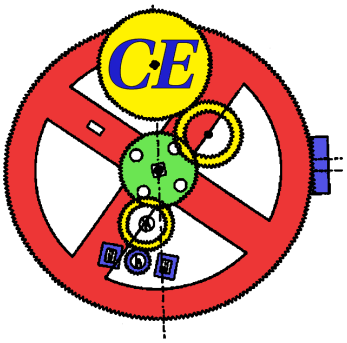

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Code Integrity Check targetting RISC Processors

Andre Michel Abikhaled

CE-MS-2009-13

Abstract

Security is emerging as an important concern in embedded system design. The security of embedded systems is often compromised due to vulnerabilities in the software that they execute. Security attacks exploit such vulnerabilities to trigger unintended program behavior, e.g. the leakage of sensitive data or the execution of malicious code. Many computer security threats involve execution of unauthorized foreign code on the victim system. Viruses, network and email worms, Trojan horses, backdoor programs used in Denial of Service attacks are just a few examples. Program code in a computer system can be changed either by malicious security attacks or by various failures in microprocessors. In this work, we present an enhanced methodology for code integrity at run-time in Reduced Instruction Set Computer (RISC) processors. This is achieved by pre-computing checksums over parts of the binary code before program execution. These pre-computed checksums are usually generated by a Certified Trusted Authority. These values embedded on the binary code are verified at run-time during program execution. For this purpose the targeted processor is augmented with a cryptographic unit and dedicated control unit. The added cryptography unit increases the overhead delay about 30 percent and the area doubles in size.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Code Integrity Check targetting RISC Processors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Andre Michel Abikhaled
born in Beirut, Lebanon

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Code Integrity Check targetting RISC Processors

by Andre Michel Abikhaled

Abstract

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-13

Committee Members :

Advisor:	dr.ir. Georgi Gaydadjiev, CE, TU Delft
Chairperson:	dr. Koen Bertels, CE, TU Delft
Member:	dr.ir. Stephan Wong, CE, TU Delft
Member:	dr. Marjan Popov, EE, TU Delft

Special Thanks to my advisor for his support in this Thesis work. I would like also to thank my family and all the friends for their support.

Contents

List of Figures

List of Tables

Acknowledgements

Andre Michel Abikhaled
Delft, The Netherlands
September 11, 2009

1 Overview

1.1 Problem statement

Let us assume a system of central authority and users. The authority wants to share secrets and data with users having portable computing devices. Our target is protecting the executed code from malicious modifications. The possible attacks can be classified as software and physical attacks on code and data. The authority can be a military organization with remote devices used by soldiers in the field. It could be a bank with devices used by customers as personal ATMs for dispensing electronic cash and accepting deposits into bank. It could even be a cell phone network provider where users are allowed to download trusted software[1]. The main problem of all the above systems is that most of the new security attacks result in violating the integrity of the software code of an application program. Such security attacks or threats try to change the instructions so that adversary try to gain control over the program execution flow[26]. Figure 1.1 illustrates an example of an attack using malicious code.

In Figure 1.1(a), function g() is vulnerable because it consists of an operation - strcpy which copies the string str1 into buffer without considering the size of the string, str1. Since buffer is a local array, it will usually be stored in part of a stack frame belonging to local variables. A program that copies passing the end of this array, will overwrite anything stored after the array. Figure 1.1(b) depicts the program stack when function g() is executed. Function g() is called by function f() which has placed the arguments (arg0..argN) of function g() in the stack just after function f()s local variables before running a call instruction. Figure 1.1(c) illustrates how the return address is overwritten using buffer overflow, by making use of a vulnerable function (strcpy in this case). Since strcpy does not check for the length of the string copied across, the hacker is able to make the program copy data beyond the end of an array (buffer in this case). Apart from the content of the buffer, the attacker has also overwritten the return address of function g(). In most of the attacks of this type, the main concern of the attacker is to overwrite the return address and this is the easiest way to gain control of an application programs execution flow. Usually, when a function returns, it will return to the address pointed by the return address in the stack. As described here, if the adversary has changed the return address to point to the code segment injected probably by the same copy operation (as in Figure 1.1(c)), when function g() returns, it will execute the injected malicious code and therefore giving the control to the attackers code[26].

1.2 Thesis Motivation

In this master thesis we are aiming to improve the security of an existing soft- or hard-core processor by executing it with hardware cryptographic unit performing the Secure Hash

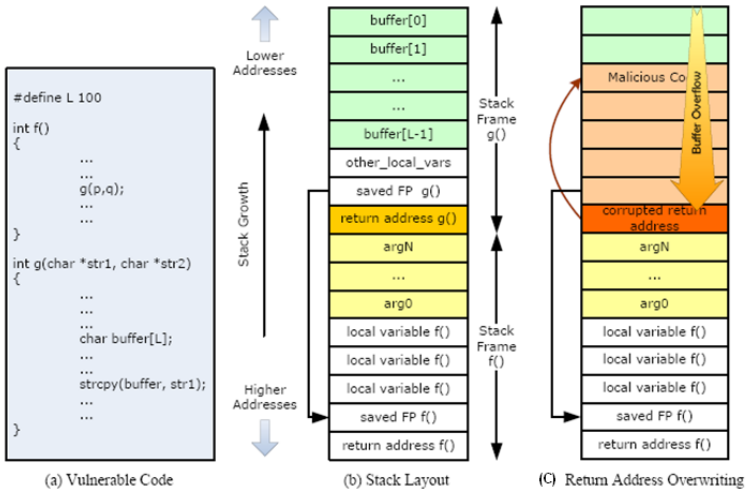

Figure 1.1: Stack Based Buffer Overflow attack overwriting return address[26]

Algorithm (SHA-1). Whenever the instructions are loaded into the on-chip instruction cache from external off-chip caches or main memory, the secure processor performs the code integrity check for the loaded instructions[3]. The secure processor verifies the integrity of the instructions by comparing the hash embedded in the cache line which is calculated statically before starting the execution of the code with the one that is computed at run-time over the instructions in the cache line[2]. The code integrity check guarantees that the binary code loaded in the memory was not altered before execution. For this reason we compute for every cache line the checksum signatures before the program starts. A certified authority generates the checksums loaded into the binary file. At run-time when the program is executed, the real hash is calculated by the cryptographic hash unit in hardware. Different hashes imply that the binary code has been changed and an interrupt is raised[21]. If the hash comparison indicates that both checksums are the same, then the embedded hash in the cache line will be replaced by no-op instruction so as not to affect the execution of the instructions in the pipeline [1].

1.3 Outline of the thesis

The remainder of this thesis is organized as follows. Chapter 2 discusses the different classes of attacks into computer systems like the Hardware-Based Attacks , the architecture for a secure execution of programs on embedded processors is discussed. Moreover, the hash function which is a computationally efficient function converting binary strings of arbitrary length to binary strings of some fixed length called the checksums is going to be discussed. Chapter 3 discusses the 32-bit LEON processor that complies to the SPARC V8 architecture. Chapter 4 discusses the implementation of the SHA-1 unit, the code integrity unit. Chapter 5 describes the comparison of this design with other two designs e.g.The Trusted platform and IBM Secure Coprocessor. Some measurements of

the delay are included in this chapter. Finally this thesis is concluded in chapter 6, where recommendations for future work can also be found.

2 Background

This chapter discusses the background information. Section 2.1 discusses the security issues and the different classes of attacks into computer systems like the Hardware-Based and the Software-based Attacks. The architecture of the design of the Secure Processor is discussed in section 2.2. Section 2.3 illustrates the related work. A hash function which is a function converting binary strings of arbitrary length to binary strings of some fixed length called the checksums is illustrated in section 2.4. Section 2.5 discusses the proposed SHA-1 architecture. Section 2.6 illustrates the implementation of the SHA-1 core in hardware.

2.1 Security Issues

The definition of dependability in a computing system is the trustworthiness of a system which delivers a reliable and trusted system[26]. Dependability includes the following attributes of a computing system[26]:

1. Availability: The correct service of the system is ready or available.
2. Reliability: The correct service is always or continuously available.
3. Safety: The users and environment must have no damage.
4. Security: The security services is defined as the availability for authorized users only, the confidentiality protection of messages from unintended eavesdroppers, the authentication of data, and the integrity protection that shows that the message was not modified in transit.

In this section, the different classes of attacks into computer systems are presented .

1. Hardware-Based Attacks: direct access of the adversary on the hard disk by using direct hardware block access. Special Cryptography is implemented in hardware that will protect the hardware from such attacks. Cryptography like encrypting all sensitive data is used, where the encrypted data can only be accessed after decrypting the data with the corresponding key where the key can be private or public. Another kind of attacks in hardware is the attacks over busses in hardware[27].
2. Software-based Attacks: Special attacks where the hacker tries to execute his code to access protected data or to insert errors in system software to get administrator access. The adversary tries to insert software implementation errors[27]. Some of the common software based attacks were explained in details in [26] [27], these attacks will be again briefly discussed.

- Buffer Overflow: For instance, if the programmer forgets to check if the length of the input data is not more than the size of the buffer located on the stack where data has to be hold, then the adversary inserts errors. The attacker replaces the return address which is stored on the stack with an address pointing to the overflowed buffer containing its own malicious code, where this last can be used to copy passwords or other valuable data to an attacker.
- Heap Overflow: The heap does not contain the return address, but it contains the actual buffer, the size of the buffer and pointers to next and previous buffers are stored in the same memory chunk. The adversary overwrites one of the pointers to point to the return address on the stack such that the return address is overwritten and the adversary inserts malicious software.
- Double free vulnerability: The adversary overwrites the administrative data structure.
- Format string attack: The attacker modifies the format string, to overwrite the contents of the stack to execute a malicious code.
- Temporary file vulnerability: The attacker tries places a link of symbols to force the program to write to a file where he has access to.

2.2 Related Work

Extensive previous work has been done in the area of code integrity check like the Runtime Execution Monitoring (REM) [34]. The main concern in REM is to verify the code integrity at the basic block(a sequence of instructions with a single entry point, and no internal flow control instructions, such as branch, call, return instructions [34]) level. This is done by pre-calculating the hashes for every basic bock before execution of program. During the execution of the program, these precomputed values are verified again by calculating the hash values with the aid of a hash unit implemented in hardware. The datapath of a pipeline processor has been modified to implement the code integrity check. The Secure Hash Unit (implemented in hardware to compute the hashes of the instructions in the basic block) reads instructions in 128-bit blocks and computes the corresponding hash values to the current basic block. This unit is connected to the pipeline control and processes a basic block only after all the instructions in the basic block are executed. At the same time with the hash computation, the stored hash of the current basic block is read from the memory and stored in the first-in-first-out (FIFO) hash read buffer. This buffer is trivial because the hash computation delay in hardware is longer than the latency of looking up the hash value in the buffer. Whenever the hash value computation is finished, it is compared to the stored value, and an interrupt is raised if the hash values mismatch [34]. The block diagram in Figure 2.1 depicts the REM datapath.

Similar work to the REM is done in [22][24]. The embedded processor is modified by adding extra hardware unit that observes the dynamic execution of the processor, checks whether the execution of the allowed program behavior is not tampered [22]. The embedded processor is a five stage pipeline RISC processor. The inputs to the new hardware unit or monitor contains the program counter (PC) and instruction register

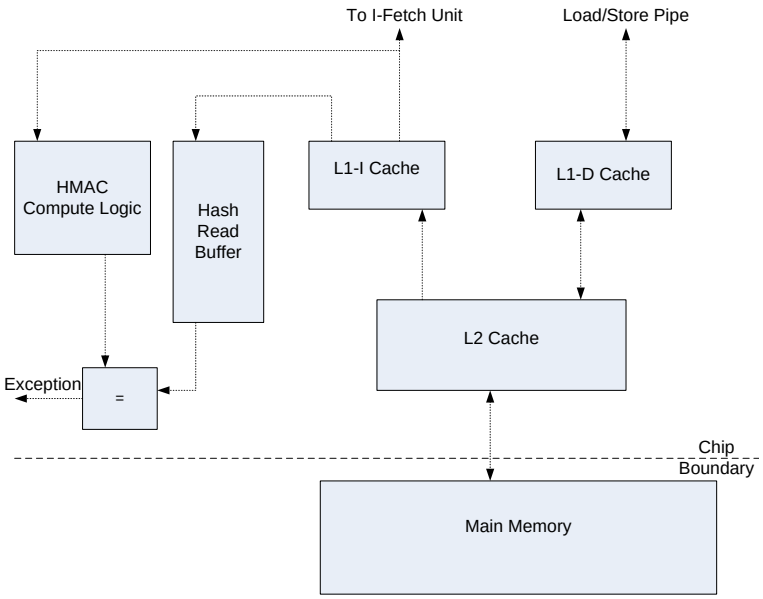

Figure 2.1: REM datapath[34]

(IR) of the completing instruction, and the pipeline control signal from the pipeline control unit. The output of the monitor includes a stall signal and an invalid signal. When the monitor detects a deviation in the program behavior, it raises the invalid signal, which results in an interrupt to the processor [22].

The monitor is splitted into three sub blocks which check program properties at different level of abstractions. The top-most level is the application level, the second level describes the intra-procedural control flow by validating each branch/jump instruction within a function and finally, the lowermost level verifies the integrity of the instruction stream [24]. Some security attacks may not result in a control flow violation, for instance the alteration of a basic block in the program code segment during execution. Hash values of each basic block in the program are pre-computed before the execution of the program, loaded into the hardware monitor when the application is loaded for execution, and subsequently checked during program execution. To be able to check the instruction stream integrity, each row in the basic block lookup table, is augmented to contain another field that stores the statically-computed cryptographic hash of the instruction sequence in the basic block. During program execution, the monitor buffers the instruction stream corresponding to a basic block until a branch/jump instruction is encountered. At this point, the monitor switches to an empty buffer and enables the hardware hash unit to compute the hash of the buffered basic block. The computed hash value is then compared against the value stored in basic block lookup table. When the buffers are full, the processor is stalled in order to allow the instruction integrity checker to catch up[22][24]. The design in Figure 2.2 depicts the Hardware Assisted Monitoring Architecture.

Similar work to the Hardware Assisted Monitoring Architecture is done in [20][21].

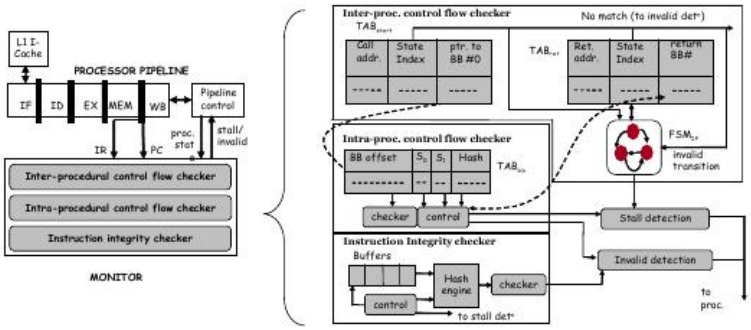

Figure 2.2: Hardware Assisted Monitoring Architecture[24]

The same idea of having a hardware monitoring unit, but the only difference with the work mentioned above is that the code is checked only at lower level of instructions. A generalized idea for monitoring code integrity at run-time in application-specific instruction set processors (ASIPs) is presented, where both the instruction set architecture (ISA) and the microarchitecture can be customized for a particular application domain. The monitoring microoperations are embedded in machine instructions, and the processor is modified with a hardware monitor. The monitor analyzes the processors execution trace of basic blocks at run-time, checks whether the execution trace aligns with the expected program behavior, and signals any mismatches [21]. Moreover, soft errors in microprocessors can also change program code and result in system malfunction. In [20], an approach for monitoring code integrity at run-time in ASIPs is presented similar to the one in [21] is presented where a compiler-assisted and application-controlled management design for the monitoring architecture is done.

The XOM allows the execution of instructions that are stored in memory but do not allow the execution of malicious instructions. The idea behind the XOM is that the Execute-only code which is stored in memory as encrypted form has to be decrypted when a load instruction occurs[41]. This means that the data are loaded to the processor chip in decrypted form. On the other hand data written to the memory on a store instruction has to be encrypted before it is send to the memory. According to [34], the software is given by the vendor in encrypted form. During the execution of the software on the target processor, it is decrypted using a secret key. Therefore, there exists an encryption/decryption unit between the cache and the main memory. On a read miss from the cache, the data from the memory are first decrypted and loaded into the cache.

On a write back to the memory the data are encrypted because the memory is not trusted, and the data verified for integrity. This is done by embedding in every data block a checksum [34]. On the other hand, the memory can be tampered from an adversary. To overcome this problem the memory hashing concept is used. The memory is considered as a structured tree where the program data are placed at the leaves of the tree. The nodes in the tree or the parent nodes contain the hash values of every data block at the leaves. When a cache miss occurs, the incoming data block is checked for integrity by checking its hash and all the parent hashes up to the root nodes. In case both values are

not the same, an exception is raised. When writing back the data into the memory, the hashes in the tree are updated including the root hash. Figure 2.3 explains the XOM overview and how the hashes are organized in the memory.

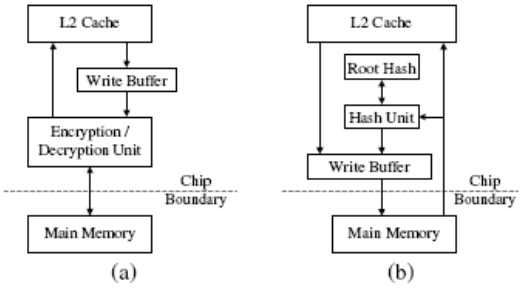

Figure 1: (a) XOM, (b) Memory Hashing

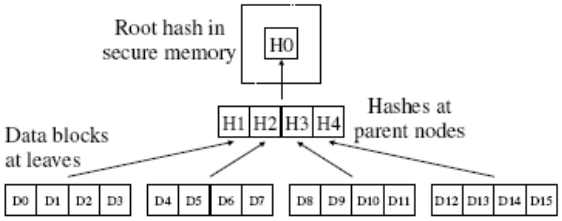

Figure 2: Memory organization in MH architecture

Figure 2.3: XOM overview and memory organization[34]

The limitations of XOM are the following [34]:

- Nowadays the software is unencrypted and some of the software is open source where XOM only protects encrypted code.
- XOM does not protect shared library code and does not fully protect the untrusted I/O channels.
- XOM does not detect the vulnerability or the alteration that happen during program execution.

The implementation of the XOM machine is described in details in [41]. We will discuss briefly the implementation of the XOM machine. The simple XOM machine is implemented by slightly modifying a CPU to add a special unit XOM Virtual Machine Monitor (XVMM), an on-chip private memory, microcode specially designed to store the private key[41]. The XVMM is implemented in software or in microcode. The XVMM code is a trusted, authorized, and privileged program. For this reason, the XVMM has access to the private key and the on-chip private memory [41]. The block diagram in Figure 2.4 depicts the architecture of the XOM.

Another similar work to the XOM architecture is the IBM4758 Secure Coprocessor [42]. The secure coprocessor provides a secure platform for the secure distributed applications where an application program can be executed without being tampered or

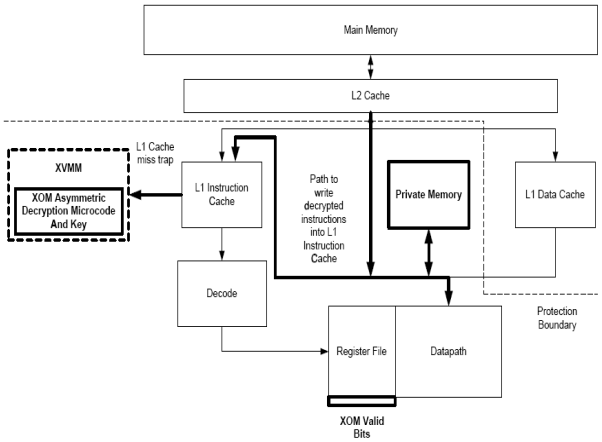

Figure 2.4: Simple XOM Architecture[41]

violated from an adversary with direct physical access to the device [42]. The secure co-processor must offer high computational and cryptographic resources and must be easily programmable. These requirements are met by using good cryptographic accelerators, adding Dynamic RAM (DRAM), and using smaller amount of battery-blacked RAM (BBRAM) as the non-volatile, secure memory [42]. The block diagram in Figure 2.5 shows the Hardware architecture of the high-end secure coprocessor.

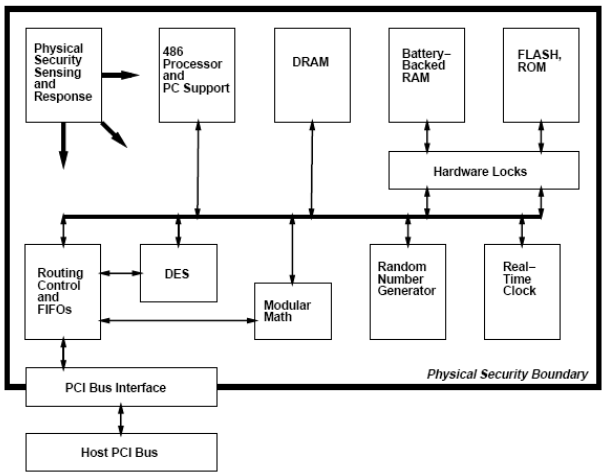

Figure 2.5: Hardware Architecture of High-end Secure Coprocessor[42]

The hardware architecture shown in Figure 2.5 must guarantee secure loading and execution of code. Physical attacks tamper the system by actually destroying the secret keys that are stored in the memory. This secure architecture detects the physical attacks that actually destroys the keys in the secure memory, it ensures that the secrets, when they are first loaded in the embedded system, are not known outside this

system, and despite the fact that many software programs are caused to attack, the secrets stored in the DRAM and BBRAM are not modified[42]. Another important example of Secure processors is the Trusted Computing Module architecture designed by the Trusted Computing Group (TCG)[44]. The Trusted Computing Group (TCG) is a non profit organization with the concern of improving the security of the computing environments in computer platforms by protecting the identity of the platform from being observed by an adversary or unauthorized entities [44]. The Trusted Platform provides at least three basic features like protected capabilities, integrity measurement and integrity reporting[44]. Shielded locations are locations where data can be accessed safely in memory and registers. Set of commands with exclusive permission that access the Shielded locations is called protected permissions. Shielded locations are used to protect and report integrity measurements [44]. According to [44], the TCG systems contain set of roots which have the concern of describing the embedded system characteristics that affect the trustworthiness of the system. Men distinguish between three types of roots of trust: a root of trust for measurement(RTM), root of trust for storage(RTS) and root of trust for reporting(RTR). The RTM is a computing engine which provides the reliable integrity measurements while the RTS is a computing engine which gives an accurate summary of values of integrity digests and the sequence of digests, on the other hand the RTR is a computing engine capable of reliably reporting information held by the RTS [44]. The block diagram in Figure 2.6 shows the architecture of the Trusted Platform Module(TPM).

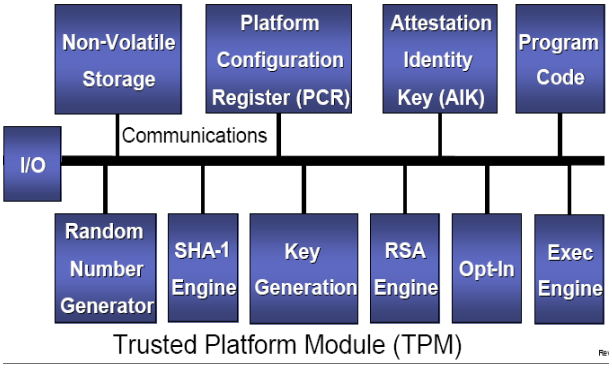

Figure 2.6: TPM Architecture[44]

Let us explain briefly the functionality of the components inside the TPM [44].

- Input/Output(I/O): The I/O component controls the flow of information over the communication bus by routing messages to appropriate components.
- Non-Volatile Storage: This component is used to store keys like the root key and the owner authorization data.
- Program Code: is a firmware that controls the devices in the TPM platform.
- Random Number Generator(RNG): The RNG generates the key.

- SHA-1 Engine: is a message digest engine, generates checksums.
- RSA Engine: is used to generate the encryption/decryption keys.
- Execution Engine: is used to execute the program code.

2.3 Secure Processor Architecture

This section describes the different steps of how a secure system works. The authority in a secure processor skips normal execution, changes into the secure mode execution, and checks for the integrity of the binary code where the binary code is loaded in the ROM. The authority software determines the checksums over each cache line prior to the execution of the code and the hashes are stored in every cache line. The block diagram in Figure 2.7 gives an example of a Secure System.

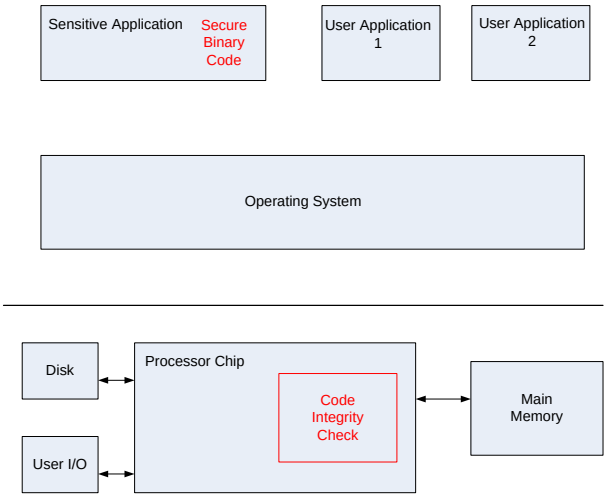

Figure 2.7: Secure System Overview[1]

According to the related work in section 2.2, we selected to implement the idead of Code Integrity Check in hardware like it is explained in the REM(Runtime Execution Monitoring). The reason is that the Code Integrity Check guarantees that the secure instructions remain unmodified throughout its execution. The secure instruction code is signed by computing a checksum over pieces of code, and embedding the hash into the binary code[1]. Figure 2.8 depicts the calculation of checksums statically and embedding them in the cache line. During execution, when the instructions are fetched from off-chip memory into the instruction cache. The processor will verify the integrity of the instruction code by computing the checksum at run-time and comparing it with the stored hash already pre-computed [2]. If both hashes are the same, this implies that the instruction code is secure otherwise the system has been tampered. If the hash check passes, the embedded hashes in the cache line are discarded and replaced with no-op instruction so as not to affect the execution of the processor[3].

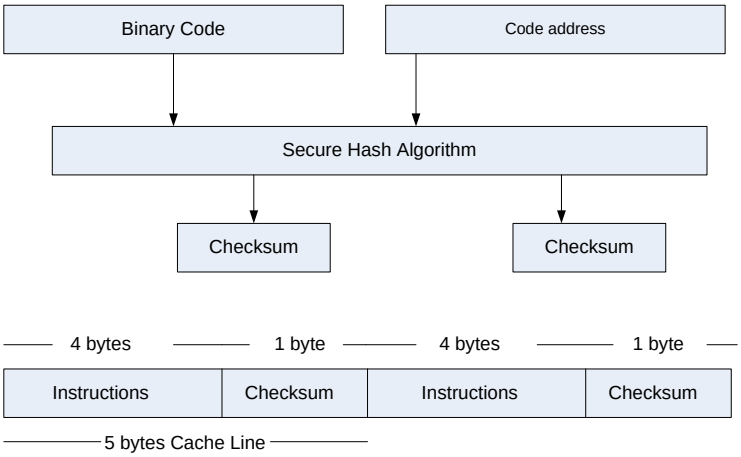

Figure 2.8: Code Integrity Check[1]

2.4 Secure Hash Algorithm(SHA)

A hash function is a function converting binary strings of arbitrary length to binary strings of some fixed length, called checksums[6]. The hash functions are a one-way functions that compute a small fixed length output value, the digest message. One of the trivial characteristics of the hash functions is that no information of the data input can be obtained. It is seldom to have two different data messages generating the same hash value[4]. The uses of hash functions are for digital signatures. The entity receiving the message then hashes the received message, and verifies that the received signature is correct for this hash value [6]. Hash functions can also be used for code integrity check. The hash value of a particular input is computed. To check that the input data has not been altered, the hash value is recomputed using the input message, and compared for equality with the original pre-calculated hash value [6].

According to [4], the Secure Hash Algorithm (SHA-1) computes a 160 bit message digest or output hash value from the input message. The input data stream is separated into multiple input bocks of 512 bits each. The input block is split into 80*32 bits words, one 32-bit word for each computational round of the SHA-1 algorithm. Each round comprises additions and logical operations, such as bitwise logical operations and bitwise rotations to left. Figure 2.9 illustrates the pseudo code of the SHA-1 algorithm.

The Secure Hash Algorithm (SHA-1) contains two stages of calculations: preprocessing and hash computation[13]. The preprocessing stage converts a message M which is a sequence of bits of arbitrary length L into m-bit blocks[13][18]. At this stage, the message M will be padded and parsed into m-bit blocks. The purpose behind padding is to guarantee that the padded message is a multiple of 512 bits. This is done by concatenating the message M by one bit with value of 1, followed by k bits 0, (k being the least non-negative solution to the expression $(L + 1 + k) \bmod 512 = 448$). The length of the padded message will become a multiple of 512 bits. After the message has been padded it will be parsed into several m-bit block before starting with the calculation of

```
For each data_block do

    Wt = expand(data_block);
     A = DM0 ; B = DM1;  C = DM2; D = DM3; E = DM4;

     For t = 0 ; t <= 79; t = t + 1 do
       Temp = Rotl(A) + f_t(B,C,D) + E + K_t + W_t ;
        E = D;
        D = C;
        C = Rotl(B);
        B = A;
        A = Temp;
      End for

    DM0 = A + DM0; DM1 = B + DM1; DM2 = C + DM2;
    DM3 = D + DM3; DM4 = E + DM4;
End for;
```

Figure 2.9: Pseudo Code for SHA-1 function[4]

the checksum value. The last step in the preprocessing stage is to set the initial hash values.

The second stage is the hash computation. The SHA-1 algorithm that is already described in Figure 2.9 performs 80 rounds. Every round uses a 32-bit word obtained from the current input data block. Since, each data block only has 16*32 bit words (512 bits), the rest of the bit words which are 64*32 bit words are obtained from data expansion. The following formula deduced from [4] shows the remaining 64*32 bits words how they are obtained.

$$W_t = Rotl_i\left(W_{t-3} xor W_{t-8} xor W_{t-14} xor W_{t-16}\right) \tag{2.1}$$

The computation of the final checksum is an eighty iteration algorithm over each message block, where every message block is a sequence of 32 bit words. At the beginning of every computation, the five internal registers are initialized with the initial hash values and at the end of every round computation each of the five registers contain the current hash value [18]. Figure 2.10 illustrates the operation in a single step of the Secure Hash Algorithm:

Let us explain what every rotational and logical operation does. The logical operations ft and bitwise rotations to the left RotL are involved in every round computation. The calculation of ft depends on the round t being executed as well as the value of the constant Kt [4]. Table 2.1 shows the logical functions and the values of the constants for every group of 20 rounds.

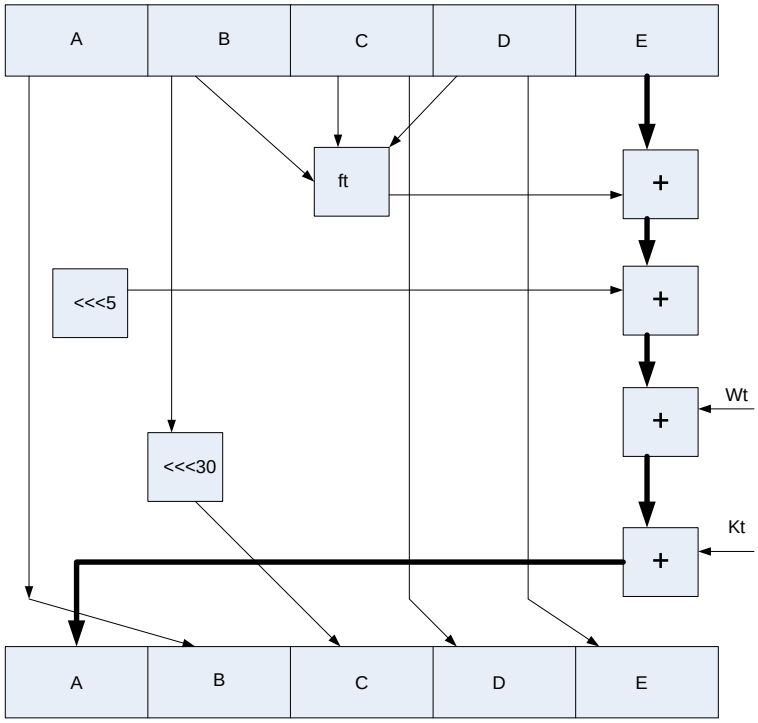

Figure 2.10: Operation in a single step of SHA-1[17]

Table 2.1: SHA-1 Functions and Constants[4]

Rounds	Function	Constants
0 to 19	(B and C) xor ((/B) and D)	0X5A827999
20 to 39	B xor C xor D	0X6ED9EBA1
40 to 59	(B and C) xor (B and D) xor (C and D)	0X8F1BBCDC
60 to 79	B xor C xor D	0XCA62C1D6

2.5 SHA-1 Hardware Implementation

This section describes the efficient implementation of Secure Hash Algorithm (SHA-1) in hardware. First the design of the SHA-1 unit is described in general. The pre-processing unit performs the appending of the input message, padding it with zeros and forms message blocks of the fixed length 512 bits for an input message of variable length. The pre-processing unit forwards message blocks to the message scheduler unit. The Message scheduler unit computes message dependent words, W(t), each of the message word is 32 bits. The message digest unit calculates the checksums. At every step, the message digest unit computes the hash value of a new word generated by the message scheduler unit. The message digest unit is the most crucial part of the implementation, as it determines both the performance and area of the whole design[39]. The block diagram

in Figure 2.11 describes the general SHA-1 architecture.

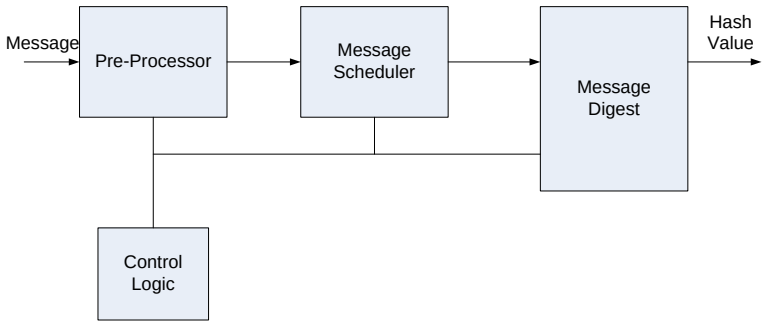

Figure 2.11: General SHA-1 Implementation[39]

On the other hand, the data dependencies of the algorithms SHA-1 do not allow for efficient pipelining. The computation time of the message digest unit is very high[4]. Some work has been done in [4] to improve the computational throughput by unrolling the calculation structure, but at the expense of more hardware resources [4]. Some methods are proposed in [4] to improve the hardware implementation of the SHA-1 algorithm. The most important ones are:

1. Parallel counters and balanced Carry Save Adders(CSA): are used to improve the partial additions .

2. Unrolling techniques: are applied to optimize the data dependency .

3. Memory based block expansion structures: the embedded memories are used to store the constant values described in section 2.4.

4. Operation rescheduling: is used to improve the speed of the circuit.

The block diagram in Figure 2.12 shows the top level design of the SHA-1 core and the signals that interface this unit. The block diagram in Figure 2.13 shows the structural design of the SHA-1 core with the interface signals. The SHA-1 core consists of two units which are the SHA-1 round unit and the Data Expansion unit.

2.6 Proposed SHA-1 round with operations rescheduling and data expansion unit

From Figure 2.9 in section 2.4, we can conclude that the SHA-1 round computation is based on the calculation of the A value. The remaining values do not require any computation, except the value of the rotation of B. The value of A depends on its previous value, no parallelism can be directly deduced, as depicted in the following formula[4]:

$$A_{t+1} = RotL^5(A_t) + [f(B_t, C_t, D_t) + E_t + K_t + W_t] \tag{2.2}$$

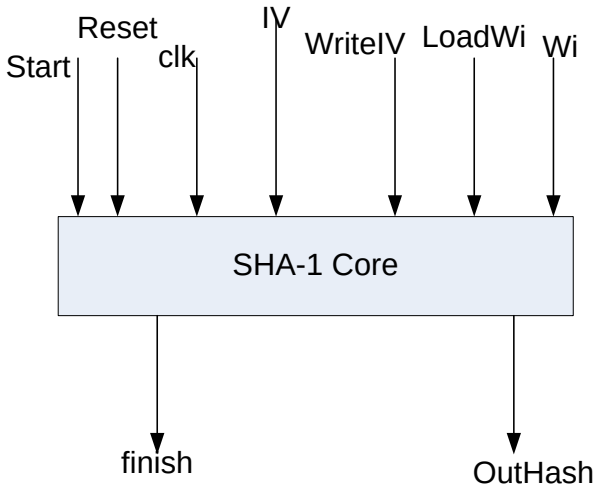

Figure 2.12: Top level design of SHA-1 core

The operation RotL(At) depends on the variable A(t), the remaining terms depend on variables that require no calculation and do not depend on the value of A(t), therefore some rescheduling in operations can be done. From equation (4.1), we can rearrange the terms that are not dependent of the variable A by producing the carry Carry(t) and save S(t). Thus, the equation becomes[4]:

$$S_t + Carry_t = [f(B_t, C_t, D_t) + E_t + K_t + W_t] \tag{2.3}$$

If we combine equation (4.1) with equation (4.2), we get:

$$A_{t+1} = RotL^5(A_t) + (S_{t-1} + Carry_{t-1}), S_t + Carry_t = [f(B_t, C_t, D_t) + E_t + K_t + W_t] \tag{2.4}$$

According to [4], the critical path of the SHA-1 round unit is optimized by splitting the computation of the value of A and by rescheduling as it is shown in formula (4.3). This shows that the function f (B, C, D) and the partial addition operations are no longer in the critical path. Therefore, the critical path of the SHA-1 round unit is reduced to a three input full adder. The final values of the internal variables (A to E) are added to the current values after completing 80 rounds, where the current values remains unmodified until the end of each data block computation. The final summation is computed by one adder for each 32 bits portion of the 160-bit hash value. Figure 2.14 depicts the implementation of the SHA-1 round unit in hardware.

According to equation (2.1) in section 2.4, the 512 bits generated from the padded unit is expanded to obtain the 80 32-bit words as input to the SHA-1 round unit. This unit is implemented with registers and XOR operations. A select logic is also needed to select between the first 16 rounds and the remaining rounds. Figure 2.15 shows the design of the SHA-1 data expansion unit.

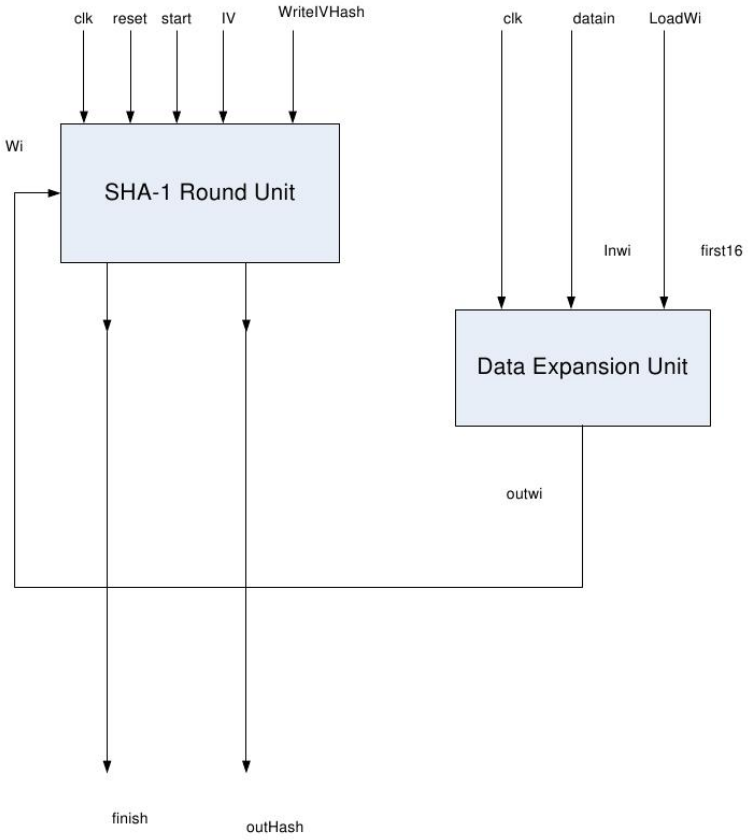

Figure 2.13: Structural Design of the SHA-1 core

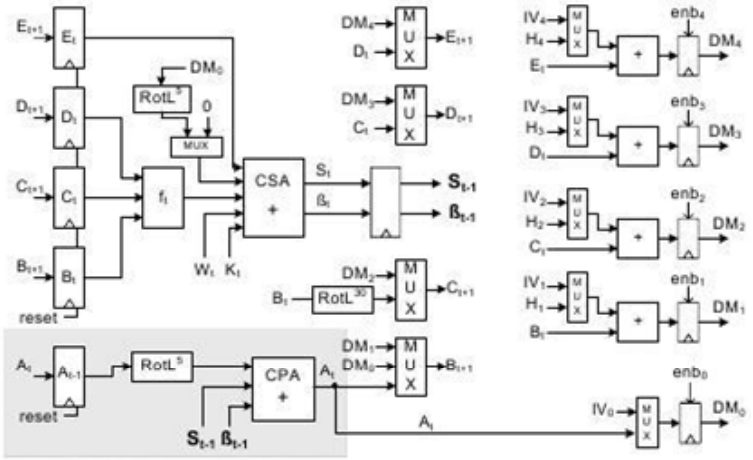

Figure 2.14: SHA-1 round unit Implementation[4]

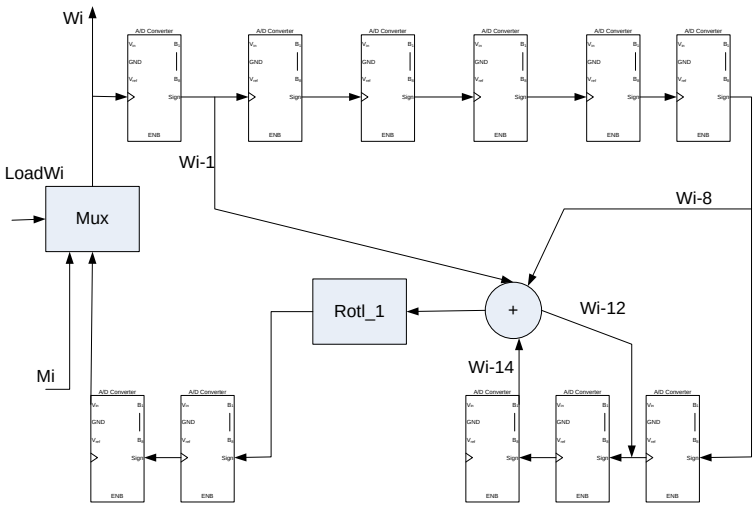

Figure 2.15: SHA-1 data expansion Implementation[4]

2.7 Conclusion

Early in this chapter, some security issues were discussed. Different classes of attacks into computer systems were described like the Hardware-Based. Attacks where an adversary has direct physical hardware access, the Software-based Attacks that are specially used remotely to break into computer and embedded systems. Some of the common software based attacks that were explained are Buffer Overflow, Heap Overflow, Double free vulnerability, format string attack and Temporary file vulnerability. The architecture of the design of the Secure Processor was discussed. A hash function which is a function converting binary strings of arbitrary length to binary strings of some fixed length called the checksums was discussed. The hash functions are one-way functions that computes a small fixed length output value, the digest message, that is highly correlated with the input data. One of the trivial characteristics of the hash functions is that no information of the data input can be obtained. The efficient implementation of Secure Hash Algorithm (SHA-1) in hardware was described.

3 Selection of the processor

This chapter discusses the 32-bit LEON processor that complies to the SPARC V8 architecture. Section 3.1 describes how the choice of the Leon processor was made. Section 3.2 gives an introduction for the Leon processor. Subsection 3.2.1 describes the Leon Integer Unit that consists of five pipeline stages. The on-chip memory in LEON3 that is implemented with separate instruction and data busses, and how to configure the cache direct map or set associative and the size of the cache are discussed in subsection 3.2.2. The memory controller of the Leon-3 processor that controls a memory bus holding external memory devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). The AMBA (Advanced Microcontroller Bus Architecture) is an on-chip bus specification for interconnection and organization of various functional modules that are a part of System-on-Chip are explained in subsection 3.2.3. Subsection 3.2.4 explains about the compiler of the Leon3 processor.

3.1 Choices while selecting the processor

A choice has to be made between using a hard-core or a soft-core processor to implement the envisioned extensions. First of all, we have tried to implement the code integrity check targeting the Xilinx Virtex-II Pro that uses the PowerPC 405D5 processor core, which is a 32-bit high performance, low power, scalar RISC(Reduced Instruction Set Computer) architecture, using separate data and instruction caches. Since the PowerPC is embedded into the virtex-II Pro device this means that the hard IP cores are diffused at any place within the FPGA platform, connected with the different neighboring Configurable Logic Block(CLB)[35].

The choice of implementing cryptography in hardware where the Xilinx Virtex II Pro is used as the FPGA platform is going to be rejected. The reason is that none of the IP cores can be modified, all what can be done is to add an extra IP core on the OPB bus. Another reason might be the fact that the hard-core PowerPC has limited design flexibility. On the other hand, the soft-core platforms has better design flexibility than the hard-core processor. According to [32], a soft-core can be eliminated from the design when it is not needed. Therefore, a reduction in area is going to be less. On the other hand, a soft-core uses a lot of resources from the FPGA, while the hard-core has its own hard-wired hardware. Another positive point of soft-core is that it can be customized to the requirements of the designer and it can be reconfigured at run-time to meet the constraints. On the other hand, the hard-core is a lot more optimized for the silicon-technology than the soft-core.

The next choice is to study three soft-core processors: the Leon-3, the Xilinx Microblaze and the OpenRISC 1200 RISC Core. The Leon-3 processor is a VHDL model that is fully synthesizable 32-bit processor implemented with SPARC V8 standard.

1. Integer Unit: The IU (Integer Unit) contains 32-bit RISC architecture five stage instruction pipeline, 8 global registers, 2-32 register windows of 16 registers each and 32-bit instructions [35].

2. Cache: The cache is implemented with a harvard model (separate instruction/data cache). Instruction and data cache size are modified from 1 KB to 64 KB. The cache contains direct mapped or multi-set cache with set associativity of 2-4. The cache lines can be modified between 8 and 32 bytes of data [35].

3. Memory: The memory controller has an interface with the PROM, SRAM, SDRAM and memory mapped I/O devices. The memory contains 2 Gbyte address space. the memory areas can be programmed to 8-16-32 bit data width[35].

4. Design Tools: Simulation is done via a generic testbench and test program is available, including support files for Modelsim.

The second platform that is going to be discussed is the Xilinx Microblaze Softcore. The Microblaze is a 32-bit soft processor designed by Xilinx. It contains a RISC architecture with Harvard Model separate data and instruction busses. The main concern of developing the Microblaze platform is to build complex systems for networking, telecommunication, data communication and embedded systems[35].

1. Processor Unit: It contains 3-stage pipeline with 32-bit RISC architecture. Every instruction has 32 bits. The Instruction Set Architecture (ISA) has two types. The first type which is Type A contains two source and one destination operand while the second type contains one source and one immediate operand. It contains a RISC architecture with Harvard Model separate data and instruction busses but no cache[35]. The memory access can be done via the Local Memory Bus (LMB) and On-chip Memory Bus (OPB).

2. Cache: Separate instruction and data cache. The placement scheme is only direct-mapping.

3. Memory: The memory controller supports up to 8 memory (flash/SRAM) banks. It contains a separate control register for each bank. Moreover, it supports 8,16,32 and 64-bits bus interface [35].

4. Design tools: The Xilinx Software integrated development environment, which creates software like Standard C [35]. It contains the GNU C compiler tools including compiler, assembler.

5. Performance: System frequency is about 150 MHz.

The third platform is the OpenRISC 1000. This architecture is 32- and 64-bit RISC processors. It targets medium and high performance networking, portable, embedded, and automotive applications because it is designed to get a better performance, low power consumption[35].

1. Processor Unit: It is a scalar, single-issue 5 stage pipeline. It consists of a single-cycle instruction execution for most of the instructions.

2. Cache: It follows the Harvard model with split instruction and data cache. The instruction /data cache size can be configured from 1KB to 64 KB.

3. Design tools: It contains the GNU ANSI C, C++, Java and Fortran compilers[35].

4. Performance: System frequency is about 250 MHz.

The question remains which of the three soft-core is going to be used in this project. To answer this question, let us revise the differences between the three soft-cores. All three processors are 32-bit RISC processor big endian synthesizable pipelined processors. LEON-3 and OpenRISC 1200 contain 5-stage pipelines while the Microblaze contains 3 stages of pipeline. LEON and OpenRISC processors are available for free under LGPL license, on the other hand Microblaze is available by the company XILINX. According to the analysis done in [36], we have chosen the Leon-3 processor as soft-core to implement cryptography for the following reasons:

1. Cache: All three processors have Harvard caches with almost equal values for the cache size. The Microblaze and the OpenRISC 1200 implement in their placement scheme only the Direct mapping scheme while the Leon-3 provides support for a 2-4 way set associative cache configuration, in which three replacements strategies can be choosed. The choice of the cache is very trivial in this thesis because we need to have more flexibility in configuring the cache. Leon soft-core offers better flexibility than the remaining two. On the other hand the architecture of the cache of Leon-3 is more advanced and complicated than the cache of the remaining two[36].

2. Documentation: The documentation for LEON-3 is acceptable while the documentation of the Microblaze is a bit more extensive. On the other hand, the documentation of the OpenRISC 1200 is not so extensive and well documented like the other two cores[36].

3. Tools: The configuration tools of the Leon-3 and the Microblaze are very easy to change even the configuration tools of the Leon-3 is easier to use and to access than the Microblaze. On the other hand, the OpenRISC 1200 configuration is done manaully. HDL simulation and debugging is less complex for Leon-3 compared to the other two processors[36].

3.2 Introduction to Leon Processor

Leon-3 is a 32-bit processor that complies to the SPARC V8 architecture. The Leon-3 processor is a soft-core processor that can be configured and made suitable for embedded applications and System-on-Chip designs with the following features on chip: separate instruction and data caches, hardware multiplier and divider, interrupt controller, two 24 bit timers, two UARTs, power-down function, watchdog, 16 bit I/O port and flexible memory controller. Additional modules can easily be added using the on-chip AMBA

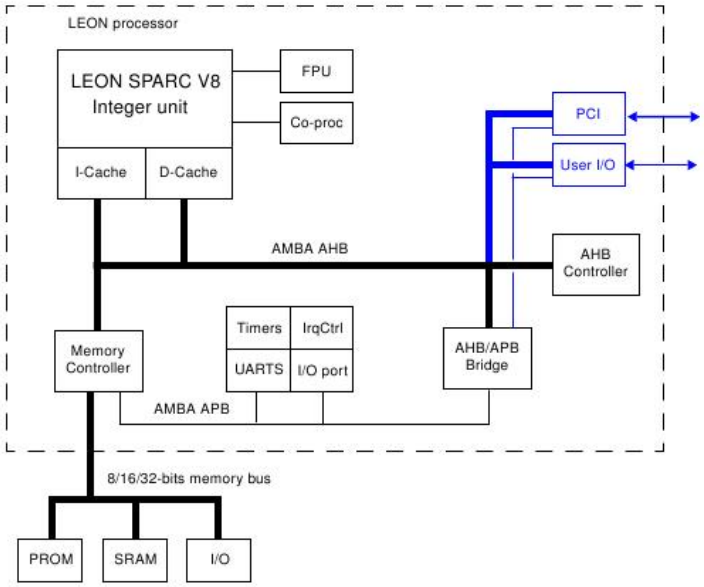

Figure 3.1: LEON block diagram[8]

AHB/APB buses[7]. All the features of Leon processor are illustrated in the block diagram of Figure 3.1.

3.2.1 Leon Integer Unit

The LEON integer unit (IU) is implemented using the SPARC Version 8 integer instructions. The Leon integer unit has the following features:

1. 5-stage instruction pipeline.

2. Separate instruction and data caches (Harvard Architecture).

3. Support for 2 - 32 register windows.

A block diagram of the LEON Integer Unit can be seen in figure 3.2 [8].
The Leon Integer Unit as defined in [8] consists of five pipeline stages and they are the following:

1. FE (Instruction Fetch): In this stage the next instruction to be executed is fetched from the memory. In case the instruction cache is enabled, then the instruction fetch is done directly from the instruction cache.

2. DE (Decode): In this stage the instruction is decoded and the operands are read. The operands may come from the register file or from internal data bypasses. The CALL and Branch target addresses are also determined.

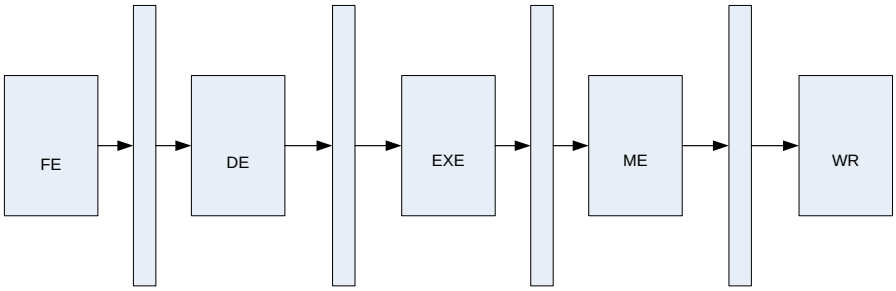

Figure 3.2: LEON integer unit block diagram

3. EXE (Execute): In this stage, the arithmetic operations are performed in the ALU (Arithmetic Logical Unit). Logical and shift operations are also computed in the ALU. Moreover, the address of memory operations like load , store, jump and return will be determined.

4. ME (Memory) : In this stage, the data is valid at the end of the stage. For data write, the data will be written to the data cache during the execute stage.

5. WR (Write): In this stage, the outcome of any arithmetic, logical, shift, and cache read operations are written back to the register file.

3.2.2 Instruction and Data Cache System

The on-chip memory in LEON3 is implemented with separate instruction and data buses. The LEON-3 processor uses a Harvard Architecture. Both instruction and data cache are connected to two independent cache controllers where these last can be configured to implement either a direct mapped or a multi-set cache with set associativity of 2-4. The set size is configured from 1-256 KBytes. The simplest one is where for instance the Leon instruction cache is configured from 1 - 64 Kbytes. It is a direct-mapped cache[28]. Instruction and data cache operations are controlled via a Cache Control Register(CCR). Every cache can be operating in one of the three modes: enable, disable and frozen. In case where the cache is in disable mode, no memory operations like load and store are performed. The frozen mode operation keeps the cache synchronized with the main memory as if it was in the enable mode, so the cache can be accessed but whenever there are misses the cache line will not be updated from the memory. In the enable mode, the instruction cache is divided into cache lines with 8 - 32 bytes of data assuming direct mapped cache. In this mode, the cache line is filled from main memory. The instructions are forwarded at the same time to the Integer Unit(IU) or processor pipeline. Sometimes, due to internal dependencies or multi-cycle instruction, then the processor pipeline is halted until the line fill is completed. In case where the processor pipeline executes a control transfer instruction like branch or call instruction during the line fill, the termination of the line fill will be on the next fetch. Every cache line has a cache tag associated for it. The cache tag consists of an address tag (ATAG) and valid (V) bits[8][28].

On the other hand, the Leon data cache is configured also from 1-64 Kbyte. It is a direct-mapped cache. The data cache is divided into cache lines with 8 - 32 bytes of data. Every cache line has a cache tag associated for it. The cache tag consists of an address tag (ATAG) and valid (V) bits. The following two figures show the instruction and data cache tag respectively:

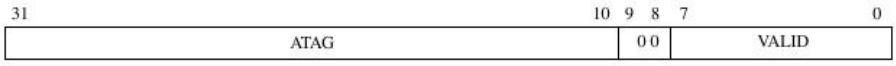

Figure 3.3: Instruction cache tag layout[8]

1. Address Tag (ATAG)[31:10]: tag address of the cache line.
2. Valid (V) [7:0]: When a sub-block of cache line is filled with data, then the valid bits are set. A cache fill which results in a memory error will leave the valid bit unset. A flush instruction will clear all valid bits.

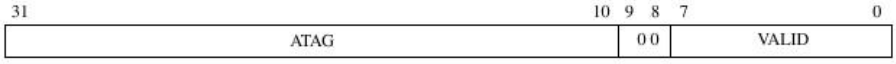

Figure 3.4: data cache tag layout[8]

1. Address Tag (ATAG) [31:10]: tag address of the cache line.
2. Valid (V) [7:0]: When a sub-block of cache line is filled with data, then the valid bits are set. A cache fill which results in a memory error will leave the valid bit unset.

3.2.3 Memory Access and AMBA on-chip buses

The memory controller of the Leon-3 processor controls a memory bus holding external memory devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). The memory controller acts as a slave on the AHB (Advanced High Speed) bus. The memory bus supports four types of devices: prom,sram,sdram and local I/O. The memory bus can also be configured in 8-bit or 16-bit for applications with low memory and performance demands[28]. Figure 3.5 shows the interface between the AHB bus and the external memory. The controller decodes three address spaces (PROM, I/O and RAM) whose mapping is determined through VHDL generics. The controller decodes in total a 2 Gbyte address space. The following table shows that:

AMBA (Advanced Microcontroller Bus Architecture) is an on-chip bus specification for interconnection and organization of various functional modules that are a part of System-on-Chip. The AMBA specification improves the reusable system on-chip platform by including a common standard for data communication in a System-on-Chip module[37]. Men distinguish three distinct AMBA buses and they are:

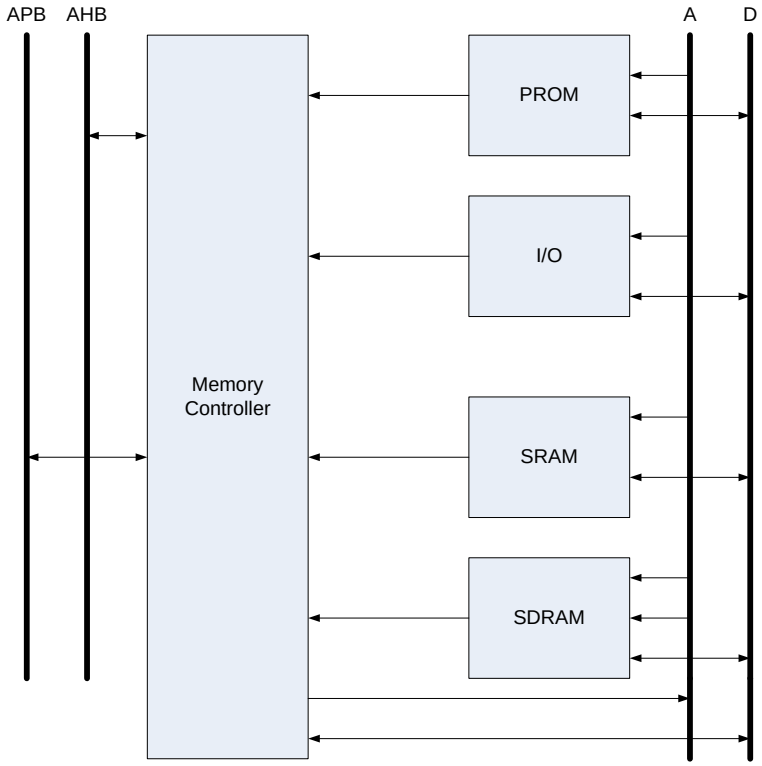

Figure 3.5: Memory controller connected to AMBA bus and external memory devices[28]

Table 3.1: Address Space map [8]

Address range	Size	Mapping
0X00000000-0X1FFFFFFF	512 M	Prom
0X20000000-0X3FFFFFFF	512 M	I/O
0X40000000-0X7FFFFFFF	1 G	RAM

1. Advanced High-performance Bus (AHB): The AMBA AHB is used for system modules with high clock frequency. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces.

2. Advanced System Bus (ASB): The AMBA ASB is for systems with high performance. AMBA ASB is an alternative system bus suitable for use where the high-performance features of AHB are not required.

3. Advanced Peripheral Bus (APB): The AMBA APB is used in systems where low-power peripherals exists. AMBA APB is optimized for minimal power consumption.

The AMBA AHB system consists of the following components[38]:

1. AHB master: The bus master initiates read and write operations by providing informations about the addresses of the data. One master is only allowed to actively use the bus.

2. AHB slave: A bus slave responds to a read or write operation within a given address-space range. The bus slave replies back to the active master the success, failure or waiting of the data transfer.

3. AHB arbiter: The bus arbiter guarantees that only one bus master at a time is allowed for data transfers.

4. AHB decoder: The AHB decoder is used to decode the address of each transfer of data and provides a select signal for the slave that is involved in the transfer.

The AMBA ASB has the same components as the AMBA AHB. The following figure shows all three bus systems and the cores connected to the three busses.

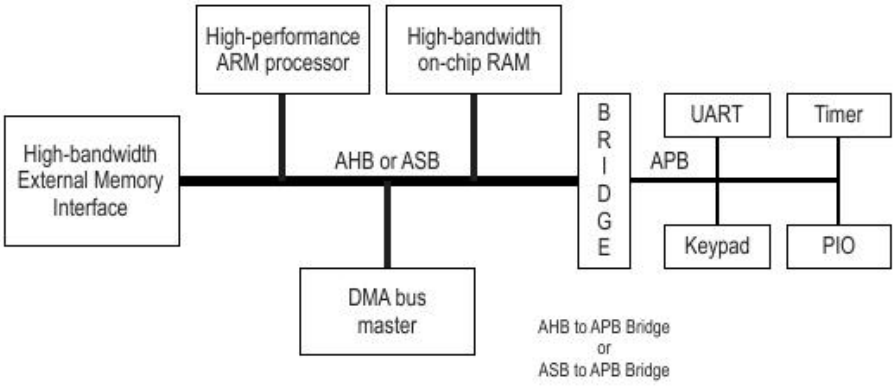

Figure 3.6: AMBA AHB/ASB and APB Bus [38]

3.2.4 Bare-C Compiler

BCC(Bare C compiler) is a cross compiler for LEON-3 processors. It is based on the GNU compiler tools and the Newlib standalone C-library [9]. The cross compiler supports floating point operations, as well as SPARC V8 multiply and divide instructions. For further information check [9]. The next thing was to compile a very simple program like hello world and this is done with the following command : sparc-elf-gcc -msoft-float -g -O2 hello.c -o hello.exe.

The compiler has some extra options like -msoft-float, it emulates floating point and it is used if no Floating Point Unit exists in the system. The other option is -O2 where it optimizes code maximum performance and minimal code size, for more informations about the compiler options check [9]. After the simple code of hello world has been compiled, the next step was to learn how to make Leon boot the PROM and this is done with the command sparc-elf-mkprom. Note that sparc-elf-mkprom creates ELF files. To

create an SRECORD file for a prom program, the command sparc-elf-obj is used. Let us summarized the steps from compilation, linking and copying from ELF to SRECORD:

1. sparc-elf-gcc -g -O2 hello.c -o hello -msoft-float
2. sparc-elf-mkprom hello -o hello.exe -msoft-float

   MKPROM boot-prom builder v1.0

   section: .text at 0*4000000, size 31040 bytes section: .data at 0*4007940, size 1904 bytes
3. sparc-elf-objcopy -O srec hello.exe hello.srec

3.3 Conclusion

In this chapter, three soft-cores was studied, the Leon-3 processor, the Xilinx Microblaze and the OpenRISC 1200. The 32-bit LEON processor that complies to the SPARC V8 architecture was discussed. The Leon Integer Unit as defined that consists of five pipeline stages, the on-chip memory in LEON3 is implemented with separate instruction and data buses, and how to configure the cache direct map or set associative and the size of the cache were illustrated. The memory controller of the Leon-3 processor controls a memory bus holding external memory devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM) were also explained. AMBA (Advanced Microcontroller Bus Architecture) is an on-chip bus specification for interconnection and organization of various functional modules that are a part of System-on-Chip. The AMBA specification that improves the reusable system on-chip platform by including a common standard for data communication in a System-on-Chip module and the three distinct AMBA buses which are Advanced High-performance Bus (AHB), Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). The compiler of the Leon3 processor is also explained.

4 Implementation and Validation

This chapter describes the implementation of the Code Integrity Check (CIC) unit in hardware using the Leon3 Template. Section 4.1 discusses how the emulated certified authority reads the binary file of the compiled program, assigns the instructions into a certain number of bytes, calculates the checksums with the SHA-1 algorithm, embeds the checksums to the cache line that contains the instructions and loads them in the binary file. Section 4.2 describes the design of the hardware CIC (Code Integrity Check) unit that consists out of some key components called the controller of the cryptographic unit, the cryptographic unit (SHA-1 core), additional control units to detect whenever the data read from the cache is valid, selection logic components and a compare unit.

4.1 Certified Authority Emulation

The first step in implementation was to add to the binary file generated from the sparc-elf-gcc compiler the checksum calculated for every cache line in the binary file. A software program is implemented in C language. The binary file is generated after compiling with the sparc-elf-gcc compiler of the Leon-3 processor and loaded in the PROM of the LEON-3 soft-core. The certified authority reads the binary file of the compiled program, assigns the instructions into a certain number of bytes, calculates the checksums with the SHA-1 algorithm, embeds the checksums to the cache line that contains the instructions and loads them in the binary file. Figure 4.1 illustrates the flow chart diagram that shows these steps.

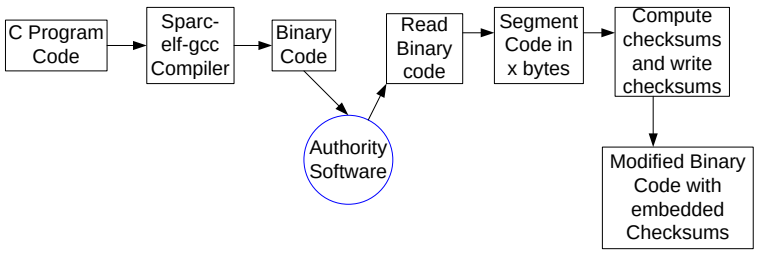

Figure 4.1: Certified Authority

A small example illustrates the overal functionality. The software program which is running on the LEON-3 processor, is shown in Figure 4.2. After compiling the software program, the binary file is obtained.
000081D8200003000004821060E081884000
00108190000081980000818000000A1800000
00200100000003002040821060OFC2A00040

00308410000001000000010000000010000000

```
int sum (int a , int b)
{
    int result;
    result = a + b;
    return result;
}

int mult (int a , int b)
{
    int result;
    if (a > b)
        result = b * a;
    else
        result = a * b;

    return result;
}
main()

{
    report_start();

    base_test();
    int result1,result2;
    int x = 2;
    int y = 3;
    result1 = sum(x,y);
    result2 = mult(x,y);
    printf("the sum is %d \n",result1);
    printf("the product is %d \n",result2);
    report_end();
}
```

Figure 4.2: Application Software

The first four characters correspond to 16-bit address. Every eight characters represent four bytes of instructions loaded in the memory. The authority software is going to read the binary file and computes the checksum for every line. The following checksums are generated.
ee750e0f90421b951f0761e37f9e2889a861e3b1
28552f99df09a02a2f6fbe219251aeb0c494a6b3
9496dc19a05cf4c0aad3d6ce3a4064165cce252a
536538f090b1ce3fbb3d35aeb028afb836f19ab5

The idea originally was to embed the checksums in the existed binary file. So we need to expand the cache to embed all these checksums in it. The checksums are instead stored in an on chip memory (look up table) where all the checksums are read from memory. This look up table is necessary because the SHA-1 computation latency is longer than the latency of looking up the checksums in the Block RAM. This is why we thought this is a representative implementation because also the costs of the SHA-1 are much more than the costs of additional logic we did not consider.

4.2 Implementation of the Code Integrity Check Unit

4.2.1 Architecture and Organization of the non extended processor

This subsection describes the standard implementation of the Leon3 processor. The general organization of the Leon-3 template consists of the memory controller which is the interface between the PROM of the Leon3 processor and the AMBA-AHB(Advanced High Speed) bus controller. On the other hand the Leon3 top level entity consists of the cache control unit which is the interface between the memory controller, the cache itself (Instruction and data cache) and the Integer Unit pipeline. The block diagram in Figure 4.2 shows the organization of the LEON-3 template. From Figure 4.2, the AHB controller is a combination of an arbiter, bus multiplexer and slave decoder. The LEON3S entity is a 32-bit processor core conforming to the SPARC architecture. The memory controller hosts a memory bus PROM. It acts as a slave on the AHB (Advanced High Speed) bus[28].

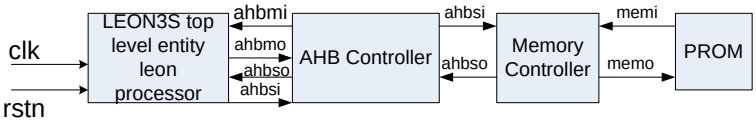

Figure 4.3: Organization of the Leon-3 template

The signals that interface the AMBA-AHB bus with the Leon3S top level entity and memory controller are *ahbmi* (AHB master input), *ahbmo* (AHB master output), *ahbsi* (AHB slave input), *ahbso* (AHB slave output). The interface signals between the memory controller and the PROM are *memi* (memory input) and *memo* (memory output). The block diagram in Figure 4.3 illustrates the architecture of the LEON3S unit.

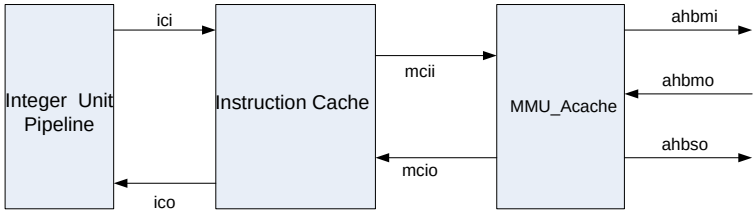

Figure 4.4: Organization of the LEON3S Unit

When a miss occurs in the instruction cache, it changes into the streaming mode, fetching one entire cache line, where the buffer *waddr* holds the next address to ask from memory [40]. Figure 4.4 illustrates the organization and behavior of the LEON Instruction cache. The next subsection explains the modification done to the datapath of the Leon3 processor.

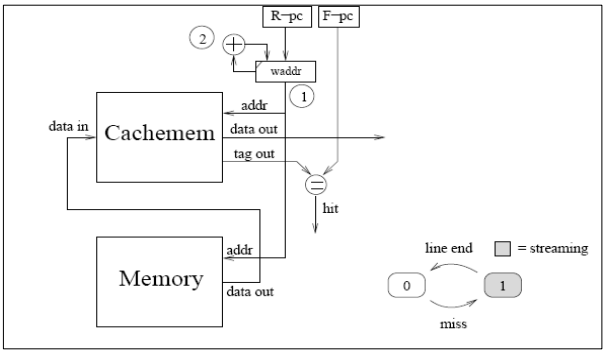

Figure 4.5: Architecture of Leon3 Instruction Cache[40]

4.2.2 Extension of The Leon3 processor

This subsection describes the implementation of the Code Integrity Check (CIC) unit in hardware using the Leon3 processor. The top level entity of the leon3 processor is modified to implement the cryptography in hardware. The datapath is extended by adding key components like the controller of the cryptographic unit, the cryptographic unit (SHA-1 core), unit to detect if the instructions read from the cache are valid, a block RAM memory to dump the pre-computed cheksums and a compare unit. The block diagram in Figure 4.5 shows the extension of the datapath with the Code Integrity Check.

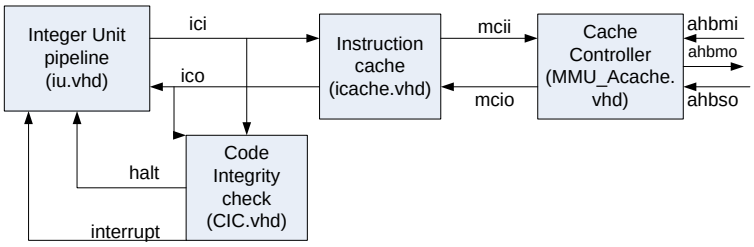

Figure 4.6: Extended Datapath

At run time the checksum is computed by the cryptographic unit and compared to the pre-computed checksum computed statically prior to the execution of the program code. The pre-computed checksums are stored in an on-chip Block RAM module. The pre-computed hashes of the cache lines are stored in BRAM at the same address of the cache line. The two checksums are compared when the signal start compare from the control unit is enabled. In case the two checksums values are not the same, this means that the binary code has been tampered and an interrupt is sent to the pipeline unit so that the processor stops the execution of the code. The block diagram in Figure 4.6 shows the structural design of the Code Integrity Check.

The Code Integrity Check unit is tested by a generic testbench. The generic test-

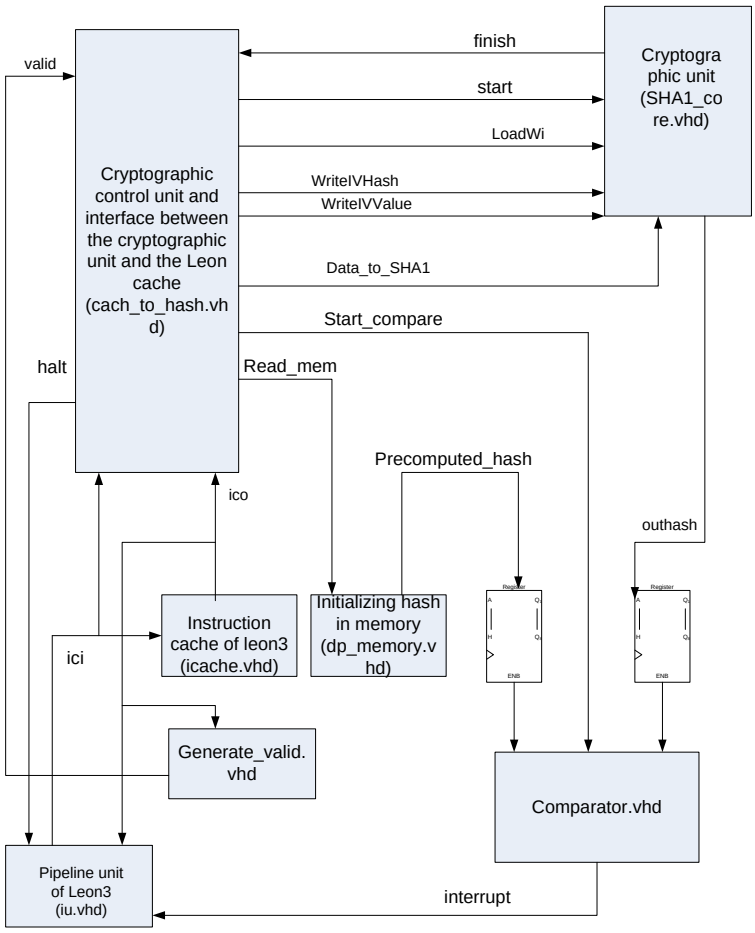

Figure 4.7: Code Integrity Check Unit

bench includes external Flash PROM which is pre-loaded with a test program. The test program will execute on the LEON3 processor and test various functionality in the design. The generic testbench has been created to read from file. It reads from the file of the PROM where the binary code is loaded. The binary code has been created after compiling a software test program like the one shown in Figure 4.2. Other test programs have been generated with nested for loops, while loops. For further details about the test program refer to Appendix B. All the interface signals between the processor chip and the external memory components are defined as internal signals. The generic testbench emulates the LEON processor where all the components that are inside this processor chip can be tested. The generic testbench emulates the top level entity of the Leon3 processor and all the components are mapped to this entity. The wave diagram in Figure 4.7 shows the evaluation of the design (the relevant signals are circled). According to the wave diagram in Figure 4.7, we can observe the following:

- Signal *mem-re* is enabled after one clock cycle, the pre-calculated *mem-hash* value is read from the offset address of the cache line *ici.dpc*. The offset of the address

0000000C is for instance 000 and this is the address of the pre-computed value.

- After 80 clock cycles, the run time hash value is computed and signal *start-compare* is enabled.

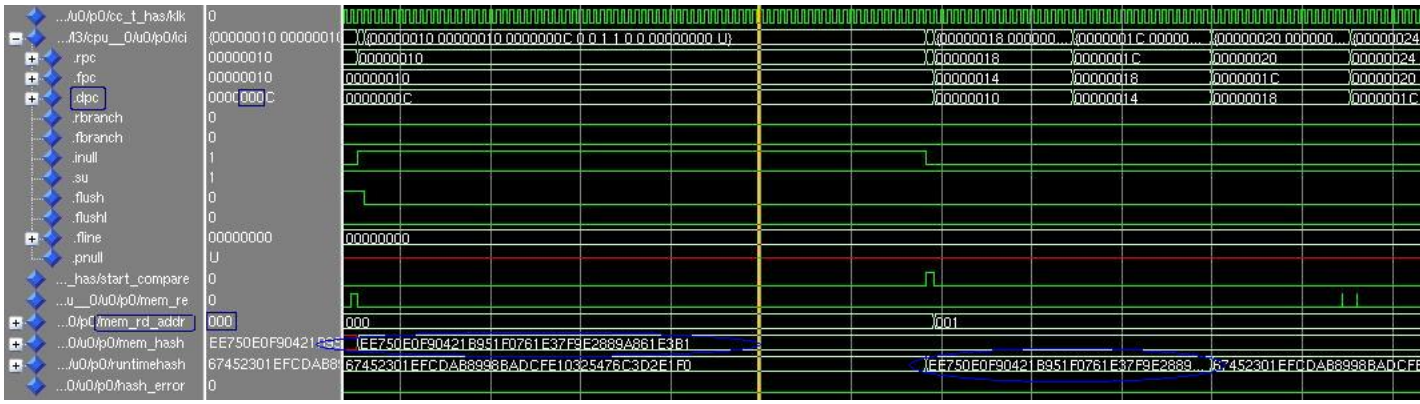

Figure 4.8: wave diagram of Code Integrity Check

4.2.3 Detection of valid Instructions from the Instruction Cache

The main concern is in detecting whether the instructions in the cache are valid. The instruction address is valid at the beginning of the Fetch stage of the pipeline and is generated in the execution stage from incrementing the program counter(pc) or from a previous branch. In case a miss is detected in the fetch stage, the instruction cache is switched into the streaming state, sending a memory request and at the same time the pipeline is in halt mode. When the memory request is ready, the data is latched in the pipeline by using the signal *ico.mds* which is an output from the instruction cache going as an input to pipeline unit[40]. Figure 4.8 illustrates the design of the generate valid unit.

Whenever the signal *ico.mds* is equal to zero, the instructions in ico.data(0) are ready in the instruction cache. The transition from state s0 to state s1 in the state machine in Figure 4.14 detects the valid signal, this means that in state s1, the instructions are valid in the cache. The simulation of the generate valid unit with all the interface signals are shown in wave diagram in Figure 4.9 where the relevant signals are circled in the waveform.

4.2.4 Cryptographic Control Unit

The main concern is controlling the cryptographic unit (SHA-1 core) by reading the instructions from the instruction cache if a valid has been generated from the generate valid unit. The SHA-1 core can accept any message of any size and digest the message into an output of 160 bits. The pipeline unit handles four instructions from the instruction cache. In our case the input to the cryptographic unit is 4 instructions each

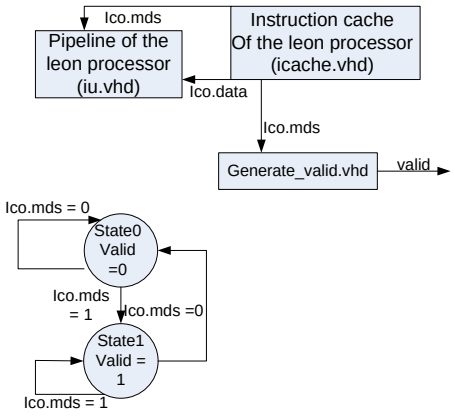

Figure 4.9: generate valid unit

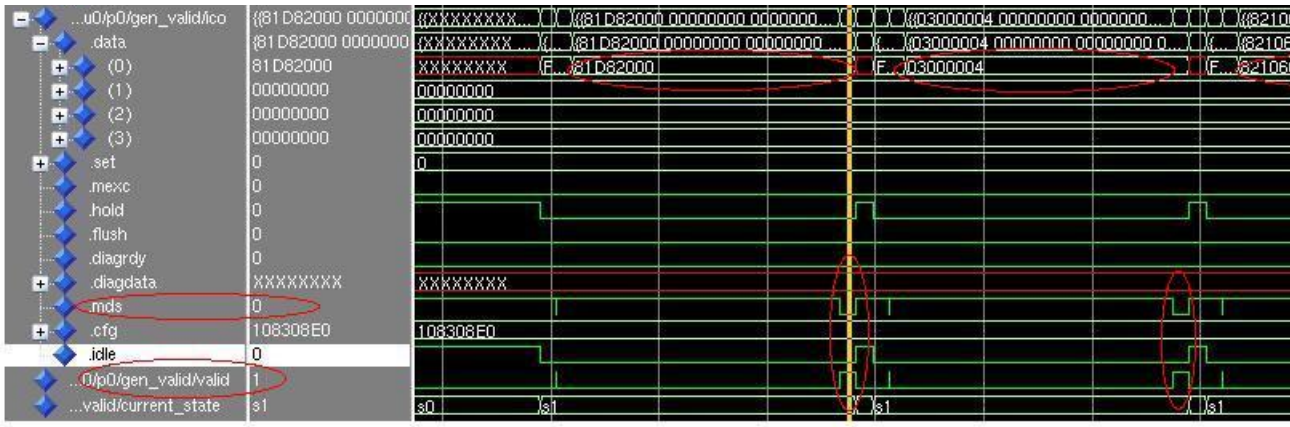

Figure 4.10: wave diagram

of 32 bits. On the other hand, the SHA-1 core performs pre-processing operations by appending the message by one and padding the message with zeros as it was described in section 2.4. The control unit makes sure that the cryptographic unit is reading the valid instructions from the cache, loads each of the four instructions in registers, enable the start signal for the SHA-1 core to begin computing the checksums, halts the pipeline of the processor during the computation of checksums, and makes sure that all the inputs to the cryptographic unit are appended with one, padded with zero and adding the size of the message at the end of the input. The block diagram in Figure 4.16 illustrates the design of the unit in hardware.

The Finite State Machine in Figure 4.12 illustrates the design of the SHA-1 core controller the cache to hash unit.

- The cache to hash unit describes the interface between the instruction cache of the Leon-3 processor and the cryptographic unit(SHA-1 core). The output of the

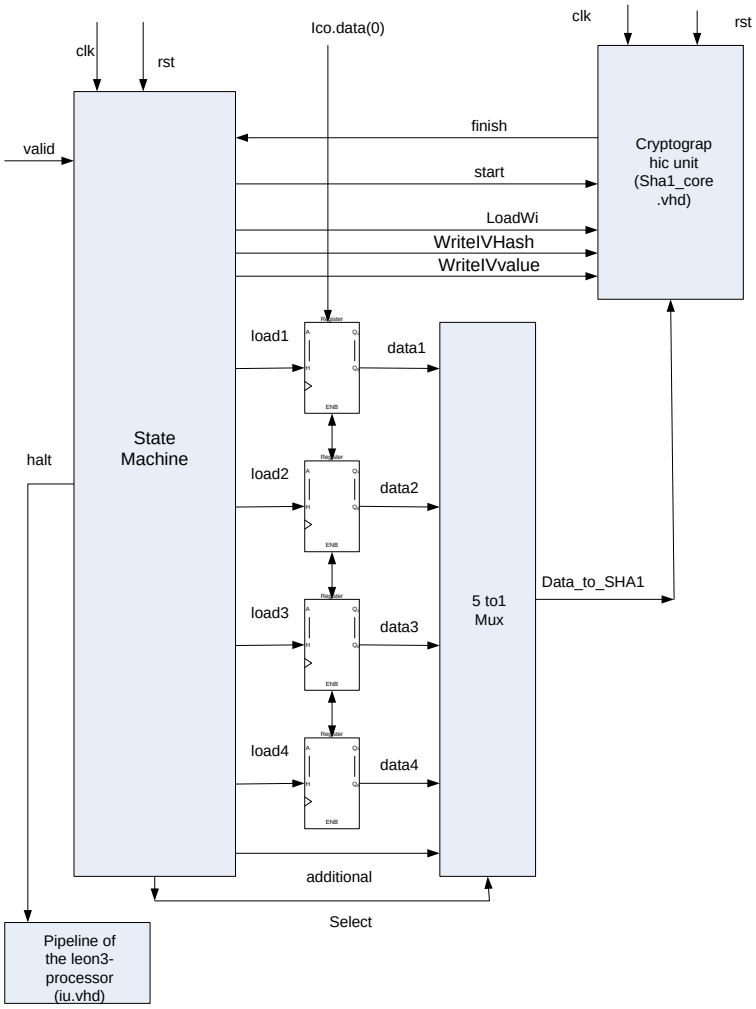

Figure 4.11: Cryptographic Unit Interface

instruction cache *ico.data(0)* is assigned as an input to the control unit. Whenever the valid signal is detected from the generate valid unit, then the cryptographic unit is reading the valid instructions.

- The transition from state S1, S2, S3 to state S4 in Figure 4.17 describes that the registers are enabled and the instructions data are loaded in the registers. The first four states take into consideration the branch instructions. In case of a branch instruction the special state jump handles those branch instructions, the remaining instructions and the embedded checksum in the cache line are neglected since there is a jump to another address in the code. This state jump checks whether there *branch* is still equal to 1 or the offset of the address of the cache line not equal to zero. In this case we stay hanging in this state until there is no more branch or the offset of the address is equal to zero.

- In state S4 the cryptographic unit is enabled by setting the *start* signal to one and the pipeline of the processor is halted. From state 5 to state 20 the pipeline is halted and the cryptographic unit is doing computation work.
- The signal *LoadWi* is enabled once the valid instructions are loaded in the registers. The select signal of the multiplexers select the relevant data as an input to the cryptographic unit. The select signals select also the additional data, this means that the value for the appending, padding with zeros and the length of the input message is also generated from the cache to hash unit.
- At state 21 the cryptographic core is done with the calculations of checksums and the state transition goes back to state 0, to start the next computation round.

4.2.5 Timing Simulation Results

This subsection shows the simulation results of the cryptographic control unit in case a binary code is running from the main function and a binary code where the code contains jumps instructions. The wave diagram in Figure 4.12 illustrates the time simulation of the cache to hash unit or the cryptographic control unit, the interface signals and the state transitions of a binary code running from main. The expected behavior is explained in subsection 4.2.4. (The relevant signals are circled). According to the wave diagram in Figure 4.12, we can observe the following:

- The four instructions which are *ico.data(0)* in the simulation wave diagram are loaded in registers reg1, reg2, reg3 and reg4 when the signals *load1*, *load2*, *load3* and *load4* signals are enabled.
- The *Loadwi* signal of the cryptographic unit is enabled, immediately after the start signal is enabled, the instructions are loaded to the cryptographic unit and the *LoadWi* signal is enabled for a certain period of time.
- During the computation of the checksums, the *halt* signal is enabled until the computation is finished, so that the processor stops the execution of the instructions.
- This simulation is indicated only for the case of basic blocks so it does not include any jumps instructions. The binary code runs only from the main function.

The wave diagram in Figure 4.13 shows the time simulation of the cache to hash unit taking into account the jump instructions. The binary code includes jumps instructions. The expected behavior is explained in subsection 4.2.4. In case of jump instruction the transition from the current state(S0... S3) to state jump takes place. The state jump handles the branch instructions, the remaining instructions and the embedded checksum in the cache line are neglected since there is a jump to another address in the code. This state jump checks whether there branch signal is still equal to 1.

According to the wave diagram in Figure 4.13, we can observe the following:

- In the case of jump the signal *branch* is asserted whenever a jump instruction occurs. The *branch* is detected whenever the *rbranch* signal or the *fbranch* signal is asserted.

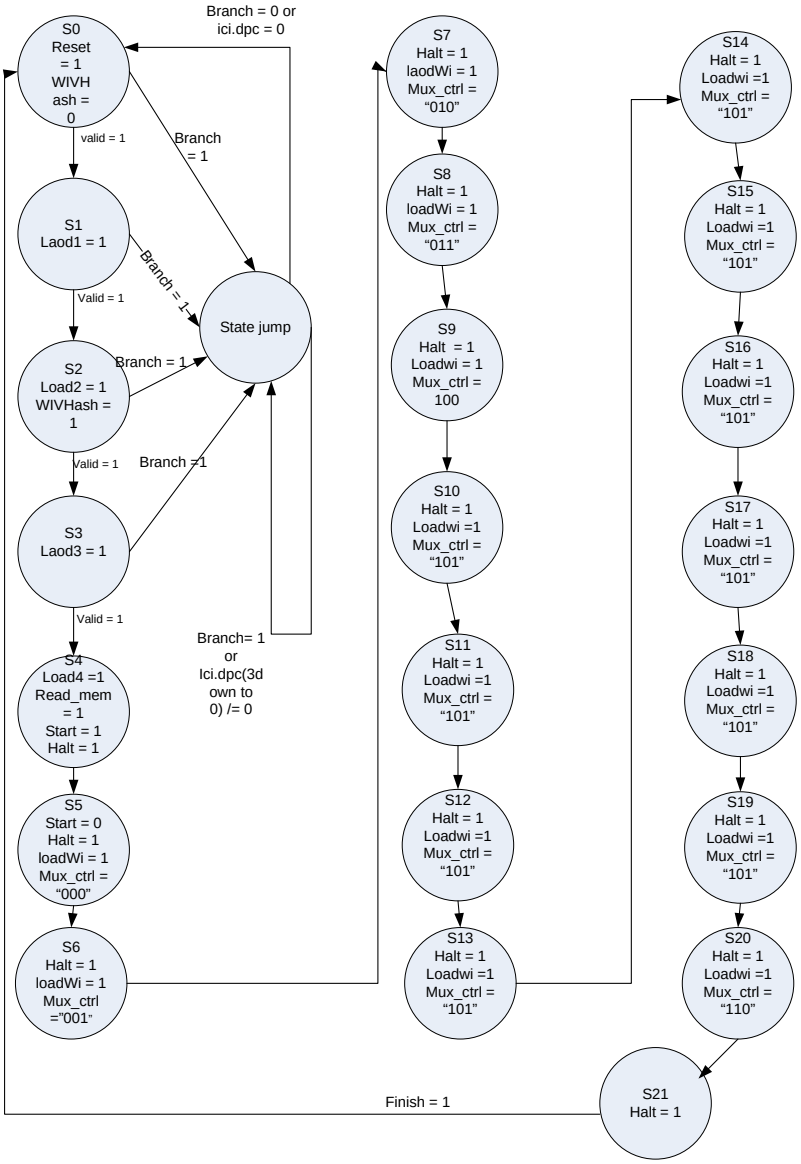

Figure 4.12: Cache to Hash Unit

- The current state is hanging in the state jump and the next address is stored in *ici.fpc*.

4.2.6 Halting the Pipeline Unit

Let us show how the pipeline unit looks like when the standard execution of the Leon3 processor is considered and the execution when the cryptography extension is added. The pipeline unit has been modified where the halt signal is assigned to the hold pc signal of the Leon processor. The *hold pc* signal controls the fpc (fetch program counter) register

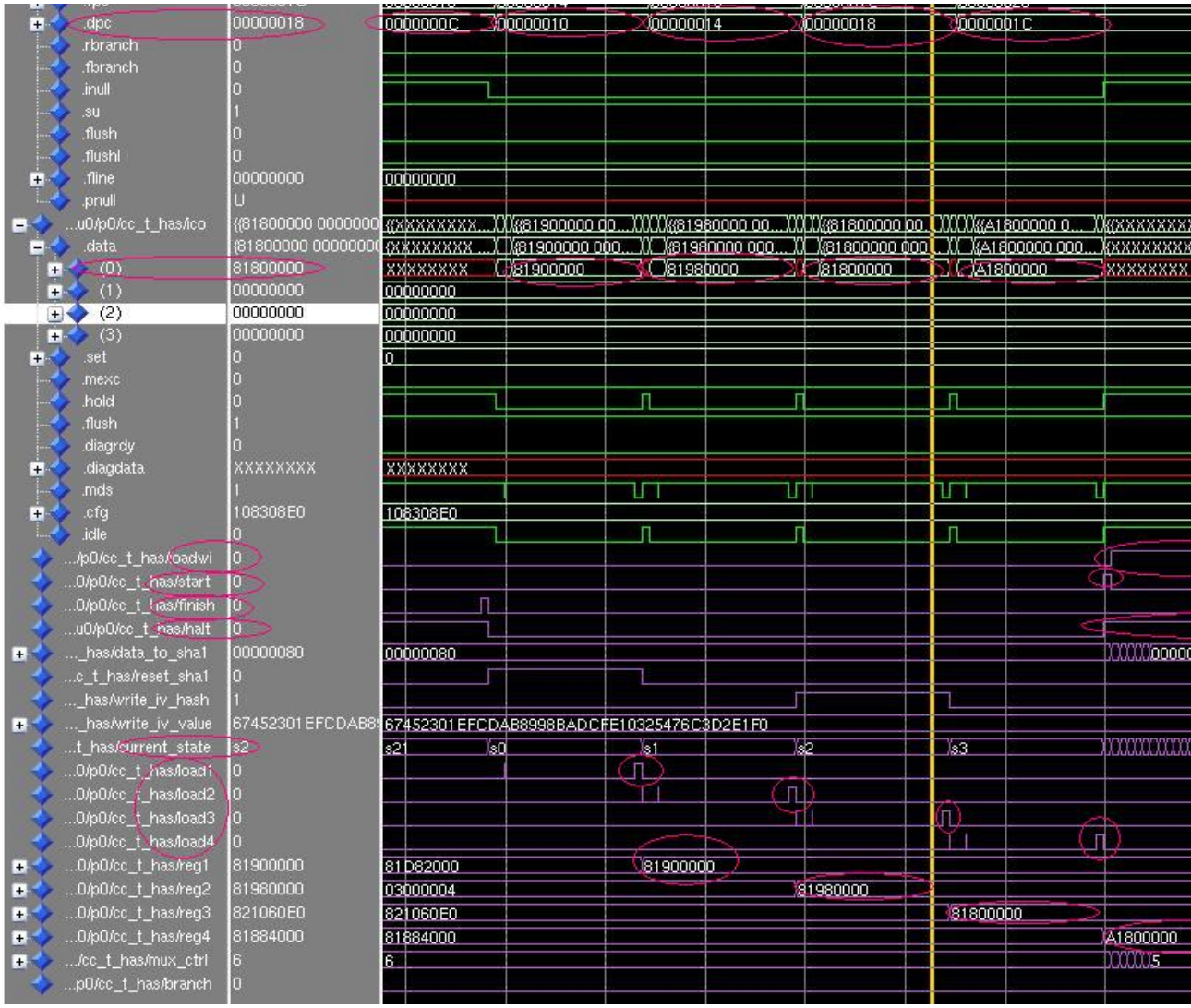

Figure 4.13: wave diagram of Cache To Hash(no jump instructions)

in the pipleine unit of the Leon processor, if *hold pc* is enabled then the current program counter is held in this register and the pc is not increased. The data are not latched in the pipeline stages but nops are included. The *halt* signal controls a multiplexor to choose between the normal instruction data or the nops. When *hold pc* is disabled or when *halt* signal is disabled, this means that the cryptographic unit is done with the computation of the checksum, the program counter is increased and the new instruction is fetched. Figure 4.14 shows the normal execution of the pipeline unit and Figure 4.15 illustrates the pipeline unit when cryptography occurs.

Figure 4.14 illustrates the standard 5 stages pipeline execution from Fetch, Decode, Execute, Memory and Write Back. In the Fetch stage the next instruction to be executed is fetched from memory. In the decode stage, the instruction is decoded and the operands

Figure 4.14: wave diagram of Cache To Hash (with jump instructions)

are read. All arithmetic operations are performed in the Arithmetic Logical Unit (ALU). The data are valid in the memory stage. In the write back stage all results from the ALU are written back to the register file. Figure 4.15 describes the modification of the pipeline unit whenever the halt signal is enabled, the fetch program counter register will hold the current value of pc and nops are included to the pipeline unit so that the unit does not execute anything.

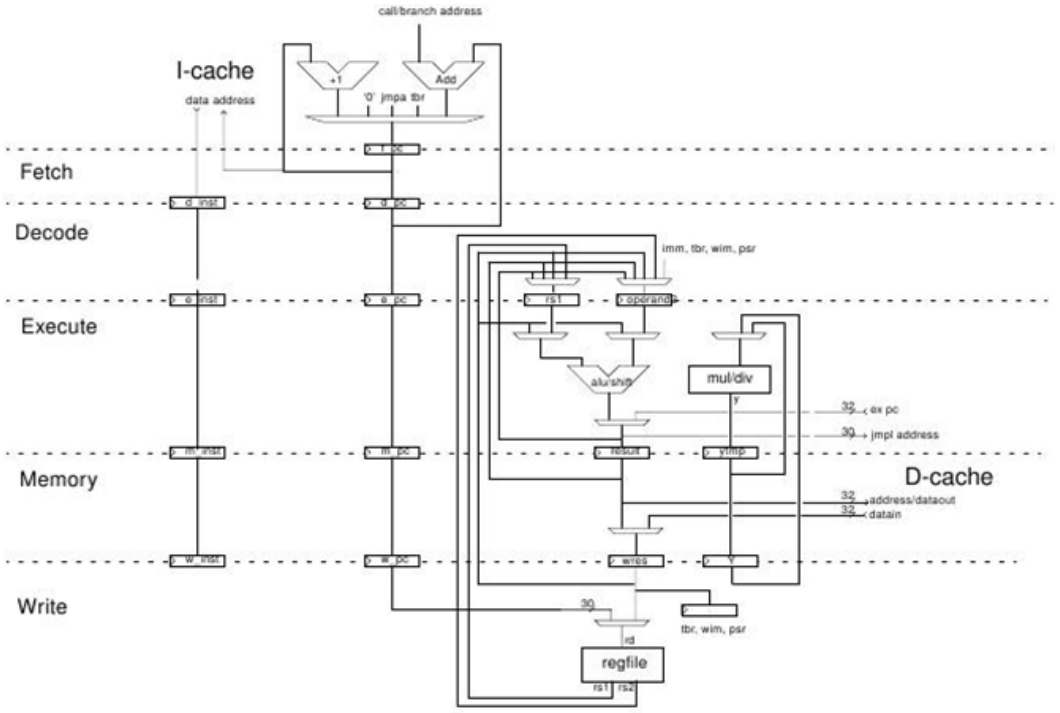

Figure 4.15: Normal Execution of the pipleine unit[8]

4.3 Conclusion

In chapter 4, The certified authority which is a software program reads the binary file of a compiled program, assigns the instructions into a certain number of bytes, calculate the checksums with the SHA-1 algorithm and embed the checksums in the binary file and the binary file is loaded in the the PROM. The implementation of the Code Integrity Check (CIC) unit in hardware using the Leon3 Template is described. The hardware CIC (Code Integrity Check)unit that was build out of the controller of the cryptographic unit, the cryptographic unit (SHA-1 core), the generate valid units to detect whenever the data read from the cache are valid, Block RAM to store the pre-computed checksums and a compare unit were illustrated. The simulation results of the behavior of the Code Integrity Check unit, the controller of the cryptography unit were discussed. Finally, the modification of the pipeline unit was discussed.

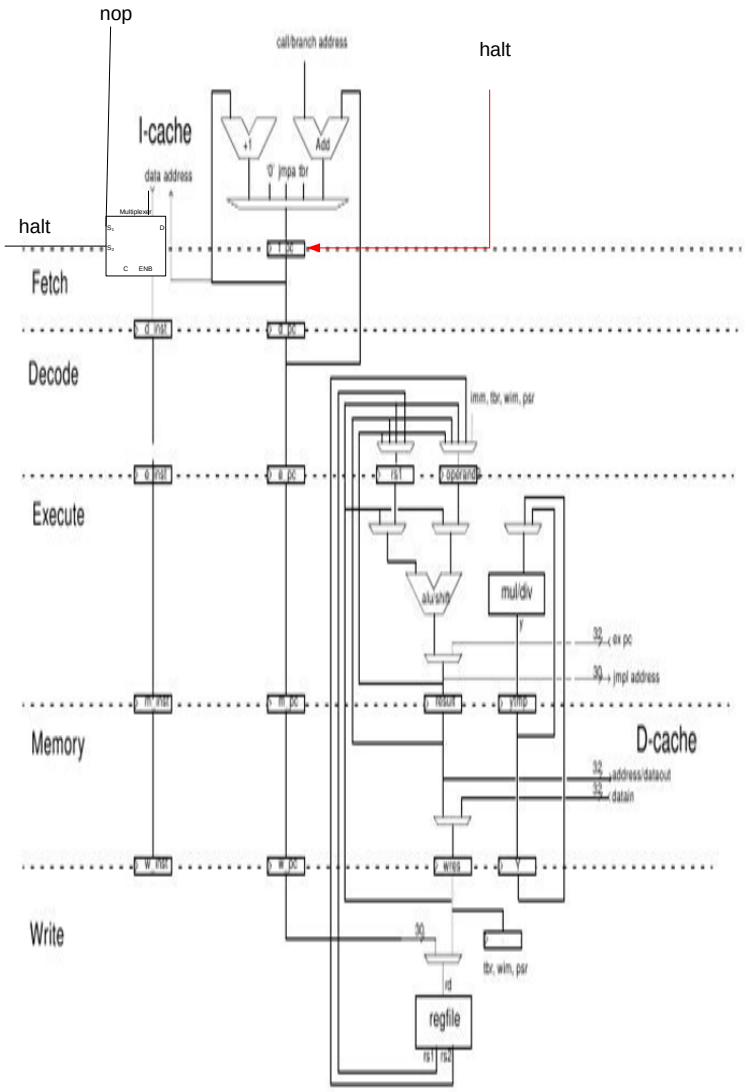

Figure 4.16: Execution of the pipleine unit when cryptography occurs

5 Evaluation

Section 5.1 discusses the comparison of the three different secure processors that were discussed in Section 2.3 Related work. Section 5.2 shows some experimental results and takes into consideration the area and delay overhead when the design is scaled.

5.1 Design Comparisons

The code integrity check controls whether the binary code has been modified by pre-computing checksums prior the execution of the program. During execution of the program, the checksum is computed and compared to the pre-computed one (assuming the operating system is trusted). The Code Integrity Check Unit provides only the integrity of the binary code but does not provide memory integrity verification. If memory integrity verification is required, then a combination of our design and eXecute Only Memory (XOM)[41] can be used. Some of the available secure processors are already discussed in Section 2.2.

We have discussed three different designs the XOM architecture, the Secure Coprocessor of IBM4758 and the Trusted Computing Group. As we mentioned in the beginning of this section that the Code Integrity Check Unit provides only the integrity of the binary code but does not provide memory integrity verification. If memory integrity verification is required, then a combination of our design and eXecute Only Memory (XOM)[41] can be used. The common characteristics of the discussed architecutres is that the hardware is assumed to be correct. Architectures like XOM provide a software tamper-resistant execution environment. The check is done inside the processor so no separate hardware is needed while the Trusted Computing Module of TCG group provides the same thing as XOM except that the check is done outside the processor, so a separate hardware is required. On the other hand, the IBM4758 secure Co-processor protects secret keys for security applications where the secret keys that are stored are never leaked. Table 5.1 explains the comparisons between our design and the XOM processor, the Secure Coprocessor of IBM4758 and the Trusted Computing Group(e.g Intel Lagrande).

In table 5.1 the comparisons between different designs is made in terms of whether the design requires separate hardware to implement the cryptography, the measure of the trust of the user, this means that why a user should trust the system, security perimeter is taken into account, this means that where the security has been concentrated, is it inside the processor and not outside or is it in both, e.g in our design/XOM the security has been concentrated only inside the processor chip, where the memory is untrusted.

Table 5.1: Design Comparisons

	Our Design /XOM	Secure Co-processors, e.g. IBM4758	Trusted Computing Group , e.g. Intel LaGrande
Goal	Copy and tamper-resistant software distribution and execution (inside processor)	Secret protection for Security applications	Copy and tamper-resistant software distribution and execution (outside processor)
Separate Hardware	No	Yes	Yes
Measure of User's trust	Integrity of Software and its computation result is checked	The Secrets that are stored are never leaked	Integrity of Software is checked at load time only
Security perimeter	Processor Chip Boundary	The private keys are securely stored inside the casing during manufacture	Processor, DRAM, TPM chip and busses
Common Characteristics	Hardware is correct , Permanent device secret(Public-private key)	Hardware is correct , Permanent device secret(Public-private key)	Hardware is correct, Permanent device secret(Public-private key)

5.2 Experimental Results

This section describes some of the test results that we have obtained from synthesizing our design. The Xilinx ISE tool is used to synthesize the design in hardware. The default values of the LEON-3 architectural parameters that we used and the latency of the SHA-1 core are depicted in table 5.2.

Table 5.2: Architectural Parameters

Architectural Parameter	Value
L1- Instruction Cache	64 KB/set, 1-set, 32 Bytes/line
L1-latency	1 cycle
Load cycles	2 cycles
Store Cycles	2 cycles
SHA-1 Latency	80 cycles

First, we synthesize the original Leon3MP design without modifications and then we synthesize the Leon3MP after modification with adding the cryptographic unit. We check the overhead in delay by taking into account the associativity (number of sets) and the set size (Kbytes/set) of the Leon3 processor cache. We measure the delay by increasing the set size for different number of sets between one to four. We do that to check what the delay is when the design is scaled. The delay is considered to be inverse proportional to the frequency of the design. We start first with the measurements of the delay in ns without any modification. From the synthesis results, we discovered that the delay stays constant when the set size of the cache increases. The delay varies only when the number of sets varies between one to four. The results are illustrated in table 5.3.

Table 5.3: Delay measurements without modifications

number of sets	1	2	3	4
Delay[ns]	8.127 ns	8.303 ns	8.305 ns	8.309 ns

The area usage is about 24 percent usage of the LUT(Look up table on the FPGA). On the other hand we measure the delay after modifying the Leon3 processor by adding the cryptographic unit. From the synthesis results, we noticed that the delay does not vary if the size of the cache increases. The delay increases with the variation of the number of sets. A possible explanation why the delay stays constants is that we are loading the binary code in the memory where the size of the code is staying constant. The results are illustrated in table 5.4. The area usage is 40 percent. The area overhead almost doubles.

Table 5.4: Delay measurements with modifications

number of sets	1	2	3	4
Delay[ns]	8.393 ns	8.392 ns	8.392 ns	8.392 ns

If we compare these results with the results of table 5.3, we conclude that the delay increases. This is due to the fact that the computation time of the cryptographic unit is high. The overhead in delay is presented in table 5.5 for different number of sets and set size. The graph in Figure 5.1 illustrates the delay overhead where the x axis represents the size of the cache in Kbytes/sets and the y axis the delay overhead in percentage for set associativity 1, 2, 3, 4.

Table 5.5: Delay overhead

number of sets	1	2	3	4
Delay[ns]	26.6 %	28.9 %	38.7 %	48.3 %

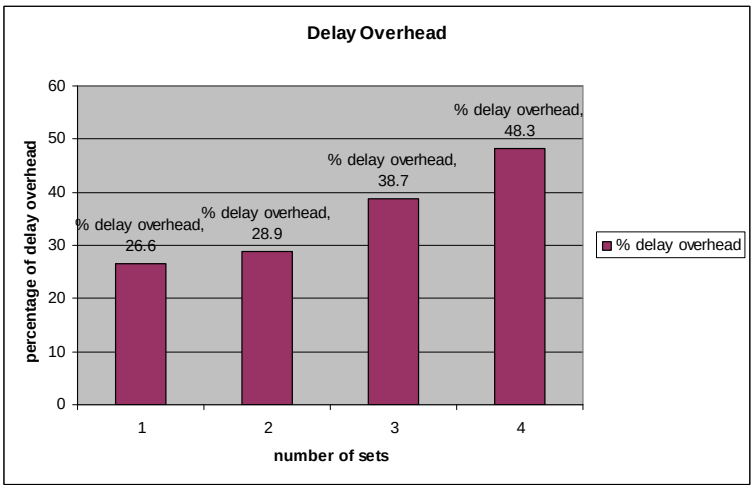

Figure 5.1: Overhead in Delay vs size of the cache

Let us discuss the reason behind the increase in delay overhead if the set associativity of the cache varies between 1 to 4. Generally speaking, in a direct mapped cache a memory block maps to exactly one cache block. At the other extreme, in a fully associative

cache a memory block can be mapped to any cache block. On the other hand to reduce cache miss rate a compromise is to divide the cache into sets each of which consists of n "ways" (n-way set associative). A memory block maps to a unique set specified by the index field and can be placed in any way of that set. The disadvantage of the n-way set associative cache is that it requires n comparators while 1 comparator in the direct map is needed. This will increase the delay overhead and area. The n-way set associative requires additional multiplexor delay for the data than the direct mapping scheme. In n-way set associative the data is available after set selection and Hit/Miss decision, while in a direct mapped cache, the cache block is available before the Hit/Miss decision.

Other reasons that lead to performance degradation is the delay caused by the Code Integrity Check unit implemented in hardware. The delay is caused at the cache-memory boundary upon the L1 cache miss, where the miss penalty is already several hundred cycles for typical microprocessors. Hence, some tens of extra cycles for hardware hash computation is going to cause much of performance degradation. The code integrity checking inserts additional no-op instructions into the instruction stream for every cache line. These will cause degradation in the efficiency of the instruction fetch since a fraction of instructions fetched are useless.

On the other hand, to reduce the performance overhead, the cryptographic unit has to be pipelined. Instead of waiting 80 clock cycles to compute the hash value for an input message, we can have 80 pipelined stages so that the throughput will be 160 bits per clock cycle. Another method to reduce performance overhead, is to modify the compiler, such that prior to the execution of the program, the hash values are loaded in a bursting fashion in the on-chip memory, this means that they are not loaded one hash value at a time.

5.3 Conclusion

This chapter discusses the comparison between the available secure processors which are XOM architecture, the Secure Coprocessor of IBM4758 and the Trusted Computing Module of TCG group. The experiment results show that when the design is scaled the area will double. On the other hand the overhead in delay increases if the set associative increases, this is because more logic is needed to calculate where the block of data should be in the set. In case one set Direct Mapping there is only one set. In case of 2 ways you need to check whether the data is left or right. In case of 4 ways you have to check where the data will be between one of the four.

6 Conclusions

6.1 Summary

In this thesis, a design of the code integrity check unit is implemented in hardware. The optimized cryptographic core SHA-1 unit is used to calculate the hash values at run time. The pre-computed values are computed statically before the program runs on the Leon processor. These values are usually generated by an authority software that reads the binary code, calculates the checksums for every cache line and embed the pre-calculated checksums in the original binary code. When the program starts executing the checksum values are generated in hardware and checked with the pre-calculated one. If they are not similar an interrupt is raised and the pipeline of the processor is stalled.

In chapter 1, the problem statement has been discussed is that most of the new security attacks result in violating the integrity of the software code of an application program. Such security attacks or threats try to change the instructions so that adversary try to gain control over the program execution flow. An example of a malicious code has been given.

In chapter 2, some security issues were discussed. Different classes of attacks into computer systems were described like the Hardware-Based Attacks where an adversary has direct physical hardware access, the Software-based Attacks that are specially used remotely to break into computer and embedded systems. Some of the common software based attacks that were explained are Buffer Overflow, Heap Overflow, Double free vulnerability, Format string attack and Temporary file vulnerability. Some related work has been discussed like a design of an architecture for a secure execution of programs on embedded processors. A hash function which is a computationally efficient function converting binary strings of arbitrary length to binary strings of fixed length called the checksums was discussed. The hash functions are one-way functions that computes a small fixed length output value, the digest message, that is highly correlated with the input data. One of the trivial characteristics of the hash functions is that no information of the data input can be obtained. It is very rarely to have two different data streams generating the same hash value. The Secure Hash Algorithm (SHA-1) computes a 160 bit message digest or output hash value from the input message. The input data stream is separated into multiple input bocks of 512 bits each. The input block is split into 80*32 bits words, one 32-bit word for each computational round of the SHA-1 algorithm. Each round comprises additions and logical operations, such as bitwise logical operations and bitwise rotations to left. The computation of the final checksum is an eighty iteration algorithm over each message block, where every message block is a sequence of 32 bit words.

In chapter 3, the choices for the designs were described. A choice has to be made which of the three soft-core is going to be used in this thesis. The three processors

LEON-3 and OpenRISC 1200 and Microblaze are 32-bit RISC processor big endian synthesizable pipelined processors. LEON-3 and OpenRISC 1200 contains 5-stage pipelines while the Microblaze contains 3 stages of pipeline. Leon3 soft-core processor is the targeted platform chosen to modify because Leon-3 is the most configurable processor, while MicroBlaze and OpenRISC have less configuration options. Leon soft-core offers better design flexibility than the MicroBlaze and OpenRISC. The 32-bit LEON processor that complies to the SPARC V8 architecture was discussed. The Leon Integer Unit as defined that consists of five pipeline stages, the on-chip memory in LEON3 is implemented with separate instruction and data buses, and how to configure the cache direct map or set associative and the size of the cache were also illustrated. The memory controller of the Leon-3 processor controls a memory bus holding external memory devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM) were also explained. AMBA (Advanced Microcontroller Bus Architecture) is an on-chip bus specification for interconnection and organization of various functional modules that are a part of System-on-Chip. The AMBA specification that improves the reusable system on-chip platform by including a common standard for data communication in a System-on-Chip module and the three distinct AMBA buses which are Advanced High-performance Bus (AHB), Advanced System Bus (ASB) and Advanced Peripheral Bus (APB).

In chapter 4, The certified authority which is a software program reads the binary file of a compiled program, assigns the instructions into a certain number of bytes, calculate the checksums with the SHA-1 algorithm and embed the checksums to the cache line that contains the instructions and load them in the PROM. The efficient implementation of Secure Hash Algorithm (SHA-1) in hardware was described. This section describes the implementation of the Code Integrity Check (CIC) unit in hardware using the Leon3 ML403 Template. The hardware CIC (Code Integrity Check)unit that was build out of some key components called the controller of the cryptographic unit, the cryptographic unit (SHA-1 core), separate control units to detect whenever the data read from the cache are valid, some selection logic components and a compare unit were illustrated.

In chapter 5, the comparison between the available secure processors which are XOM architecture, the Secure Coprocessor of IBM4758 and the Trusted Computing Module of TCG group has been discussed. The delay is measured by configuring the size of the cache for different number of sets. The experiment results show that when the design is scaled the area will double. The cryptography unit when it is added to the Leon3 template the delay in overhead increases about 30 percent and the area doubles in size. The reason why the overhead in delay increases if the set associative increases, because more logic is needed to calculate where the block of data should be in the set. In case one set Direct Mapping there is only one set. In case of 2 ways you need to check whether the data is left or right. In case of 4 ways you have to check where the data will be between one of the four.

6.2 Future Work

The following points proposes the future research direction work:

- Implementing the Concealed Execution Mode unit that checks for integrity of data

during runtime. When the data are written to the memory. Before sending the data to the off-chip, the data has to be encrypted and hashed. When the data are read on chip, the pre-calculated checksums are checked with the one calculated during run-time(same as instruction hash).

- Modifying the compiler of the LEON processor to add instructions like hash pointer to the Instruction Set Architecture (ISA). Every basic block begins with the hash pointer where it points to the pre-computed checksum value of the basic block. During execution of the code, the checksum is calculated on run-time and compared to the pre-computed value which is found easily by the hash pointer.

Bibliography

[1] J.S. Dwoskin, R.B. Lee. "Hardware-rooted trust for secure key management and transient trust," In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 389 - 400, IEEE November. 2007.

[2] R.B. Lee, P.C. Kwan, J.P. McGregor, J. Dwoskin, Z. Wang. "Architecture for protecting critical secrets in microprocessors," In *Proceedings. 32nd International Symposiumon*, pp. 2- 13, IEEE June. 2005.

[3] J.P. McGregor, R.B. Lee. "Protecting cryptographic keys and computations via virtual secure coprocessing", pp. 16 - 26, IEEE March. 2005.

[4] R. Chaves, "Secure Computing on Reconfigurable Systems", pp. 207, December 2007, PhD Thesis.

[5] R.D. Stinson, "Cryptography-Theory and practice", CRC Press, 1995.

[6] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, "Handbook of Applied Cryptography", CRC Press, 2001.

[7] www.gaisler.com

[8] J. Gaisler, "The Leon Processor User's Manual", Version 2.3.7, August 2001

[9] J. Gaisler, "BCC-Bare-C Cross-Compiler User's Manual", Version 1.0.29, February 2007

[10] www.exforsys.com/tutorials/c-language

[11] S.G.Kochan, "Programming in ANSI C"

[12] B.W. Kernighan, D.M. Ritchie, "The C Programming Language"

[13] Federal Information Processing Standards Publication, "Specifications for the Secure Hash Standard",August 2002.

[14] Federal Information Processing Standards Publication, "Specifications for the Keyed-Hash Message Authentication Code",March 2002.

[15] S. Pongyupinpanich, S. Choomchuay. "An Architecture for a SHA-1 Applied for DSA", 3rd Asian International Mobile Computing Conference May. 2004.

[16] M. Kim, Y. Kim, J. Ryou, S. Jun. "Efficient Implemntation of the keyed-Hash Message Authentication Code Based on SHA-1 Algorithm for Mobile Trusted Computing", pp. 410-419, IEEE August. 2007.

[17] D. Zibin, Z. Ning. "FPGA Implementation of SHA-1 Algorithm", IEEE August. 2003.

[18] D. Toma, A. Perez, D. Borrione, E. Bergeret. "Design Of A Proven Correct SHA Circuit", IEEE March. 2003.

[19] J. Dwoskin, D. Xu, J. Huang, M. Chiang , R. Lee. "Secure Key Managment Architecture Against Sensor-node Fabrication attacks", IEEE Dec. 2007

[20] H. Lin, X. Guan, Y .Fei, Z.J. Shi. "Compiler-assisted Architectural Support for Program Code Integrity Monitoring in Application-specific Instruction Set Processors", pp 815 - 820,IEEE 2007.

[21] Y .Fei, Z.J. Shi. "Microarchitectural Support for Program Code Integrity Monitoring in Application-specific Instruction Set Processors",IEEE 2007.

[22] D. Arora, S. Ravi, A. Raghunathan, N. K. Jha. "Secure Embedded Processing through Hardware-assisted Run-time Monitoring", pp 178 - 183, IEEE 2005.

[23] D. Arora, S. Ravi, A. Raghunathan, N. K. Jha. "Enhancing security through hardware-assisted run-time validation of program data properties",pp 190 - 195, IEEE 2005.

[24] D. Arora, S. Ravi, A. Raghunathan, N. K. Jha. "Architectural Enhacements for Secure Embedded Processing",IEEE 2005.

[25] K.D.Wilken, T. Kong. "Concurrent Detection of Software and Hardware Data-Access Faults", IEEE 1997.

[26] R.G.Gael. "Architectural Support for Security and Reliability in Embedded Processors", August 2006, PhD Thesis.

[27] J.Platte. "A Security Architecture for Microprocessors", Nov. 2006, PHD Thesis.

[28] J.Gaisler. "GRLIB IP Core User's Manual", Version 1.0.19, September 2008.

[29] J. Gasiler. "Leon 3 ML401 Template Design", November 2007

[30] L. Buttelmann. "Leon 3 VHDL simulation guide", Version 0.1.18.10.2007

[31] "The SPARC Architectural Manual", Version 8.

[32] P. Anemaet, T. Van As. "Microprocessors Soft-Cores: An Evaluation of Design Methods and Concepts on FPGAs", part of the Computer Architecture(Special Topics) course ET4078, Department of Computer Engineering.

[33] F. Duarte, "A Cache-Based Hardware Accelrator for Memory Data Movements", Nov.2008, PhD Thesis.

[34] A. Murat Fiskiran, R. B. Lee, "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution", pp 452 - 457 , IEEE 2004.

[35] D.Driessens, T. Tierens, "Embedded Systeemontwerp op basis van soft end Hardcore FPGA's".

[36] D. Mattsson, M.Christensson, "Evaluation of Synthesizable CPU Cores ".

[37] R.R. Srivastava, "System on-chip platform", Master Thesis, August 2004.

[38] "AMBA Specification", ARM, May 1999

[39] R. Lien, T. Grembowski, and K. Gaj, A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512, in CT-RSA, pp. 324338, 2004.

[40] K.Eisele, "Design of a Memory Management Unit for System-on-a-chip Platform LEON", Master thesis, University of Stuttgart.

[41] D.L.C. Thekkath, M.M.P. Lincoln, D.B.J. Mitchell, M.Horowitz, "Architectural Support for Copy and Tamper Resistant Software". pp 168 - 177 , IEEE 2000.

[42] S.W. Smith, S. Weingart, "Building a High-Performance, Programmable Secure CoProcessor". IEEE 1998.

[43] Trusted Computed Group.LaGrande Technology Architectural Overview.

[44] TCG Specification Architecture Overview, August 2007.

[45] S. Bajikar, "Trusted Platform Module (TPM) based security on Notebook PCs. June 2002.

Appendix A: LEON3 VHDL simulation steps with Modelsim

7

This appendix explains the steps of how to run an application on LEON3 processor. Most of the tutorials do not explain clearly the methods and steps that need to be taken in order to compile, simulate the design. We start first explaining how to configure the Leon3 processor with make xconfig GUI.

- make xconfig : changes the leon3 configuration by creating a config.vhd

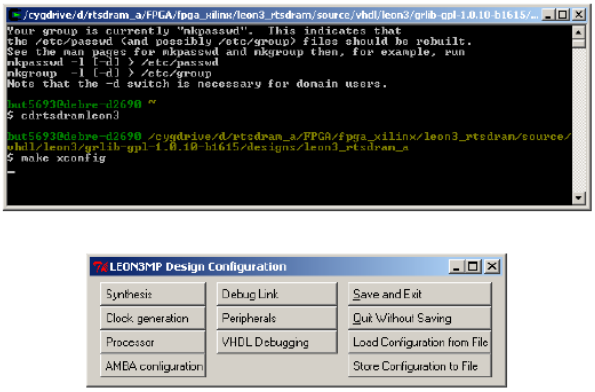

Figure 7.1: Leon processor Configuration

- Processor : by clicking on processor , a GUI filled with several options is launched. You have a flexibility of modifying the cores inside the Processor.

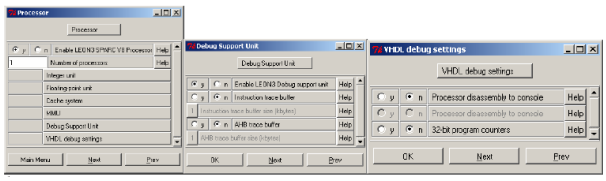

Figure 7.2: Configuration Inside processor

Next we describe the steps to compile an application. We give a very simple example of how to do it.

- gcc hello.c :This step compiles the C code and creates and executable file. The command used is : sparc-elf-gcc -msoft-float-O2 hello.c -o hello.exe
- mkprom : This step loads the binary file into the prom of the Leon processor. The command used is : sparc-elf-mkprom.exe -rmw -msoft-float -v -ramsize 1024 hello.exe
- objcopy : this step creates a copy of the object file. The command used is : sparc-elf-objcopy -O srec prom.out prom.srec

After the steps of compilation of the applications are fulfilled, we describe the steps used for simulation and synthesize with XILINX ISE.

- make vsim : This step compiles all the IP cores of the LEON3 template.
- vsim testbench : This step loads the testbench in modelsim and runs the simulation.
- make scripts : This step creates a compile.xst file which contains commands for analyzing all GRLIB files and creates also .npl files for the ISE project.
- ise leon3mp.ise : This step creates the .ise project file.
- make ise : This step generates the netlist with XST. The final programming file is the "LEON3mp.bit" is the file that can be used to be run on the FPGA board.

Appendix B: Software Test program

8

This appendix explains the software programs that run on LEON3 processor.

- Simple test program, testing on for basic block.

```
main()

{

        int result1,result2;
        int x = 2;
        int y = 3;
        result1 =  x + y;

}
```

Figure 8.1: Simple Test program

- Simple Test program but including some standard function like printf and basic test function for the LEON processor but the program is still running from main.

```
main()

{
        report_start();

        base_test();
        int result1,result2;
        int x = 2;
        int y = 3;
        result1 = sum(x,y);
        result2 = mult(x,y);
        printf("the sum is %d \n",result1);
        printf("the product is %d \n",result2);
        report_end();
}
```

Figure 8.2: Test program 2

- Test program 3 including jump instructions and many if statements
- Test progam 4 including while loop and for loops

```
int sum (int a , int b)
{
    int result;
    result = a + b;
    return result;
}

int mult (int a , int b)
{
    int result;
    if (a > b)
        result = b * a;
    else
        result = a * b;

    return result;
}
main()

{
    report_start();

    base_test();
    int result1,result2;
    int x = 2;
    int y = 3;
    result1 = sum(x,y);
    result2 = mult(x,y);
    printf("the sum is %d \n",result1);
    printf("the product is %d \n",result2);
    report_end();
}
```

Figure 8.3: Test program 3

```
Void sum (int a , int b)
{
        int *result,
        result = malloc(sizeof(int)*1000);
        int i;
        for(i= 0; i <1000;i++)
                result[i] = a + b;

}

int mult (int a , int b)
{
        int result;
        while (a > b)
                result = b * a;
        else
                result = a * b;

        return result;
}
main()

{
        report_start();

        base_test();
        int result1,result2;
        int x = 2;
        int y = 3;
        result1 = sum(x,y);
        result2 = mult(x,y);
        printf("the sum is %d \n",result1);
        printf("the product is %d \n",result2);
        report_end();
}
```

Figure 8.4: Test program 4