Bus_PostedWrInh      => ZERO_IP2Bus_PostedWrInh,
    IP2RFIFO_Data           => ZERO_IP2RFIFO_Data,
    IP2RFIFO_WrMark         => '0',
    IP2RFIFO_WrRelease      => '0',
    IP2RFIFO_WrReq          => '0',
    IP2RFIFO_WrRestore      => '0',
    RFIFO2IP_AlmostFull     => open,
    RFIFO2IP_Full           => open,
    RFIFO2IP_Vacancy        => open,
    RFIFO2IP_WrAck          => open,
    IP2WFIFO_RdMark         => '0',
    IP2WFIFO_RdRelease      => '0',
    IP2WFIFO_RdReq          => '0',
    IP2WFIFO_RdRestore      => '0',
    WFIFO2IP_AlmostEmpty    => open,
    WFIFO2IP_Data           => ZERO_WFIFO2IP_Data,
    WFIFO2IP_Empty          => open,
    WFIFO2IP_Occupancy      => open,
    WFIFO2IP_RdAck          => open,
    IP2Bus_IntrEvent        => ZERO_IP2Bus_IntrEvent,
    IP2INTC_Irpt            => open,
    Freeze                  => '0',
    Bus2IP_Freeze           => open,
    OPB_Clk                 => Clk,
    Bus2IP_Clk              => sBus2IP_Clk,
    IP2Bus_Clk              => '0',
    Reset                   => Reset,
    Bus2IP_Reset            => sBus2IP_Reset);

------------------------------------------------------------------------------
-- Forwarded Logic
------------------------------------------------------------------------------
```

```
    oControl_Clk     <= sBus2IP_Clk;
    oControl_Reset   <= sBus2IP_Reset;
    oControl_Addr    <= sBus2IP_Addr(C_OPB_AWIDTH-CONTROL_AWIDTH-2 to C_OPB_AWIDTH-2-1);
    oControl_BE      <= sBus2IP_BE(0 to CONTROL_DWIDTH/8-1);
    oControl_WrBus   <= sBus2IP_Data(0 to CONTROL_DWIDTH-1);
    oControl_Req     <= sBus2IP_CS(USER01_CS_INDEX);
    oControl_RNW     <= sBus2IP_RNW;
    sIP2Bus_Ack      <= iControl_Ack;
    sIP2Bus_Data     <= iControl_RdBus(0 to CONTROL_DWIDTH-1);
    sIP2Bus_toutSup <= iControl_Wait;

end arch;
```

D.2 iBench Assembly

D.2.1 Compile and Execution

It is possible to design *iBench* to use self–modifying code, however it is not possible when securing the application for reasons discussed in this work. Instead, for each iteration in size for a stride, many executables require compilation and encryption. A series of scripts are required, specific to size and stride, to perform these operations. The following are the steps required to build, encrypt, and execute the benchmark suite.

- Compile the script generator: host–gcc [makescripts.c] -o makescripts
- Compile the time converter: target–gcc [time_conv.c] -o time_conv
- Compile the walk–table append utility: host–gcc [append.c] –o append
- Generate the scripts: makescripts [stride] [size]
- Build each benchmark: sh ./build
- Encrypt each benchmark: sh ./enc
- Execute unencrypted benchmarks (on target): sh ./run
- Execute encrypted benchmarks (on target): sh ./run

D.2.2 walk.s

```
.section ".data"
timers:
        .string "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
usagestring:
        .string "Usage: walk <size> <iterations>\n"
        usagestring_size = . - usagestring

.section ".text"
.globl _start

_start:

        ###############################################
```

```
        # Initialize
        ##############################################
        lwz     4, 0(1)                 # r1 contains ptr to argc/argv
        cmpwi   4, 3                    # need, 3 parameters: exec name
        bne     _usage                  #  size(mult of stride), iterations

        lwz     5, 8(1)                 # grab pointer for 2nd param (size)
        bl      _atoi                   # pointer to numeric
        addi    8, 7, 0                 # backup size into r8

        lwz     5, 12(1)                # grab pointer for 3rd param (iter)
        bl      _atoi                   # pointer to numeric
                                        # iterations stored in r7

        li      6, STRIDE               # store strides into test input r6
        divwu   6, 8, 6                 # divide size by stride

                                        # iter test input is already in r7
        b       _walk                   # begin walking


# Input:
#       r5: base string pointer
# Output:
#       r7: resulting atoi conversion
_atoi:
        addi    7, 0, 0                 # zero out r7
        _atoi_start:
                lbzu    6, 0(5)                 # grab next byte from buffer
                addi    5, 5, 1                 # increment pointer
                cmpwi   6, 0                    # end of string?
                beq     _atoi_end
                mulli   7, 7, 10                # new value, multiply current by 10
                subi    6, 6, 0x30              # convert from ascii
                add     7, 7, 6                 # add new value
                b       _atoi_start
        _atoi_end:
        blr

        ##############################################
        # Time Walk
        ##############################################
_walk:
        # Warm up
        addis   3 , 0, _page@ha                 # r3 contains addr of page to walk
        ori     3,  3, _page@l
        addis   31, 0, _f_Walk@ha
        ori     31,31, _f_Walk@l
        mtlr    31                              # load LR with walk function
        blrl                                    # branch to LR, return here

        # Get start time
        li      0, 78                           # call = gettimeofday
        addis   3, 0, (timers+16)@ha            # param1 = ptr struct timeval
        ori     3, 3, (timers+16)@l
        li      4, 0                            # param2 = ptr struct timezone
        sc

        # Walk table
        addis   3,  0, _page@ha                 # r3 contains addr of page to walk
        ori     3,  3, _page@l
        addis   31, 0, _f_Walk@ha
        ori     31,31, _f_Walk@l
        mtlr    31                              # load LR with walk function
```

```
        blrl                                    # branch to LR, return here

        # Get stop time
        li      0, 78                           # call = gettimeofday
        addis   3, 0, (timers+24)@ha            # param1 = ptr struct timeval
        ori     3, 3, (timers+24)@l
        li      4, 0                            # param2 = ptr struct timezone
        sc

        ###############################################
        # Output timings
        ###############################################

        # Write test start time
        li      0, 4                            # call = std_write
        li      3, 1                            # param1 = FileDesc = stdout
        addis   4, 0, (timers+0)@ha             # param2 = ptr buffer
        ori     4, 4, (timers+0)@l
        dcbst   0, 4                            # force cache to write back
        sync
        li      5, 8                            # param3 = buffer len
        sc

        # Write test stop time
        li      0, 4                            # call = std_write
        li      3, 1                            # param1 = FileDesc = stdout
        addis   4, 0, (timers+8)@ha             # param2 = ptr buffer
        ori     4, 4, (timers+8)@l
        dcbst   0, 4                            # force cache to write back
        sync
        li      5, 8                            # param3 = buffer len
        sc

        # Write walk start time
        li      0, 4                            # call = std_write
        li      3, 1                            # param1 = FileDesc = stdout
        addis   4, 0, (timers+16)@ha            # param2 = ptr buffer
        ori     4, 4, (timers+16)@l
        dcbst   0, 4                            # force cache to write back
        sync
        li      5, 8                            # param3 = buffer len
        sc

        # Write walk stop time
        li      0, 4                            # call = std_write
        li      3, 1                            # param1 = FileDesc = stdout
        addis   4, 0, (timers+24)@ha            # param2 = ptr buffer
        ori     4, 4, (timers+24)@l
        dcbst   0, 4                            # force cache to write back
        sync
        li      5, 8                            # param3 = buffer len
        sc

        b       _end
_usage:
        # Write walk stop time
        li      0, 4                            # call = std_write
        li      3, 1                            # param1 = FileDesc = stdout
        addis   4, 0, (usagestring)@ha          # param2 = ptr buffer
        ori     4, 4, (usagestring)@l
        li      5, usagestring_size             # param3 = buffer len
        sc
        b       _end                            # quit out
```

```
_end:
        # Exit
        li      0, 1                            # call = exit
        li      3, 13                           # param1 = return code
        sc



# Function Walk
#       Walks the jump table.
#       Memory pre-initialized to branch in stride-length units. last entry in
#               table branches to LR, which is in local loop
#       Registers modified:
#               CTR, r29, r30, r31
#       Inputs:
#               LR: return address
#               r3: page to walk
#               r6: number of strides to walk
#               r7: number of iterations
#       Internal:
#               r30: iteration down counter
#               r31: caller function LR backup
#               LR: jump back and forth between walk counter and walking
#       Returns:
#               none
_f_Walk:

        # Init
        mflr    31                              # Backup LR into r31

        mullw   30, 6, 7                        # Total branch count=nstrides*iterations

        mtlr    3                               # Load LR with page to walk
        blrl                                    # branch to walk buffer, LR to return here

        mtlr    31                              # restore caller LR
        blr                                     # return to caller
```

D.2.3 append.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
        FILE * fp;
        unsigned int i;
        unsigned int stride;
        unsigned int size;

        if (argc != 4) {
                printf("Usage: %s <Appendable file> <stride> <size>\n", argv[0]);
                exit(1);
        }

        stride=atoi(argv[2]);
        size=atoi(argv[3]);

        fp = fopen(argv[1], "a");
        if (fp == NULL) {
                printf("Unable to open %s\n", argv[1]);
                exit(1);
        }

        if (size % stride != 0) {
                printf("Size is not a multiple of stride!\n");
                exit(1);
        }

        // Print section information
        fprintf(fp, ".section \".pages\", \"awx\"\n");
        fprintf(fp, ".balign 0x1000 \n");
        fprintf(fp, "_page:\n");


        fprintf(fp, "mtctr 30\n");
        fprintf(fp, "bdnz STRIDE-4\n");
        fprintf(fp, "blr\n");
        fprintf(fp, ".rept (STRIDE/4)-3\n");
        fprintf(fp, "nop\n");
        fprintf(fp, ".endr\n");

        // Print repeating strides
        for(i=0; i<(size/stride)-2 ; i++)
        {
                fprintf(fp, "\tbdnz STRIDE\n");
                fprintf(fp, "\tblr\n");
                fprintf(fp, "\t.rept (STRIDE/4)-2\n");
                fprintf(fp, "\t\tnop\n");
                fprintf(fp, "\t.endr\n");
        }

        fprintf(fp, "mfctr 30\n");
        fprintf(fp, "mtctr 3\n");
        fprintf(fp, "bctr\n");
        fprintf(fp, ".rept (STRIDE/4)-3\n");
        fprintf(fp, "nop\n");
        fprintf(fp, ".endr\n");

        fclose(fp);
        return 0;
}
```

D.2.4 time_conv.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define DBG 0

int main(int argc, char * argv[])
{
        unsigned long size;
        unsigned long stride;
        unsigned long iter;
        double itime;
        double walktime;
        double latency;

        struct timeval test_start, test_stop, walk_start, walk_stop;

        if (argc != 4) {
                printf("Incorrect usage: %s <stride> <size> <iterations>\n", argv[0]);
                exit(1);
        }

        stride = atoi(argv[1]);
        size = atoi(argv[2]);
        iter = atoi(argv[3]);
        fread(&test_start, sizeof(struct timeval), 1, stdin);
        fread(&test_stop, sizeof(struct timeval), 1, stdin);
        fread(&walk_start, sizeof(struct timeval), 1, stdin);
        fread(&walk_stop, sizeof(struct timeval), 1, stdin);

        // Get stop time in usecs
        walktime  = (double)( (double)walk_stop.tv_sec * 1000000.0) +  (double)walk_stop.
        tv_usec;
        // Subtract start time in usecs
        walktime -= ((double)((double)walk_start.tv_sec * 1000000.0) + (double)walk_start.
        tv_usec);
        // Convert from us to ns
        walktime *= (double)1000.0;

        // Get number of instruction executions (1 cycle/instruction)
        itime  =  (double)((size/stride - 2 + 2*2 + 3) * iter) * (double)iter;
        // Multiply # instructions by period for each instruction
        itime /= (double)300000000.0;

        latency = (walktime - itime)/((size/stride + 1)*iter);

        printf("%lu, %.5f, %lu, ", stride,  size / (1024. * 1024.), iter);
        printf("%.3f, ", walktime);
        printf("%.3f, ", itime);
        printf("%.3f", latency);

        printf("\n");
        return 0;
}
```

D.2.5 makescripts.c

```
// ./makescripts 256 8388608
#include <stdio.h>
#include <stdlib.h>

#define LOWER 4096

unsigned int
step(unsigned int k)
{
        if (k < 1024) {
                k = k * 2;
        } else if (k < 4*1024) {
                k += 1024;
        } else {
                unsigned int s;

                for (s = 32 * 1024; s <= k; s *= 2)
                        ;
                k += s / 16;
        }
        return (k);
}


int main(int argc, char * argv[])
{
        FILE * fpmake, *fprun, *fprunenc, *fpenc;
        unsigned long range;
        unsigned long stride;
        unsigned long maxsize;
        unsigned long iter;

        if (argc != 3) {
                printf("Usage: %s, <stride> <size>\n", argv[0]);
                exit(1);
        }

        stride=atoi(argv[1]);
        maxsize=atoi(argv[2]);
//      iter=atoi(argv[3]);

        fpmake = fopen("make", "w");
        if (fpmake == NULL) {
                printf("Unable to open 'make'\n");
                exit(1);
        }
        fprun = fopen("run", "w");
        if (fprun == NULL) {
                printf("Unable to open 'run'\n");
                exit(1);
        }
        fprunenc = fopen("runenc", "w");
        if (fprunenc == NULL) {
                printf("Unable to open 'runenc'\n");
                exit(1);
        }
        fpenc = fopen("enc", "w");
        if (fpenc == NULL) {
                printf("Unable to open 'enc'\n");
                exit(1);
        }
```

```
        fprintf(fprun, "echo \"Stride, Size, Iterations, Walk Time, Instruction Time, Walk
        Latency\" > log_%lu.csv\n", stride);
        fprintf(fprun, "chmod 777 ./log_%lu.csv\n", stride);
        fprintf(fprunenc, "echo \"Stride, Size, Iterations, Walk Time, Instruction Time,
        Walk Latency\" > log_%lu_enc.csv\n", stride);
        fprintf(fprunenc, "chmod 777 ./log_%lu_enc.csv\n", stride);
        for (range = LOWER; range <= maxsize; range = step(range)) {
                if (range < stride) continue;

                iter = (6553600*stride)/range;
                fprintf(fpmake, "cp src/walk.s newwalk_%lu.s\n", range);
                fprintf(fpmake, "./append newwalk_%lu.s %lu %lu\n", range, stride, range);
                fprintf(fpmake, "ppc-405-as newwalk_%lu.s -o walk_%lu.o -v --defsym STRIDE=%
                lu\n", range, range, stride);
                fprintf(fpmake, "ppc-405-ld walk_%lu.o -o walk_%lu -v\n", range, range);

                fprintf(fprun, "./walk_%lu %lu %lu >& /dev/null\n", range, range, iter);
                fprintf(fprun, "./walk_%lu %lu %lu | ./time_conv %lu %lu %lu >> log_%lu.csv
                \n", range, range, iter, stride, range, iter, stride);

                fprintf(fprunenc, "./walk_%lu.self %lu %lu >& /dev/null\n", range, range,
                iter);
                fprintf(fprunenc, "./walk_%lu.self %lu %lu | ./time_conv %lu %lu %lu >> log_
                %lu_enc.csv \n", range, range, iter, stride, range, iter, stride);

                fprintf(fpenc, "../selfcreatetony_new walk_%lu header 1 2 3 4 5 >& /dev/null
                \n", range);
                fprintf(fpenc, "mv josh walk_%lu.self\n", range);
                fprintf(fpenc, "chmod 755 walk_%lu.self\n", range);
        }

        fclose(fpmake);
        fclose(fprun);
        fclose(fpenc);

        return 0;
}
```