Secure Processor Design

by

Nicholas Butta

Spring 2019

EN.525.612.81.SP19 Computer Architecture

Project Report

Table of Contents

Introduction

From consumer electronics and transportation, to medicine and military applications, embedded computer processors exist in almost every industry of modern society. As the usage and accessibility of embedded processors has grown, so too has the importance of protecting intellectual property of software developers and maintaining the security of data being processed by these embedded systems. Breaches in hardware security can result in wide range of consequences including the loss of revenue and even human life. For this reason, many large CPU manufacturers like ARM are investing heavily in designing cost-effective, resource-efficient and high-performance secure processors from the physical hardware level for usage in embedded applications. Today, secure processors from ARM are being licensed by the industry's biggest microcontroller companies and have even become the de facto processors of choice for Internet of Things (IoT) devices. These processors are designed to fend off attacks designed to subvert the original, intended operation of a running program by modifying stored instructions/data or stealing sensitive data from the CPU hardware, whether it be stored passwords, banking information or health records. Said attacks can be launched remotely or with direct physical access to a CPU die and can come in the form of malware, invasive physical attacks or non-invasive physical attacks. Luckily, however, through the usage of anti-tamper integrated circuit technology and some basic cryptographic theory, many basic malicious attacks can be prevented from succeeding. Thus, even with significant investment and highly-qualified specialists, attackers are left helpless in circumventing many secure processing solutions.

Project Scope

For this project, the idea was to study common design practices and challenges/trade-offs faced in the computer processor industry to design secure processors. I then wanted to investigate the feasibility of adding security extensions to one of the MIPS CPU designs from the Li textbook in Verilog and possibly synthesize the design on the Nexys4 DDR Artix-7 FPGA. Ultimately, I was very successful in my achieving the goals I set out for myself at the start of the project. I was able to identify one particularly useful dissertation on designing secure processors for embedded systems, which included an explanation of general principles, challenges and common architectural solutions. I studied the dissertation thoroughly in order to grasp a full understanding of the theory, while keeping in mind how the concepts might be applied to one of the processors studied up through that point of the course. At the conclusion of my research, I decided that the Multi-Cycle processor was the best processor to implement security extensions into due to it's fairly straightforward control logic. It should be noted, however, that the secure processor design I came up with should be easily translatable to practically any of the processors distributed throughout the course. I would love to see security extensions incorporated into the Pipelined CPU with Caches and TLBs (Li Chapter 12) of Module 10 and envision this as a potential project for students in future semesters. My final project deliverables include Verilog source code implementing the Li/Butta MIPS Multi-Cycle Secure Processor (MCSP) hardware, along with C code for producing a secure binary executable.

Definitions

A few concepts should be defined before going into the architectural details of the Li/Butta MIPS Multi-Cycle Secure Processor (MCSP). Within the context of this project, a *secure processor* is defined as a CPU that provides confidentiality and integrity of programs and data being executed using proven cryptographic functions. Confidentiality is provided by using a strong encryption algorithm. Integrity is provided through the usage of message authentication codes, or signatures. This means that a secure processor can execute any encrypted and signed program, ensuring that the instructions and data are illegible to any untrusted parties and that the CPU only ever executes instructions that have been successfully decrypted and verified by the processor itself. By doing this, various attacks on a processor designed to extract sensitive data or force the processor into a state where it is susceptible to execute malicious code are avoided. The MCSP design provides both confidentiality and integrity using a sign-and-verify architecture through the usage of a Galois/Counter Mode block with AES as the underlying cryptographic cipher. This is explained in greater detail further in the project report.

When designing a secure processor, one must define the processor's *security boundary*. The security boundary is simply the boundary beyond which data are potentially vulnerable to unauthorized viewing or tampering by attackers. Instructions and data within the security boundary are safe from attacks because it is assumed sufficient anti-tamper measures are in place, which would prevent an attacker from modifying or viewing the program within the specified boundary. Data outside the security boundary must never exist unencrypted, else we risk an attacker being able to view and potentially reverse-engineer our system. In most cases, the physically boundary of the CPU die is chosen as the security boundary; when a cache miss occurs, and the processor must go off-chip to external main memory, the security boundary is crossed. The security boundary of the MCSP, however, is the CPU control logic and register file, meaning that the instruction and data cache are considered outside the security boundary. Therefore, all programs and dynamic data are encrypted and signed before storage in

the cache and decrypted and verified before being fetched/loaded from the cache. This is the basic premise of our sign-and-verify architecture.

In addition to security boundary, the concept of *security mode* should also be defined. The security mode specifies the protection level for our programs (instructions and static data) and dynamic data. This is normally a parameter that is selected by the programmer. In the case of the MCSP, the security mode is fixed. Maximum security (SCIM/DCIM) is always maintained. This means both static and dynamic data are always encrypted and signed when stored beyond the security boundary, and decrypted and verified when entering the security boundary (fetched or loaded from cache).

Finally, the *protected block size* must be defined before implementing a secure processor in hardware. A protected block is the smallest, discrete unit of a security within the context of a secure processor. Each protected block is encrypted independently and has its own signature associated with it attesting to it's integrity. The cache line size, cipher width and memory constraints must all be considered when determining the protected block size. The smaller the protected block size, the more signatures required and vise versa. The MCSP's protected block size is four bytes or one word. Because MIPS is a 32-bit architecture, this means that each instruction and data in the cache has its own signature.

Li/Butta Multi-Cycle Secure Processor Detailed Design

Top-level Design

The Li/Butta MIPS Multi-Cycle Secure Processor provides confidentiality and integrity of a running program's static and dynamic data. Static data is data that does not change and in most programs, consists solely of program instructions. This type of data is vulnerable to spoofing and splicing attacks. Dynamic data, on the other hand, can change during the course of program's execution and is written and read during program execution using the Store Word (SW) and Load Word (LW) instructions of the MIPS architecture. In addition to spoofing and splicing, dynamic data is vulnerable to replay attacks. The protected block size of the MCSP is four bytes, or one word, so every instruction and data word has its own signature associated with it. As mentioned previously, the MCSP uses a sign-and-verify architecture for maintaining maximum security. The security boundary is placed around the mccpu block, which contains the CPU register file and all of the CPU control logic for executing instructions. The security boundary is crossed in three cases: Instruction Fetch (IF), Load Word and Store Word. When data leaves the security boundary (SW), the data is encrypted, signed and stored in the mcmem and sigram caches separately. When data enters the security boundary (IF and LW), the data is decrypted and verified. Verification is performed by computing a signature based on the retrieved ciphertext and comparing the fetched signature with the computed signature. If the signatures match, the data can be trusted and the instruction can continue execution. Otherwise, tampering has been detected and the instruction is discarded.

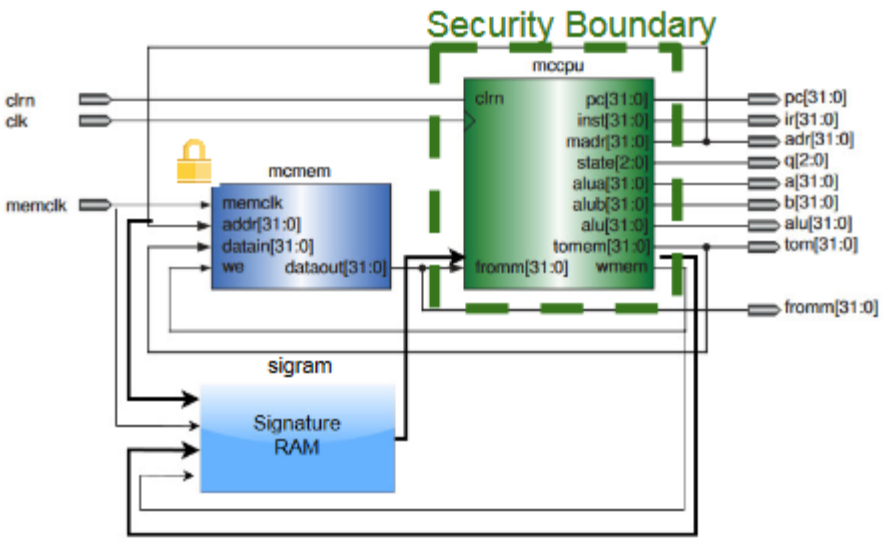


Figure 1. Top-level Detailed Design Diagram

A top-level block diagram of the MCSP can be seen above in Figure 1. The Multi-Cycle CPU distributed by Li contains a combined instruction and data cache, mcmem, with an 8-bit address space. In addition to the instruction/data cache, a separate memory was added to the MCSP design that contains the signatures of all cache entries (sigram). By indexing into the two memories using the same address output from the mccpu block, requested instruction/data and its associated signature are retrieved together in one clock cycle. This operation occurs on Instruction Fetch and Load Word instructions. Separate data buses exist to write an encrypted word and its signature to the memories, but the same write enable signal is fed to both blocks in order to write data and its signature simultaneously in one clock cycle. This operation only occurs on Store Word instructions.

Instruction Fetch

The detailed circuit diagram for performing Instruction Fetch can be found below in Figure 2. This diagram contains the same encrypted instruction/data memory from the top-level diagram in blue. These memories contain our program instructions/data and their associated signatures. Keep in mind that these memories exist outside of the security boundary. The red blocks represent registers. The Program Counter register is contained in the CPU's PC Register. During Instruction Fetch, this value is used to index into both the encrypted instruction memory and signature memory. The output is stored in the Encrypted Instruction Register (EIR) and Instruction Signature Register (ISR) respectively. At this point, the instruction must be decrypted and the signature verified in order to continue program execution. The state machine logic within the purple Control Unit enables the GCM-AES Controller block, which feeds EIR as the ciphertext input and PC as the Initial Vector (IV) to the GCM-AES Engine. Once the GCM-AES Engine has produced the decrypted plaintext instruction and it's tag (signature), the Control Unit FSM logic compares the signatures. If the signatures match, the Decrypted Instruction Register (DIR) is written with the plaintext instruction output of the GCM-AES Engine. The instruction can then continue execution until completion as normal.

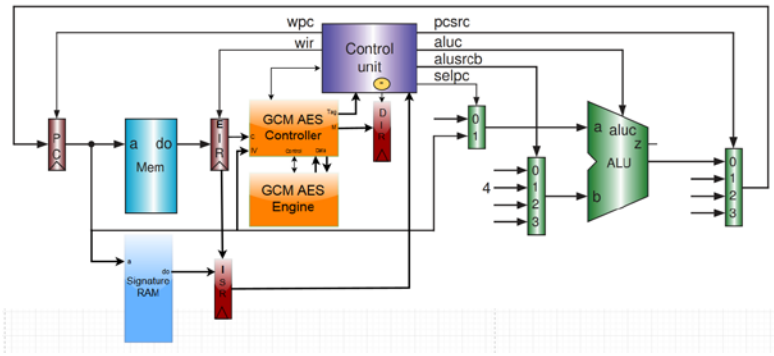


Figure 2. Instruction Fetch Detailed Design

Control Unit Design

The state machine for the MCSP, shown in Figure 3, is a modified version of the Li Multi-Cycle CPU design. Refer to the Li Textbook Chapter 7 on the Multi-Cycle Processor for more details. Extra authentication and verification states were added to the MCSP Control FSM to account for instructions and data crossing the processor's security boundary. An encrypted instruction and its signature are first fetched from the caches in the **sif** state. Then, in the **sverify** state, the GCM-AES Engine in enabled and the FSM waits for the *gcm_aes_done* signal to go high. Once this happens, the computed and fetched signatures are compared. If the signatures do not match, the state machine goes back to the **sif** state and will fetch the next instruction. If the signatures match, the state machine progresses to the **sid** state. Here, the instruction is decoded and combinational signals are set appropriately. Jump instructions are completed at this point and return to **sif.** The FSM then proceeds to the **sexe** state where the instruction is executed according to its opcode. Branch instructions are completed at this point and return to **sif**. All instructions except for LW and SW instructions move to **swb** after **sexe**. In **swb**, the specified register is written with an appropriate value based on the instruction. For SW instructions, we must encrypt the data from the register file and calculate its signature in the **sauth1** and **sauth2** states. We do this by enabling the GCM-AES block two times. This results in the encrypted data and a signature on the plaintext, as opposed to the ciphertext signature. Normally, the signature is calculated on the ciphertext, using an Encrypt-then-Sign philosophy, but due to limitations of the GCM-AES implementation used, signatures are always produced on the plaintext and stored into the signature cache. For LW instructions, we must verify and decrypt the data from the data cache. We enable the GCM-AES block in the **sdataverify** state and compare computed and fetched signatures as in the case of Instruction Fetch. Both SW and LW instructions then progress to the **swb** state to update the Register File (LW instructions only) before returning to **sif** to execute the next instruction.

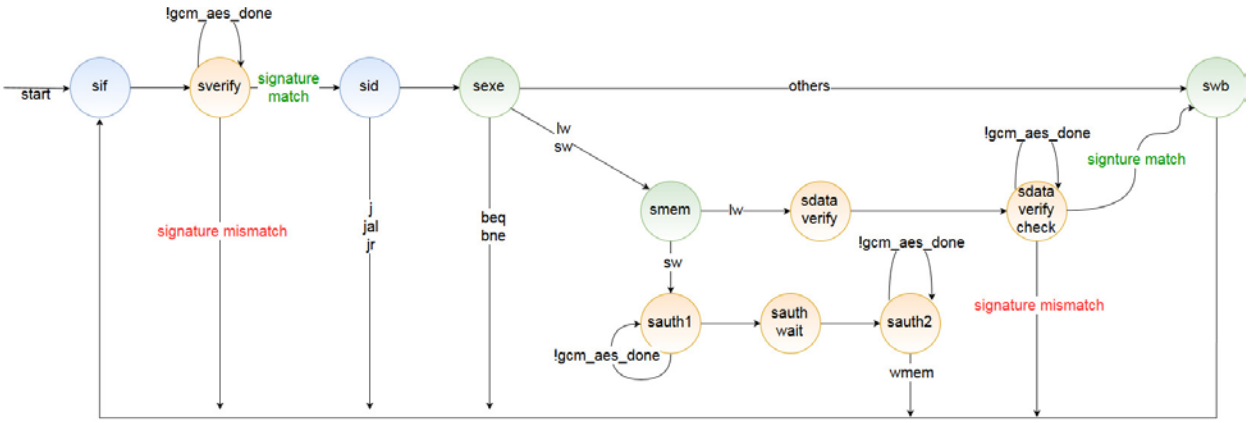


Figure 3. Control Unit State Transition Diagram

Load Word/Store Word Design

The detailed design for performing Load Word (LW) and Store Word (SW) instructions is shown in Figure 4. The circuits depicted in this diagram describe the functionality of the sign-and-verify architecture as it pertains to protecting dynamic data. Because the GCM-AES Engine uses a One-Time-Pad AES encryption and authentication scheme as the underlying cipher, a Sequence Number memory exists to make sure we use unique Initial Vectors every time a word is stored in the cache. This is one way of implementing a versioning scheme for a secure processor. The Sequence Number memory contains one independent integer value for each of the 255 entries in the CPU's data and signature caches. This value is combined with the data's address (held in Register C) and used as the IV input to the GCM-AES Controller on load and store word operations. Before each store instruction is executed, this value is incremented. The same value is used when the same data word is loaded. For SW instructions, we are exiting the secure boundary. The value to be stored from the Register File is fed as input to the GCM-AES Engine and the ciphertext and signature are produced. The encrypted output is then fed back into the GCM-AES Engine to produce a signature on the original plaintext. This is a workaround due to restrictions of the GCM-AES block. Ultimately, the encrypted data word and its signature (on the plaintext) are written to the cache and the store is complete.

Similar to Instruction Fetch, LW instructions require a decryption and verification of data that has just entered the security boundary. Data and its signature are retrieved from the data cache and signature cache using the Register C, which contains the data's effective address during and Load/Store instruction. The encrypted data is written into the Encrypted Data Register (EDR) and the data's signature is written into the Data Signature Register (DSR). The Control Unit selects the EDR as input to the GCM-AES Engine and the value is decrypted and the signature calculated. If the signatures match, the plaintext data output of the GCM-AES block is written into the Register File. Otherwise, the instruction is abandoned.

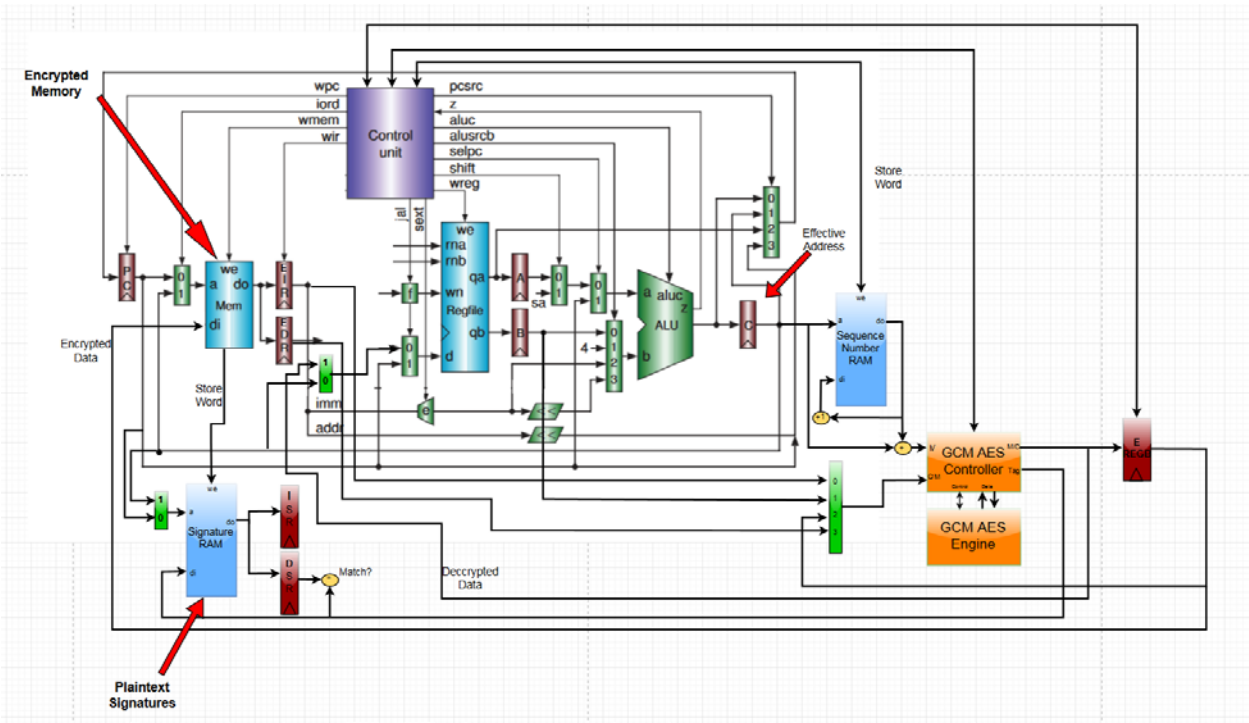


Figure 5. Load Word/Store Word Detailed Design

GCM-AES Controller

In order to stimulate the GCM-AES Engine, the GCM-AES FSM drives the various input signals and monitors output signals from the GCM-AES Engine to produce ciphertext/plaintext and signatures. Refer to the design spec from opencores.org for more details. Currently, the value for the secret key is a hard-coded input to the GCM-AES Engine and the additional authentication data (AAD) is hard-coded into the FSM logic. These parameters could be extracted from random LFSR outputs in future iterations of the processor.

1. The value of the secret key is fixed to *0xfeffe9928665731c6d6a8f9467308308*
2. The value of the AAD is fixed to *0xfeedfacedeadbeeffeedfacedeadbeefabaddad2*

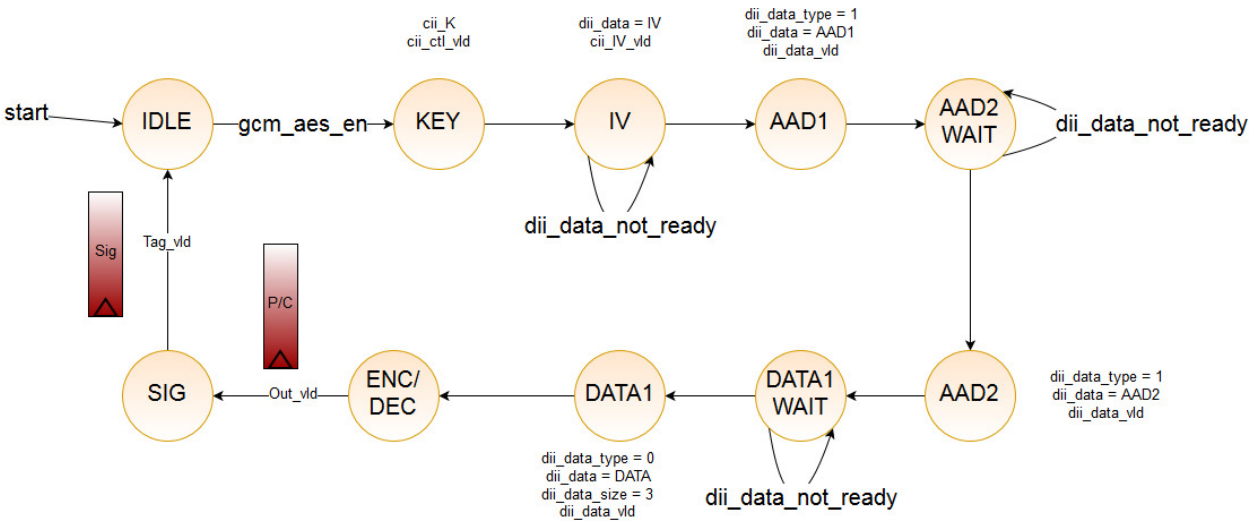
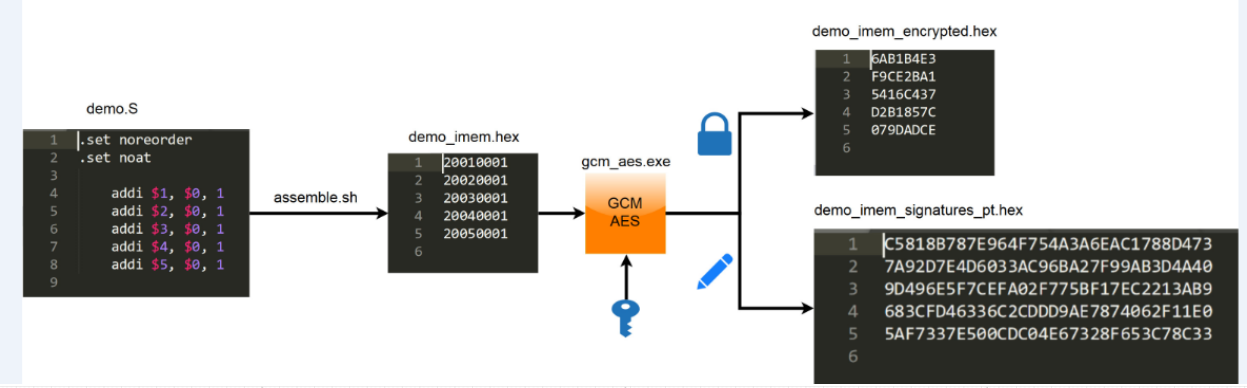
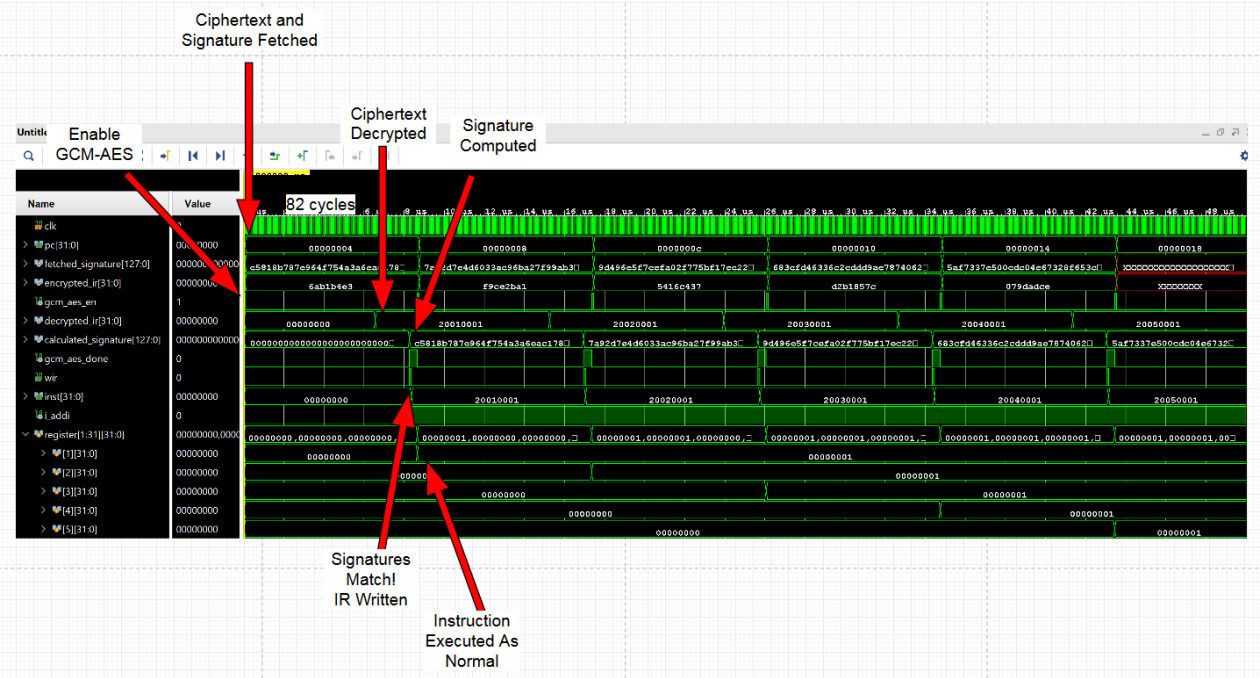


Figure 6. GCM-AES Controller State Transition Diagram

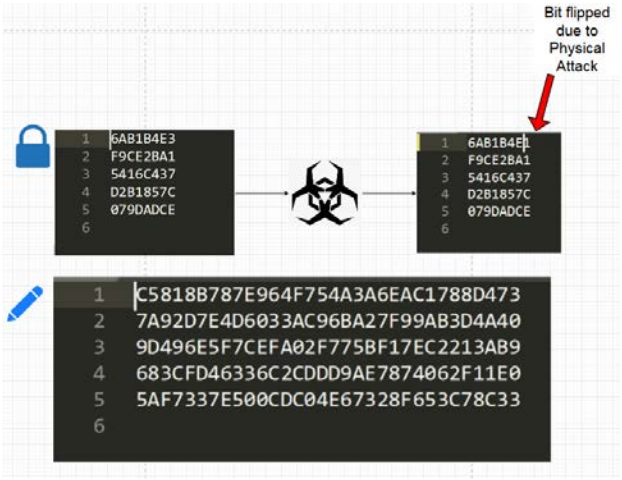
Simulation Results

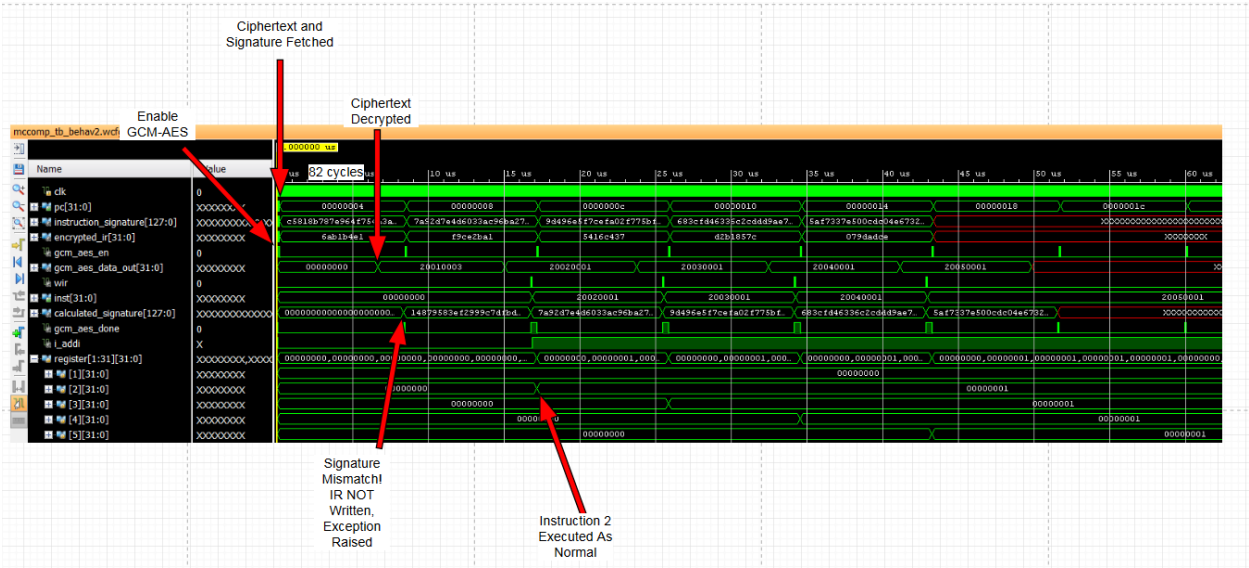
Protecting Static Data





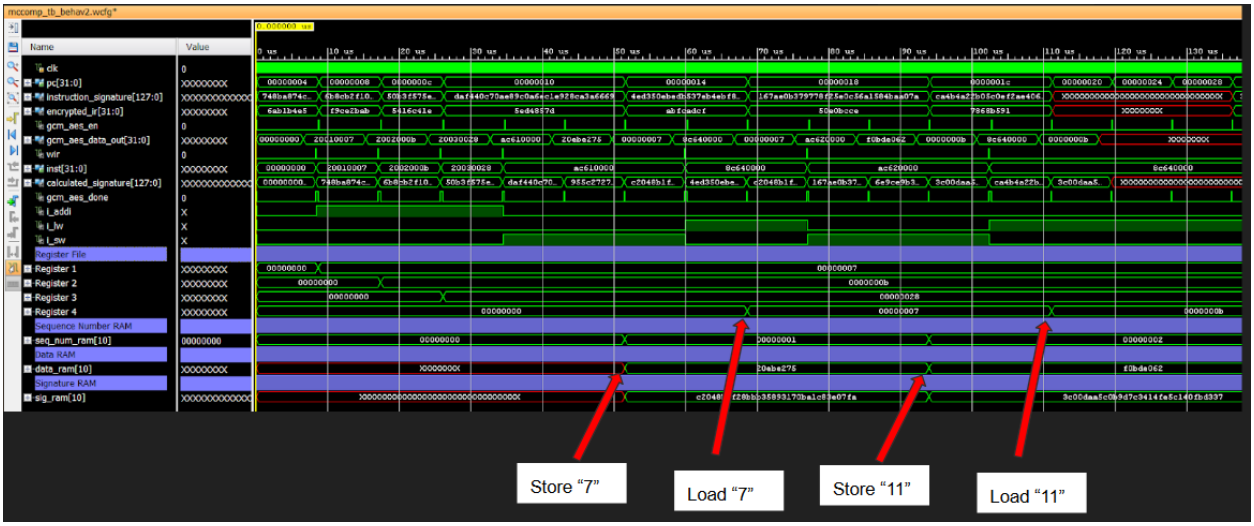
Detecting an Attack





Protecting Dynamic Data

```
1       addi $1, $0, 7
2       addi $2, $0, 11
3       addi $3, $0, 40
4       sw   $1, 0($3)
5       lw   $4, 0($3)
6       sw   $2, 0($3)
7       lw   $4, 0($3)
8
```



Conclusions

This project was ultimately very successful and a very interesting experiment. I was able to achieve the goals set out from the beginning and implement a working secure processor in Verilog. I learned a great deal about secure processing, cryptography and computer architecture. The trade-offs between speedm resources and security remain the biggest challenge for secure processing solutions. In my design, for example, Each GCM AES operation requires 82 cycles to produce a signature and ciphertext/plaintext. Every instruction must suffer an 82 cycle penalty on Instruction Fetch (verification). Every load instruction suffers an additional 82 cycle penalty (decrypt/verification). Every store instruction suffers a 164 cycle penalty (encrypt/double authentication). Using the CINT92-SPARC Benchmark (27.3% Memory, 23.3% Branch, 49.4% Other), the Average CPI for the Li Multi-Cycle CPU is 3.68. The Average CPI for the Li/Butta Secure MC CPU is 119.43. This is a speedup = 3.68/119.43 = .031 (slowdown).

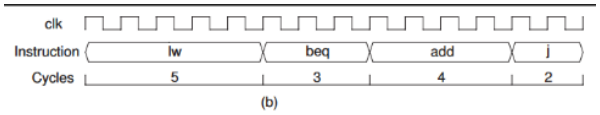


Figure 7.1 Timing comparison of (a) single-cycle and (b) multiple-cycle CPUs

Table 7.1 Clock cycles required by instructions

Instruction	Meaning	MC Cycles	MCSP
`add rd, rs, rt`	Register add	4	86
`sub rd, rs, rt`	Register subtract	4	86
`and rd, rs, rt`	Register AND	4	86
`or rd, rs, rt`	Register OR	4	86
`xor rd, rs, rt`	Register XOR	4	86
`sll rd, rt, sa`	Shift left	4	86
`srl rd, rt, sa`	Logical shift right	4	86
`sra rd, rt, sa`	Arithmetic shift right	4	86
`jr rs`	Register jump	2	84
`addi rt, rs, immediate`	Immediate add	4	86
`andi rt, rs, immediate`	Immediate AND	4	86
`ori rt, rs, immediate`	Immediate OR	4	86
`xori rt, rs, immediate`	Immediate XOR	4	86
`lw rt, offset(rs)`	Load word	5	169
`sw rt, offset(rs)`	Store word	4	251
`beq rs, rt, offset`	Branch on equal	3	85
`bne rs, rt, offset`	Branch on not equal	3	85
`lui rt, immediate`	Load upper immediate	4	86
`j address`	Jump	2	84
`jal address`	Subroutine call	2	84

References

1. Secure Processing Dissertation
    a. http://www.ece.uah.edu/~milenka/docs/AustinRogers.dissertation.pdf
2. GCM AES Specification with NIST test vectors
    a. http://luca-giuzzi.unibs.it/corsi/Support/papers-cryptography/gcm-spec.pdf
3. Open source C implementation of GCM AES
    a. https://github.com/michaeljclark/aes-gcm
4. Open source Verilog implementation of the GCM AES
    a. https://opencores.org/projects/gcm-aes
5. Li, Yamin; Tsinghua University Press, Jun 30, 2015, Computer Principles and Design in Verilog HDL
    a. Distributed with course material