FPGA-Based Testbed for Fault Injection on SHA-256

by

Najma Jose

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Author’s Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In real world applications, cryptographic algorithms are implemented in hardware or software on specific devices. An active attacker may inject faults during the computation process and careful analysis of faulty results can potentially leak secret information. These kinds of attacks known as fault injection attacks may have devastating effects in the field of hardware and embedded cryptography. This research proposes a partial implementation of SHA-256 along with an onboard fault injection circuit implemented on an FPGA. The proposed fault injection circuit is used to generate glitches in the clock to induce a setup time violation in the circuit and thereby produce error(s) in the output. The main objective of this research is to study the viability of fault injection using the clock glitches on the SHA-256.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Catherine Gebotys at the Electrical and Computer Engineering Department, University of Waterloo for her continuous guidance and support. This thesis would not have been possible without her valuable advices and motivation. For this I shall forever be indebted to her.

I thank my fellow colleagues from Professor Gebotys's research group: Dr Edgar Mateos Santillan, Dr Marcio Juliato, Masoumeh Dadjou and Mustafa Faraj for their friendship and encouragement. I would especially like to thank Edgar for his help during the hardware debugging part of this thesis.

Many thanks to Professor Anwar Hasan and Professor Guang Gong for their valuable feedback and comments on this thesis work.

I gratefully acknowledge the Department of Electrical and Computer Engineering, University of Waterloo for providing me with the opportunity and environment for graduate studies.

I would like to thank my dear friends especially Meethu, Suhas, Jithin and Akhil for their love and support during the past few years.

Finally and most importantly I thank my family – my parents Jose Antony and Subeda Sivan who laid the foundation that I stand on, my younger brother Sachin Jose, my husband Jerald and my in-laws for their love, care and support.

This thesis is dedicated to my parents, Jose & Subeda for their unconditional support and love.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CLB	Combinational Logic Block
DFA	Differential Fault Analysis
DPA	Differential Power Analysis
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Logic Array
HMAC	Hash based Message Authentication Code
IKE	Internet Key Exchange
LE	Logic Element
LUT	Look Up Table
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
PLL	Phase Locked Loop
SCA	Side Channel Attack
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SOPC	System on Programmable Chip
SSL	Secure Socket Layer

Chapter 1

Introduction

The emerging demand for secure transmission of confidential information has resulted in the increased used of cryptographic algorithms in a wide range of applications such as smart cards, communication systems, electronic commerce etc. Cryptographic algorithms and protocols can be grouped into Symmetric and Asymmetric Encryption, Data Integrity Algorithms and Authentication Protocols. They are often used to provide confidentiality, authentication and non-repudiation in different applications.

Confidentiality: maintains restrictions on access to information thereby protecting it from unauthorized people or software.

Authentication: may be one of two types namely data origin authentication or user authentication. Data origin authentication verifies that the source of data is original as claimed and user authentication verifies that the person is who she/he claims to be.

Integrity: refers to protecting information from modification and destruction by unauthorized people.

The cryptographic algorithms are designed in such a way that it is difficult to break them mathematically [1]. One can view the idea of cryptography from two different standpoints namely ***classical and physical***.

In the case of classical view the cryptographic primitive can be treated as a mathematical object that transforms an input into an output using some key. This idea is represented in Figure 1.1. In order to obtain the secret information the attacker needs to perform a brute force analysis or search the entire key space which in most of the cases is a computationally infeasible task. The

attackers might have access to the input and output to the system as well as knowledge of the algorithm. They try to get the secret key by solving the problem mathematically rather than using or tampering with the system. The chosen plain text attack, chosen cipher text attack etc. are examples of such kind of attacks.

Figure 1.1 Classical view of cryptography

But this is not the case with physical attacks where attackers may take advantage of the implementation characteristics to retrieve secret information. These physical attacks are much more powerful as compared to the previously mentioned classical cryptanalysis techniques and are a serious threat to cryptosystems. The following section discusses a kind of physical attack known as Side Channel Attack, its basic principle and various types.

1.1 Side Channel Attacks

Cryptographic algorithms are implemented in hardware or software on specific devices for real world applications. The behavior of these devices can be closely monitored and studied in such a way where information regarding the task being performed is leaked. This is the basic principle behind side channel analysis. The electronic devices performing encryption/decryption operation are inherently leaky and will produce side channel such as power consumption, timing information, electromagnetic leaks etc. This side channel view of a cryptographic device is presented in Figure 1.2. Side channel attacks based on timing information, power consumption, and electromagnetic measurements have been a major security concern in the industry for more than 15 years [2].

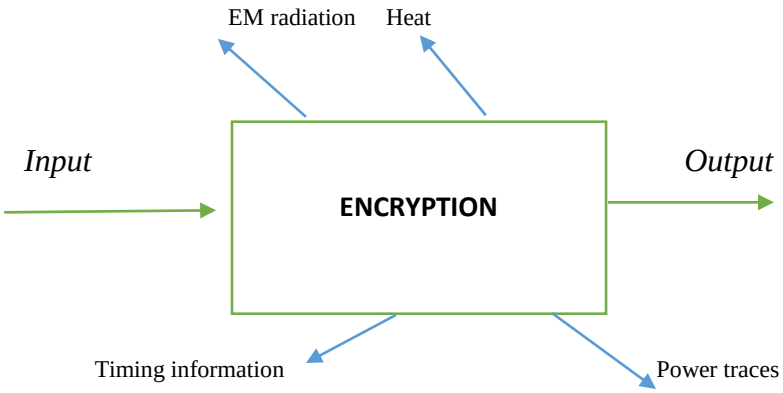


Figure 1.2 Side channel view of cryptography

These attacks usually have two phases: interaction phase and exploitation phase [3]. In the interaction phase the attacker tries to gather side channel information and in the exploitation phase the attacker performs careful analysis to reveal the secret information. Even though the implementation of these kind of attacks are highly platform dependent, they are a serious threat to the security systems. The following section gives a brief overview on the classification of side channel analysis based on various criteria.

1.1.1 Types of Side Channel Attacks

The side channel attacks (SCA) can be broadly classified into three categories. A specific attack need not necessarily belong to only one specific category [3] and the categories are orthogonal.

Based on access to the device: Invasive, Non-invasive and Semi-invasive

The ***invasive attacks*** require the attacker to have full access to the device in order to extract information from the chip and study its functionality. These attacks are expensive and require sophisticated environment to carryout the operation. On the other hand the ***non-invasive attacks*** are often low cost attacks and do not require any physical contact with the device. The non-invasive attacks make use of the available side channel information (such as emissions). In the

case of ***semi-invasive attacks*** the depackaging of a device is required to access its surface but the internal structure remains intact.

Based on process control: Active and Passive

In the case of passive attacks the system process/resources remain unaffected and the attacker gains information by observing the behavior of the device. This can be performed through eavesdropping or traffic analysis (in the case of computer networks) or side channel measurements such as power consumption, EM emissions, timing information etc. The passive attacks are often hard to detect since they involve no altering of the computational process or data whatsoever. Active attacks involve an attacker tampering with the device's operation causing it to produce erroneous results that can be further exploited.

Based on methods used in analysis: Simple and Differential

The obtained side channel information can be analyzed using different methods. The ***simple side channel analysis (SSCA)*** typically involves a single trace and the secret information is directly read from it. An example for this can be a power trace that shows a peak value corresponding to *1* and low value corresponding to *0* where the side channel is related to the instruction being executed. In the case of ***differential side channel analysis (DSCA),*** statistical methods are used to analyze the correlation between the data and the side channel information of a particular device.

The first SCA using timing information was introduced by Kocher [4] who demonstrated the attack on the RSA modular exponentiation. The differential power analysis (DPA) attacks were demonstrated in 1999 on a hardware implementation of DES [5].

1.1.2 Fault Attacks

Fault attacks are a different type of side channel attack where the attacker induces faults into the device during the cryptographic computation. The attacker may be able to extract secret information from analysis of the behavior of the cryptographic computations in the presence of the injected fault. These types of attacks are active where the attacker alters the computation

process and tries to recover secret information such as the key. For example, in 2000 Boneh et al. reported the first successful fault attack [6] on the RSA signature scheme. A detailed overview of fault attacks including various methods is given in the next chapter.

1.2 Authentication and HMAC/SHA-256

In applications such as banking, the integrity of encrypted data is a major concern. The authenticated encryption schemes based on hash functions can provide confidentiality and authentication of confidentiality. A cryptographic hash function accepts an arbitrary length input and produces an output of fixed length. It has the following properties:

One way property (pre-image resistance): given the hash (*h),* it is computationally infeasible to find a message *x* such that *hash(x) =h.*

Weak collision resistance (second pre-image resistance): given a block *x,* it is computationally infeasible to find *y* such that *hash(x) =hash(y).*

Strong collision resistance: it is computationally infeasible to find *(x, y), x* not equal to *y* such that *hash(x) =hash(y).*

Authentication functions can provide user authentication (passwords), message authentication using message authentication codes (MAC) and are also used in digital signatures to provide non-repudiation. The data origin authentication points to the integrity of the message source and it ensures that a particular message is coming from a particular source and that it has not been altered in between.

1.2.1 Hash Based Message Authentication Codes

Hash based message authentication codes (HMAC) are mainly used for data integrity and authentication applications. In order to meet the data integrity and data origin authentication in secure communication, the US National Institute of Standards and Technology (NIST) has

recommended the use of Keyed Hash based Message Authentication Code (HMAC) [7] based on the Secure Hash Standard [8]. The SHS are a set of cryptographically secure hash algorithms (SHA) specified by NIST.

HMAC along with SHS is used in several standards and protocols such as SSL, IPsec, and Internet Key Exchange (IKE) for the purpose of authentication and integrity verification. HMAC uses a cryptographic hash function to calculate the message authentication code (MAC) along with a key. In the beginning MD5 and SHA-1 were used in HMAC for instantiating the hash. The security of HMAC is directly dependent on the cryptographic strength of the underlying hash function. Since researchers have found effective collisions on both MD5 [9] and SHA-1 [10] all the applications which use HMAC are forced to move towards the use of a more secure SHS. Google has announced that starting from November 2014 the Chrome browsers will be showing security warnings for those websites with SSL certificates using SHA-1 [11].

In applications such as SSL and IPsec, implementing the HMAC/SHA as software consume a high proportion of the web server's processing power. Hence the use of embedded systems such as ***hardware accelerators*** for performing cryptographic computations is gaining popularity [12]. Due to its high speed and security, Application Specific Integrated Circuits (*ASIC*) were initially preferred for such applications. But it is costly to update a particular crypto algorithm on an ASIC chip due to high design costs.

On the other hand, Field Programmable Gate Arrays (FPGA) have faster design cycles, generally supporting hardware reprogramming ability and are becoming popular in the design of embedded systems. As compared to ASICs the FPGAs have several advantages in the context of implementing cryptographic algorithms [13] . Some of them are listed below:

Flexibility: Most of the cryptographic protocols such as IPsec, SSL are algorithm independent which means they can accommodate more than one encryption or authentication algorithm. IPsec for example allows the use of AES, Triple DES, and HMAC-SHA. These algorithms typically will be updated in the future. Due to its flexibility FPGAs can be reconfigured easily with new algorithms.

Resource Efficiency: In the case of hybrid protocols such as SSL, a private key and public key algorithm is used. The session key exchange uses the public key algorithm and the data

encryption is done using a private key algorithm. The same FPGA may be used for implementing both algorithms by means of run time reconfiguration [14].

Architecture Efficiency: To make an architecture more efficient, it can be made with specific parameters. In the case of a cryptographic algorithm these parameters can include the key, underlying finite field, etc. With the use of FPGA architectures can be designed and optimized for a certain set of parameters.

Cost Efficiency: FPGA offers a low cost platform to implement algorithms in universities for studying architectures, side channel analysis etc.

The above mentioned features offered by the FPGAs make them an attractive platform for implementing cryptographic applications. But one might wonder how secure these FPGAs are. Unlike Application Specific Integrated Circuits (ASIC) the commercial FPGAs may not have any specific feature to prevent side channel attacks. This thesis proposes a low cost environment to study the viability of fault injection using clock glitches on an FPGA partial implementation of SHA-256. The SHA-256 algorithm was selected mainly due to its importance in applications such as SSL, IPsec, and Secure Boot.

1.3 Thesis Overview

This thesis focuses on the viability of fault injection attack using clock glitches on an FPGA partial implementation of SHA-256 architecture. The thesis is organized as follows: Chapter 2 introduces fault attacks, different methods of fault injection, HMAC and SHA-256 algorithms. A literature review on previous research in fault attacks is also included in this chapter. Chapter 3 illustrates the methodology and experimental set up for generating the clock glitch. It is followed by introducing the FPGA board used. The two architectures considered for implementing the SHA-256 on FPGA are presented in Chapter 4 along with their simulation and synthesis results. The experimental results showing the viability of the set up time violation attack using clock glitches are also presented. Finally Chapter 5 provides a summary and conclusion of this work along with recommendations for future work.

Chapter 2

Background and Related Work

This chapter is dived into four sections. The first section provides a review of fault attacks – methods of fault injection and different types of faults. Section two provides a survey on previous research in fault attacks. The third section describes the HMAC-SHA-256 algorithm followed by a discussion of previous implementation techniques. The chapter concludes with section four that discusses the previous research on HMAC-SHA Fault Injection.

2.1 Introduction to Fault Attacks

Fault attacks are a type of invasive side channel attack where malicious faults are injected into the cryptographic device. The output will be erroneous and careful analysis of such outputs called **Differential Fault Analysis** have proven to be highly effective [15] [16] [17]. A wide range of fault attacks has been reported and they differ on the basis of method of fault injection and types of faults.

2.1.1 Fault Injection Methods

The most common fault injection methods are as follows

- **Fault injection using glitches**
    - Variation in clock – The attacker can tamper with the clock signal and vary its length by generating glitch cycles. If the length of a clock signal is shortened it can result in errors varying from corrupting a single byte to multiple bytes [18].

The two most important effects of clock glitch based fault attacks are skipping an instruction and corrupting the data.

- Variation in power supply – The attacker can generate glitches in the power supply line by injecting power spikes. Voltage underfeeding by using a separate unit that can tap into the device's power supply line may also inject faults. These methods have been proven effective in the case of ASIC implementations of AES [19].

- **Fault injection using heating**

  All electronic devices have a specific range of operating temperature. The devices will not work properly outside this range and extreme temperature can result in modification of RAM cells. The temperature side channel can be used for data leakages in the case of CMOS devices and has been demonstrated in a practical attack on RSA implementation [20]

- **Electromagnetic fault injection**

  Electromagnetic pulses may be used to inject transient faults into a device [21]. This attack can even be carried out on packaged circuits and does not require de-capsulation of the device since the electromagnetic pulse can travel through non metallic surfaces. For example, a fault injection attack using electromagnetic pulse on a software implementation of AES running on an 8 bit RISC microcontroller has been demonstrated [22] .

- **Optical fault injection**

  The non volatile memory of microprocessors are sensitive to UV irradiation and these memory cells can be manipulated by high intensity laser beams. For example an experiment showing how a non-volatile memory can be erased using UV radiation has been performed [23] based on the AES implemented on an 8 bit microcontroller. The results shows that the attacker can precisely control the number of bits that are modified. The 256 bit *S-box* table of AES is manipulated and a single byte change in the table together with 2,500 pair of correct and fault cipher text, the key can be recovered with 90% probability. Only a low cost UV-lamp and a standard decapsulation were required for this experiment.

2.1.2 The Fault Types and Fault Models

The faults resulting from a fault injection attack can be broadly classified as ***transient faults*** and ***permanent faults.***

- **Transient faults** – Transient faults are provisional faults with effects being reversible, i.e. the circuit can resume its original behavior by reset or by stopping the fault stimuli. The effect of transient fault can cause the system to skip an instruction, store a wrong value etc.
- **Permanent faults** – The value of cells are changed in permanent faults. These type of faults that are generally hard to inject can damage the data or the code being executed. Memory ciphering and scrambling physical address using different techniques are used to inject permanent faults.

The above mentioned fault types can result in the following fault models.

- **Bit and Byte** – A fault can result in modifying a single bit, byte or multiple bytes. Byte being the basic level for storing data, it is easy to change the value of a byte as compared to changing the value of a bit. A bit error model requires highly sophisticated equipment and it can break almost all cryptosystems.
- **Specific and random** – In the case of a specific fault model, the attacker assumes that the data will be changed to a specific value upon injecting a fault. In general it is considered much easier and practical to induce random errors.
- **Static and Computational –** The error can be induced into the memory or while performing an operation. The former type known as static is hard to induce whereas computational errors are more practical [24]
- **Data and Control** – When the fault attack results in modifying the data it is called a data fault. An example of a control fault is such that the attack will result in skipping an instruction.

2.2 Previous Research in Cryptographic Clock-based Fault Injection

This section presents a survey on the previous research in clock based fault injection attacks on some cryptographic algorithms other than HMAC/SHA. The fault based attacks on HMAC/SHA are reported in section 2.3.3.

A practical fault attack results on six different type odd symmetric block ciphers listed on ISO/IEC 18033-3 implemented on an LSI : AES, DES, Camellia, CAST-128, SEED and MISTY1 is described in [25]. They have considered seven AES modules with different implementation styles and five modules for each of the five other block ciphers.

An experimental setup to provide a clock signal with a glitch to the LSI is developed and it can be used to inject a fault into any desired round. They used a prototype ASIC called the "ISO/IEC 18033-3 Standard Cryptographic LSI" and it was mounted on the Side Channel Attack Standard Evaluation Board called SASEBO-R. The LSI was controlled using a host PC via an FPGA on the SASEBO board. The LSI controller set up in the FPGA supplies the required clock signal with glitch. Two external clocks with same frequency but different phases were selected and was switched at proper timing to generate glitches. The experimental setup is shown in Figure 2.1. Their experimental results claim that the for the same glitch width, the position of the error byte remains the same for different rounds.

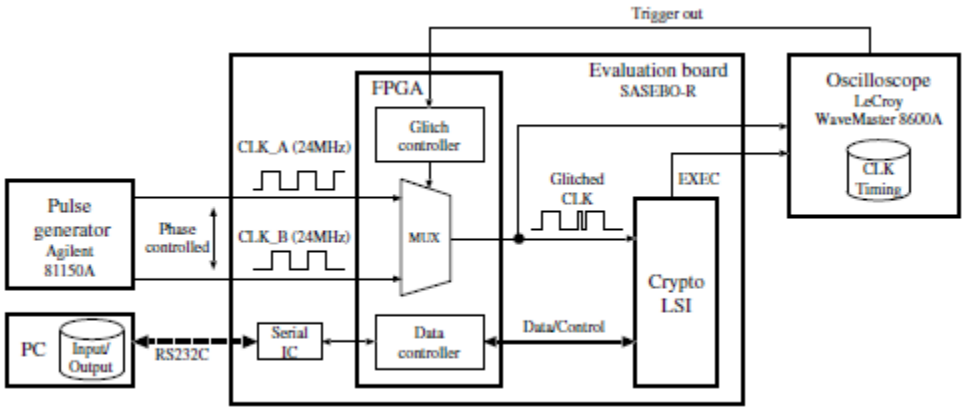


Figure 2.1 Experiment setup [25]

The interrelation between number of error bytes, glitch round and glitch width; position of error bits, glitch round and glitch width were investigated in their work. They succeed in secret key retrieval using Piret's attack [26] from all the AES modules with one pair of correct and faulty cipher text. To conduct this attack they started by performing a normal encryption using a plaintext selected at random. After this a clock glitch was generated in the final round of AES and maximized the glitch width by means of controlling the phase difference between two clock sources. That resulted in corrupting a single byte in the cipher text and it was recorded. Then they changed the position of the glitch to a different round and maintained the same glitch width. The corresponding faulty cipher text was recorded. They reduced the candidate key space using Piret's attack and executed a brute force analysis on the reduced key space. Their results show an average reduced key candidate space of 2^{32} and computational time of 8~35 minutes. The timing simulation results or the implementation information of the experiment presented in this work are not open.

In her Master's thesis [27] Masoumeh Dadjou has presented an FPGA based uniform testbed for clock glitch fault injection targeting the Advanced Encryption Standard (AES). The glitch generator module and the AES code provided in the DPA contest is implemented on an FPGA located on the SASEBO-G-II. It uses the same design idea of the glitch-clock generator used in [28]. The fault injection method described in this work was targeting the critical paths in the AES S-box rather than the key schedule. This work presents a detailed analysis on the glitch characteristics that causes the fault and also the effect of the fault.

The range of glitch period in which the fault injection happens is studied and the characterization of glitch is done based on its width and period. The author claims that the glitch width is an important factor in injecting the fault based on the experimental results. Their results show that decreasing the glitch period from a certain maximum to a minimum value within a window can increase the fault probability. The number and position of faulty bytes and bits with respect to the width of the glitch was experimented. With decreasing glitch period the number of fault bytes is said to have increased. The glitch was designed and characterized to be used in AES, and it was implemented on Xilinx FPGA with hardware modules that are not portable for use on other boards such as Altera DE2.

A glitch-clock generator based on FPGA for evaluating fault injection attack is presented in [28]. It makes use of the clock management feature available on Xilinx FPGA boards and can be integrated on a single FPGA without the need of any external circuitry. The characteristic of this glitch generator is carefully studied using Side-Channel Attack Standard Evaluation board (SASEBO-G). Figure 2.2 shows the block diagram view of the fault injection system used in the work which was implemented on the SASEBO board. The glitch generation concept explained in this paper is given on Figure 2.3.

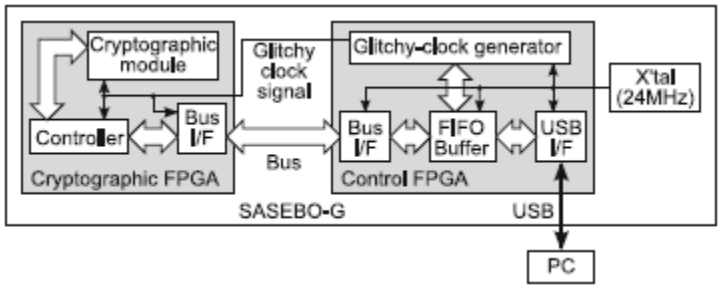


Figure 2.2 Fault injection system on SASEBO-G [28]

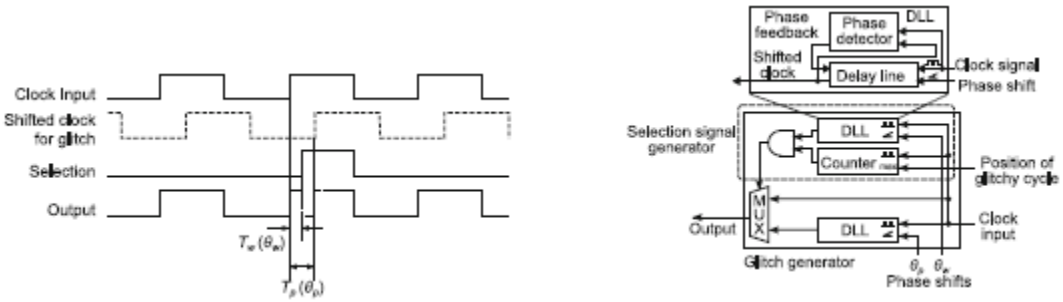


Figure 2.3 Glitch generator concept [28]

The authors claim that the timing of the glitches can be controlled at steps of 0.17 ns and they have also presented a safe-error attack against RSA using the proposed system. For this they

used an RSA processor, the proposed generator with a glitch period of 9.7 ns. In the case of safe error attack, the attacker uses power traces rather than output to distinguish between dummy operations and normal operations. Instead of this they have captured pair of power traces those ones with fault injection in multiplication operation and the ones without. The equality of operations were evaluated carefully by calculating the difference between the waveforms. In the case of dummy operations the amplitude of the differential power traces will be smaller. Thus the attacker can predict if the target operations are real or not. The authors claim to have obtained the secret exponent bits in RSA using their proposed method.

The clock glitch attack in the presence of heating described in [29] and they use the same method of glitch generation proposed in [30] and [28]. They have done practical experiments on an 8-bit AVR microcontroller and the results show an increased success rate of glitch attack with increase in temperature.

2. 3 SHA-256 and HMAC Algorithm

This section gives an overview of the *SHA-256 and HMAC* algorithms with descriptions on the individual steps and operations involved. There is a lot of published works on the hardware implementation of SHA- 256 and HMAC [31] , [32] [33] [34] A brief description of a couple of those works are also included in this section.

2.3.1 The Secure Hash Algorithm – 2 (SHA-2)

Secure hash algorithm (SHA) was proposed as FIPS 180 SHS standard in 1993. In 1994 an existing flaw was corrected and the standard was referred to as SHA-1. The SHA-2 published in 2008 refers to the FIPS PUB 180-3 [8]. SHA-2 is a family of hash functions, namely, SHA-224, SHA-256, SHA-384 and SHA-512. The input block size (B) for SHA-224/256 is 512 bits and for SHA-384/512 it is 1024 bits. The algorithm output is called message digest whose length *(L)* varies according to the algorithm used. SHA-224 produces a message digest of 224 bits, SHA-

256 produces 256 bit length message digest and in general SHA-*x* produces an *x*- bit message digest. The SHA-256 algorithm has two steps, namely **pre-processing** and **hash computation**. In pre-processing the original message is split into *N* blocks of *B* bits each. If the length of original message is not a multiple of block size, message padding is performed. The initial hash values are set according to the specification [8]. The hash computation is over 32 bit words (SHA-256) and the algorithm performs 64 iterations *(j)*. The algorithm use six logical functions and are given below:

$$Ch(x,y,z) = (x \wedge y) \oplus (\bar{x} \wedge y),$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z),$$
$$\sum_0 (x) = ROR^2(x) \oplus ROR^{13}(x) \oplus ROR^{22}(x),$$
$$\sum_1 (x) = ROR^6(x) \oplus ROR^{11}(x) \oplus ROR^{25}(x),$$
$$\sigma_0 (x) = ROR^7(x) \oplus ROR^{18}(x) \oplus SHR^3(x),$$
$$\sigma_1 (x) = ROR^{17}(x) \oplus ROR^{19}(x) \oplus SHR^{10}(x),$$

The $ROR^n(x)$ is the circular rotation of the variable *x* by *n* positions to the right and $SHR^n(x)$ is the shifting of variable *x* by *n* positions to the right.

PREPROCESSING

Message Padding

The SHA-256 accepts messages of length up to 2^{64} bits as its input. The message is then padded such that its final length is a multiple of 512 bits. The padding process can be explained as follows: for a message *M* of length *l*, append "1" to the message followed by *k* zeros, where *k* is the smallest non-negative integer satisfying the equation $l+k+1=448 \bmod 512$. Finally add a 64 bit block which is equivalent to the values of l in binary. The length of the padded message should now be a multiple of 512 bits.

Message Parsing

The next step is to parse the message into *N* different blocks of 512 bit each denoted as $M^{(1)}$, $M^{(2)}, \ldots, M^{(n)}$. Each of these blocks are divided into 16 sub-blocks of 32 bit each denoted as $M_0^{(i)}$, $M_1^{(i)}, M_2^{(i)}, \ldots, M_{15}^{(i)}$. For each message block *i*, $1 \le i \le N$, a four step digest round is performed as follows:

Step 1: Initialize the eight working variables

$$a = H_0^{(i-1)}, b = H_1^{(i-1)}, c = H_2^{(i-1)}, d = H_3^{(i-1)},$$
$$e = H_4^{(i-1)}, f = H_5^{(i-1)}, g = H_6^{(i-1)}, i = H_7^{(i-1)},$$

Step 2: Prepare the message schedule

$$W_t = \begin{cases} M_t{}^{i}, 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, 16 \le t \le j-1 \end{cases}$$

Step 3: For *t = 0* to *j-1* do:

$$T_1 = h + \sum_1(e) + Ch(e, f, g) + K_t + W_t,$$
$$T_2 = \sum_0(e) + Maj(a,b,c),$$
$$h = g,$$
$$g = f,$$
$$f = e,$$
$$e = d + T_1,$$
$$d = c,$$
$$c = b,$$
$$b = a,$$
$$a = T_1 + T_2,$$

HASH COMPUTATION

Step 4: Compute the i^{th} intermediate hash value $H^{(i)}$

$$H_0^i = a + H_0^{(i-1)}, H_1^i = b + H_1^{(i-1)}$$
$$H_2^i = c + H_2^{(i-1)}, H_3^i = d + H_3^{(i-1)}$$
$$H_4^i = e + H_4^{(i-1)}, H_5^i = f + H_5^{(i-1)}$$
$$H_6^i = g + H_6^{(i-1)}, H_7^i = h + H_7^{(i-1)}$$

After *N* blocks of messages are processed, the final message digest is obtained by concatenating the hash values as given below.

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

2.3.2 The Hash Based Message Authentication Codes (HMAC)

The message authentication codes (MAC) are widely used for data integrity and authentication [35]. HMAC involves the use of a cryptographic hash function along with the use of a secret key to compute a MAC. It was originally designed by Mihir Bellare, Ran Canetti, and Hugo Krawczyk in 1996 and later got published as a FIPS standard in 2002 [7]. HMAC can be described as follows:

$$\boldsymbol{HMAC(k,m) = [h((k \text{ xor } opad), h((k \text{ xor } ipad),m))]_t}$$

In the above equation *h()* stands for the cryptographic hash function, *k* is the secret key and *ipad and opad* are constants defined by the standard. The key must be *B* bytes long and the input text must be *n*-bits where $0 \leq n \leq 2^B - 8B$. The result of HMAC is obtained by taking the most significant *t* bytes out of the final hash such that $L/2 \leq t \leq L$, where *L* is the output length of *h ()* (e.g. 256-bits in the case of SHA-256).

In the following section a brief review of the different proposed hardware implementations of SHA-256 and HMAC is included.

2.3.3 Previous Research in HMAC/SHA-2 Architectures

The work presented in [33] introduces a specialized computational platform catering to the requirements of SHA-256 and HMAC. The authors have considered five hardware/software partitioning levels for the mentioned algorithms and depending on these levels the operations were implemented in hardware. The logical operations involved in SHA -256 mentioned in Section 2.3.1 were all implemented as Nios II custom instructions after considering their execution times. Figure 2.4 shows the architecture proposed for implementing the HMAC/SHA-256. The hash operation was designed to be performed on hardware and the rest of the computation was executed in software. Their approach allows the designer to implement the programs in C and move parts of the application execution easily to the hardware modules. They have analyzed all the partitioning levels with respect to the implementation area, program size and throughput. In the case of general purpose applications that do not involve the frequent

integrity check and MAC computation, the proposed approach is very useful. Their experimental results shows that the custom instruction approach can accelerate the speed of computation of message authentication codes and is very helpful in the case of constrained embedded systems that are used for integrity checks and authentication purposes.

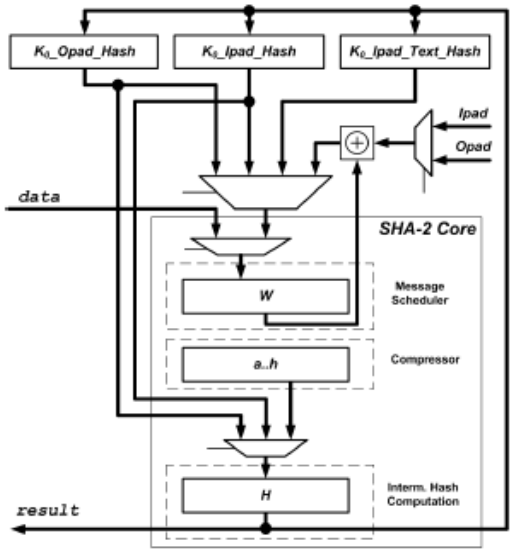


Figure 2.4 HMAC/SHA-256 architecture [33]

Dadda et al. in [36] presents a high speed ASIC unit for SHA-256 with reduced critical path delay. A basic scheme using carry save adders and a scheme using delay balancing to further shorten the critical path is also presented in their work (Figure 2.5). Their architecture was proposed mainly for ASIC implementations and the authors have examined several quasi-pipelined methods also. The quasi-pipelined method was suggested to overcome the difficulties of the existing pipeline methods. The delay balancing was used to move certain function from the critical path of the design to shorter paths so that the critical path can be reduced. The results of their work claim to have a reduced critical path in the compressor block which makes it posssible to achieve a clock frequency of 819 MHz suitable for high speed implementations.

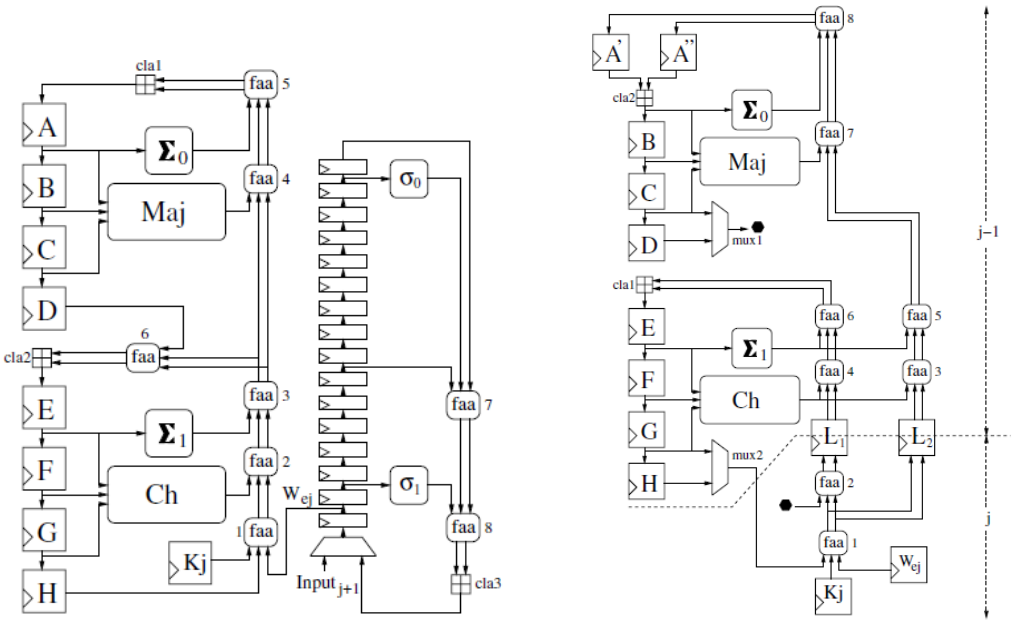
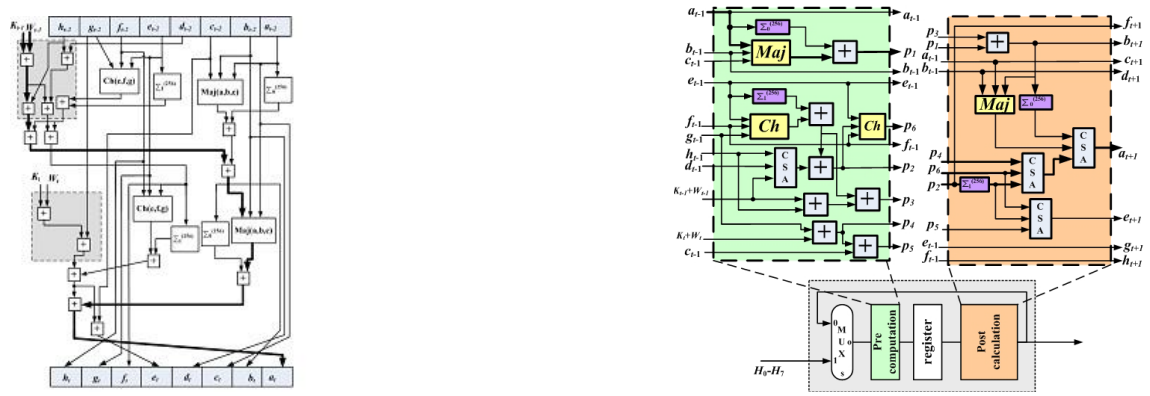


Figure 2.5 SHA -256 design using CSA (left), Delay balancing and pipelining (right) [36]

A high speed SHA-256 module, with higher throughput, ideal for IPsec software/hardware co-design is presented in [37]. The critical paths in the SHA-256 algorithm was determined and to optimize the design, a set of operations were unrolled as shown in Figure 2.6 (a). Some of the intermediate values were pre-calculated during the course of certain operations and this was applied to the partially unrolled design. It was further optimized by means of data pre-fetching which is a system level pre computation technique. The values of W_t & K_t were computed outside the operation block and their sum was stored in registers thus making these values available at the beginning of each round. A block diagram of the proposed SHA-256 operation block is given in Figure 2.6(b) which is capable of carrying out multiple operations in a single clock cycle with an increase in throughput.



(a) Two unrolled SHA-256 operation blocks (b) Proposed design

Figure 2.6 High Speed SHA-256 Hashing Module [37]

The proposed method is claimed to have high throughput (2190 Mbps) and small area requirement making it suitable for software/hardware co-design.

The FPGA based SHA-256 architecture presented in [32] was implemented on a Xilinx Virtex XCV300E-8 FPGA .They obtained a maximum throughput of 65 MB/s at 88 MHz clock. The work presented in [31] uses optimization techniques based on pipelining and unrolling to design a VLSI architecture for SHA-256 and SHA-512. It is mainly based on the quasi pipelined method suggested by Dada et al. in [36]. A VLSI implementation of SHA-256 with reduced critical path delay is given in [34] which uses a new structure for the adder by compromising delay and area.

2.4 Previous Research in Fault Attacks on HMAC/SHA-256

As compared to research work on fault attacks on other cryptosystems, there has not been much work done on the fault analysis of cryptographic hash functions. This could be due to the fact that hash functions are used primarily for computing message authentication codes and don't involve the use of secret keys. But now that HMAC algorithms use SHS and the security of HMAC relies on the security of its underlying hash, it is of utmost importance that fault attacks

on hash functions needed to be studied and further researched. This section reviews some past research in Fault Analysis on Secure Hash Algorithms.

The differential fault analysis (DFA) on the SHA -1 compression function is presented in [38]. The fault model described in this work is based on the assumption that the attacker can change the value of 32-bit words (SHA-1 operates on 32 bit word) in the last 20 rounds of the algorithm to some priori random values. There are two phases for the proposed DFA – the first phase is to computationally eliminate the final addition and the second phase is to apply the SHACAL-1 attack described in [39]. The attack was simulated on a PC using Python scripting language to show the feasibility. By simulating the injection of faults in different locations of the compression function and processing around 1000 faulty outputs the authors claim that secret inputs can be extracted with high probability.

A fault injection attack based on the method proposed in [39] is used against an FPGA implementation of SHA 512. The hardware architecture has an error injection module to emulate fault injection and the physical injection of fault is by using glitch on the power supply [40]. The fault model discussed is based on flipping the control bits to reduce the round number of SHA 512 algorithm at specific times [41]. This is said to have an impact on reducing the round numbers thereby giving intermediate values as the final hash.

The effect of the proposed attack is demonstrated by applying it to the hardware implementation of HMAC on ADM-RXC-5T1. The error injection module on the FPGA will inject errors into the HMAC module at specific times as controlled by a host PC. It is not a real attack but the error injection module just emulates the fault injection mechanism. A countermeasure to prevent such attacks by detecting the end-of-round errors is also suggested in this paper. This attack is different from the work presented in this thesis as real clock glitches are created to inject faults.

A key recovery attack using fault injection on HMAC/NMAC based on MD-5 and SHA-256 hash function is presented in [42]. The proposed attack is based on the similar idea described in [40] which is to reduce the rounds in compression function by means of injecting faults. Figure 2.7 shows the proposed fault model on HMAC where *f* indicates the hash function.

Their attack results claim to recover the *n*-word secret key with [*n/3*] fault injections in the case of HMAC-SHA-256. However it is a theoretical attack and there is no information available on the feasibility or computational complexity of this proposed method.

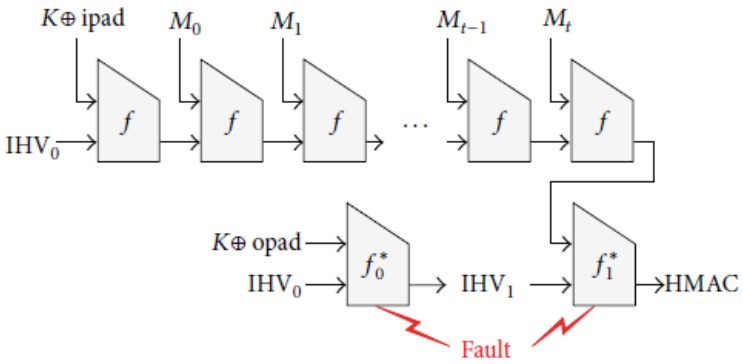


Figure 2.7 Fault injection attack on HMAC [42]

None of the above mentioned works utilized real injected faults in their attacks and they all have varying complexity. The varying complexity refers to the type of fault model used, the number of faulty outputs required to extract the secret inputs, and time of execution.

2.5 Existing Countermeasures

The need to make hardware implementation of crypto systems robust against fault injection has resulted in the study of fault detection schemes [43]. Most of the proposed countermeasures are based on redundancy and can be classified as follows:

Hardware Redundancy: based on duplicating one or more functional blocks. The error can be detected by comparing the outputs of two distinct blocks that implement the same function.

Information Redundancy: based on error detecting codes that uses check bits generated from the original message. This error code will be propagated with the message and can be validated at the end [44].

Temporal Redundancy: based on repeating the process twice or computing the inverse function in some cases. But this can result in reducing the throughput.

Juliato et al. has presented FPGA based Fault tolerant schemes for SHA-256 architectures to ensure secure communication in space applications [45] [46]. They have employed Hamming codes to protect the registers of SHA-256 and analyzed the resistance of proposed architectures against single event upsets (SEUs).

To provide a sufficient level of security against fault attacks, several factors must be considered. This will require combination of different classes of countermeasures to ensure overall protection. The countermeasures against fault attacks are not experimented in this thesis.

2.6 Summary

This chapter began with an introduction to fault attacks along with different methods of fault injection and types of faults. A brief review about the fault attacks on various cryptosystems like AES was discussed. The SHA-256 and HMAC algorithms were described along with a discussion of previous implementation techniques. A review of specific fault injection research on HMAC/SHA was presented. It was shown that very few researchers have studied real (not simulated or emulated) fault injection on HMAC/SHA. Since the hardware implementation of SHA-256 is also considered as a part of this thesis, the architecture used for implementing it will be discussed in Chapter 4. The existing fault injection attacks on HMAC/SHA are mostly based on the technique proposed by Choukri and Tunstall [40]. In the next chapter the design of clock glitch along with the glitch generator based on an FPGA will be discussed. The proposed method aims at inducing setup violations in the critical paths which will be discussed later.

Chapter 3

Methodology and Experimental Setup

This chapter describes the methodology used to design and test the proposed fault injection testbed, including the SHA-256 core and clock glitch generator. The details of the Altera DE2 board, Phase Locked Loop and the set up of an FPGA based clock glitch circuit is described in this chapter.

3.1 Fault Injection

There are other approaches apart from clock glitching for injecting faults into the FPGA, which maybe for example heating the device, under powering the device, use of electromagnetic radiations and other approaches as outlined in Section 2.2.1.

Fault injection using heating the device would require temperature sensors to verify the increase in temperature. The thermal fault injections can result in invasive faults in the sensitive areas of a device. In addition, excessive heating could permanently damage the entire circuit. Fault injection using voltage underfeeding would require a custom circuit that can drop the feeding voltage below a certain value. The use of electromagnetic radiation may also induce faults in a device without tapping into it. However electromagnetic radiation typically affects the device uniformly through the power signal. Hence the parts of the device that should not be subjected to fault injection may require proper shielding.

The main objective of this thesis is to incorporate a low cost environment including the circuit under test and the fault injection circuit onto a single FPGA chip. In addition this setup is to be

used to study the effect of fault injection on SHA-256. Hence the above mentioned methods were discarded and the use of an onboard PLL to set up a simple clock glitch generator was considered.

3.2 Setup Time Violation Using Clock Glitches

As discussed in Chapter 2, fault injection attacks are becoming popular in the field of hardware and embedded cryptography. With the use of dedicated cryptanalytic techniques, an attacker can extract secret information from the faulty bits or bytes or words. The complexity of these fault injection attacks are found to be less as compared to classical cryptanalysis techniques. This section discusses the theory behind clock glitches and how they result in setup time violation.

The configurable logic blocks (CLB) that form the basic logic unit of most reprogrammable FPGAs are made up of two components – flip flops and look up tables. The flips flops are used to synchronize the logic and save the logic states between clock cycles. There can be hundreds of flip flops that define the current state of a circuit. The combinational logic in between these flip flops calculates the next state or computes data. These logic blocks are implemented on FPGAs with the lookup tables or static RAM and a simplification is given in Figure 3.1. On every clock edge a flip flop has to latch a value on its input and hold that value constant until the next clock edge.

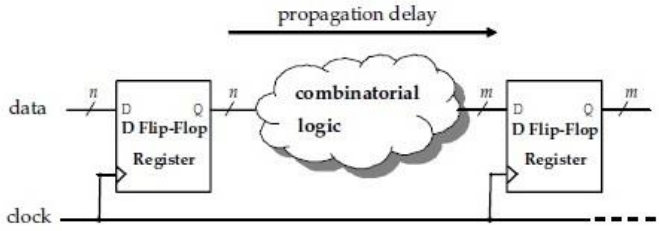


Figure 3.1 Representation of Digital ICs [30]

The registers latch the data on the rising clock edges. Between two sequential rising clock edges, the data propagates from one register through combinatorial logic blocks that might modify the

data and back into a register. The time taken by the data to propagate through the combinatorial logic is called the *propagation delay*. The maximum propagation delay of an entire circuit or chip is called its *critical path*. Another important parameter is the *setup time*, which is a characteristics of the flip-flops. The setup time is the minimum time a signal has to be available or stable at the input to the buffer before the active edge of clock arrives. Both the critical path and the set up time are parameters that determine the maximum operating frequency or speed of any circuit. For a circuit to operate correctly, the clock period must be greater than the sum of critical path and setup time.

$$T_{clk} > t_{critical} + t_{setup}$$

In the case of fault injection attacks using clock glitches, the attacker uses several methods to violate the above condition. An option that can be considered is to switch between the clock signals so as to shorten the length of a single clock cycle [25]. This shortening can result in corrupting the data to be latched.

A similar situation is shown in figure 3.2, where a normal clock is at the top and the clock with a glitch is at the bottom of the figure. The r_in and r_out signals represents the input and output of the data register respectively and CLK is the clock signal. Under normal operation the time intervals between the positive edges for each clock is longer than the circuit's maximum delay and hence the intermediate states (m_n) are correctly stored. Assuming the normal clock is set to a maximum clock frequency for the design, the glitch generates a clock cycle which is shorter than the sum of critical path and setup time causing a *setup time violation*. As a result the incorrect output r_in of the combinational logic is latched in the register r_out, shown as incorrect value of X (variable) in Figure 3.2. These kinds of fault injections which are transient in nature can induce faults without leaving tamper evidence.

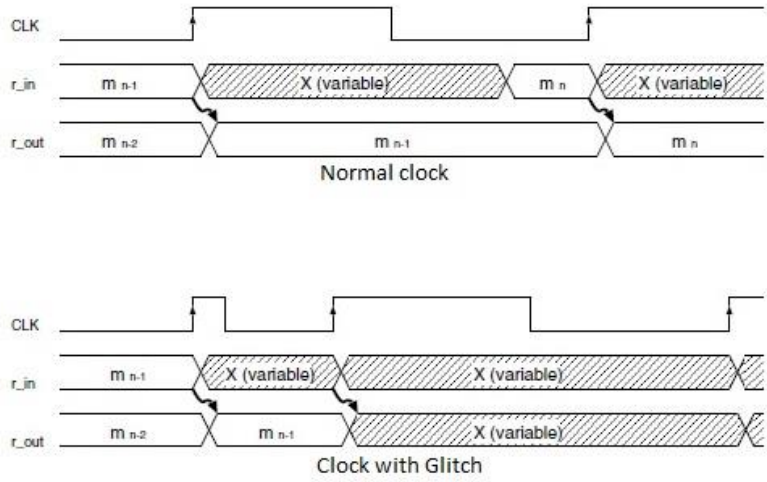


Figure 3.2 Normal clock (above) and glitch clock (below) [25]

3.3 Generating Clock Glitch

The circuit for generating the clock glitch is implemented in an Altera DE2 board. The clock glitch generator has two main components, namely a Phase locked loop (PLL) and a multiplexer. The PLL available on the DE2 boards is used to generate the clocks of a desired frequency and phase shift (delay) with reference to an input clock. It can generate three clock signals namely ***C0, C1*** *and* ***C2*** out of which *C1 and C2* will be used for the clock glitch circuit. A multiplexer is used to switch between these clocks. The select input of the multiplexer is generated by another component that has an internal counter which monitors the time to enable the select input to the multiplexer.

The block design file of the PLL with the input and output is shown in Figure 3.3. The *locked* output signal of the PLL is used to indicate whether or not the PLL has locked onto the reference clock. A logical high on this signal indicates a stable PLL clock output in phase with the PLL reference input clock [47]. The locked port can be connected to any general purpose I/O pin on the FPGA. The ***inclk0*** is the input clock to the PLL and it is driven by ***CLOCK_50*** on the Altera

DE2 board. The waveform simulation showing the various input and output signals of the PLL is given in Figure 3.4.

The parameters of the output clock signals (namely frequency, phase shift, duty cycle, and clock multiplication/division) can be set using the *ALTPLL MegaWizard* feature available on the Altera Quartus II programming tool. In this example the frequency, duty cycle and phase (delay) of *C0* was set to 50 MHz, 50 % and -3 ns respectively. The parameters of *C1* was 50 MHz, 50%, 0ns and that of *C2* was 114.28 MHz, 50 %, 0 ns. The output frequency of 114.28 MHz was achieved by keeping the clock multiplication / division to be 16/7. A detailed description of how to configure the PLL is given in Appendix A.

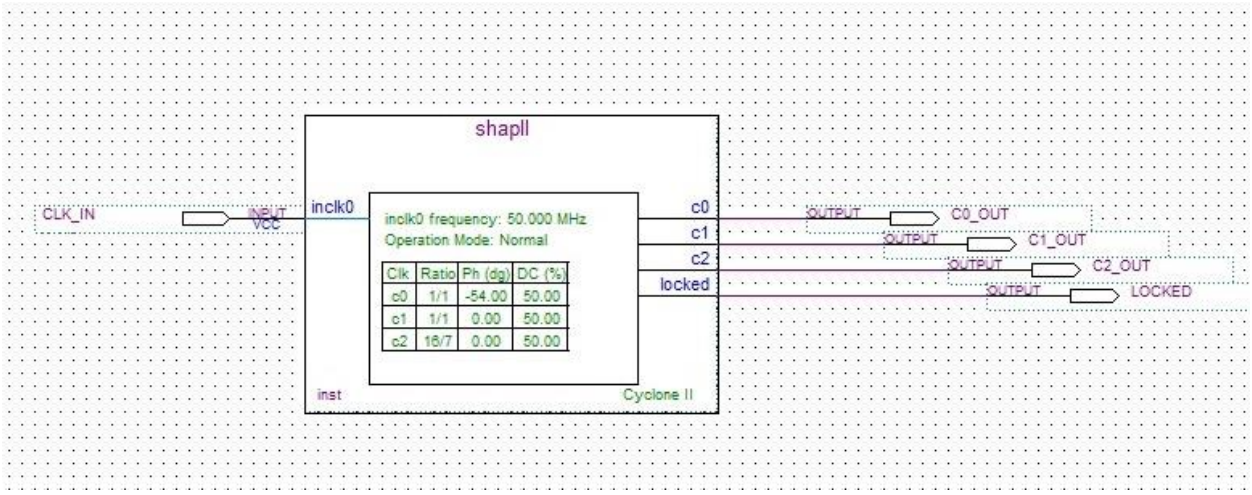


Figure 3.3 Block diagram file of Altera Phase Locked Loop

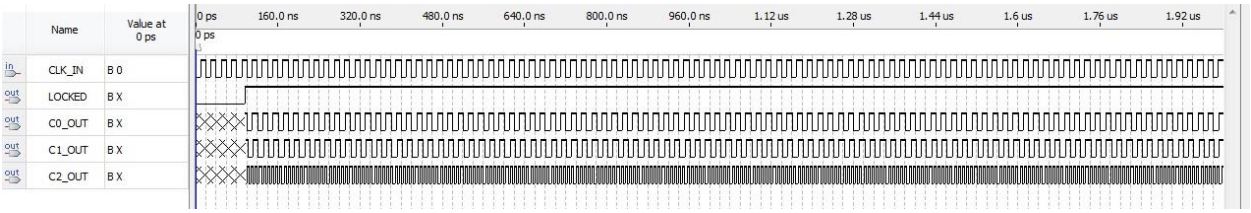


Figure 3.4 Waveform Simulation showing the input and output signals of the PLL

The clock signals *C1* and *C2* is are fed to a 2*1 multiplexer (designed using logic gates) whose output is the *CLK_OUT* signal. The multiplexer select signal *(S)* is generated by a component shown as SEL_GEN in Figure 3.5 that has an internal counter. The counter is designed to run at 50 MHz which is the default frequency of the circuit under test. This allows the circuit to keep track of the SHA-256 operation rounds, where each operation is one clock cycle. The clock switching takes place between rounds 55 and 60. The clock switching will produce glitches in *CLK_OUT*, the multiplexer output which is the clock of the SHA-256 module. The block diagram showing the connections between PLL, multiplexer and select signal generator is shown in Figure 3.5. A sample waveform simulation showing glitches in the output clock at the time of clock switching is shown on Figure 3.6.

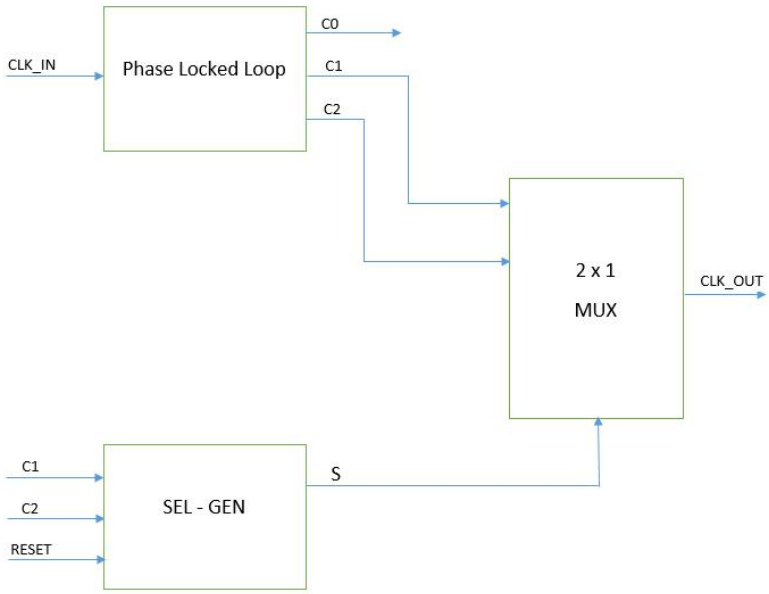


Figure 3.5 Block diagram showing the components of glitch generator circuit

The *SEL_GEN* that generates the select input(S) for the multiplexer was described in VHDL with three inputs namely *C1*, *C2* and *reset.* It has a counter that increments the count on the rising edge of clock *C1*. When the program begins the select signal *S* is set to 1 and the counter will keep track of the SHA-256 operation rounds. Once the count reaches 54 the select signal will be made 0 and then back to 1 when the count is 60. This switches the clock to *C2* from round 55 to 60 and back to *C1* from round 61.

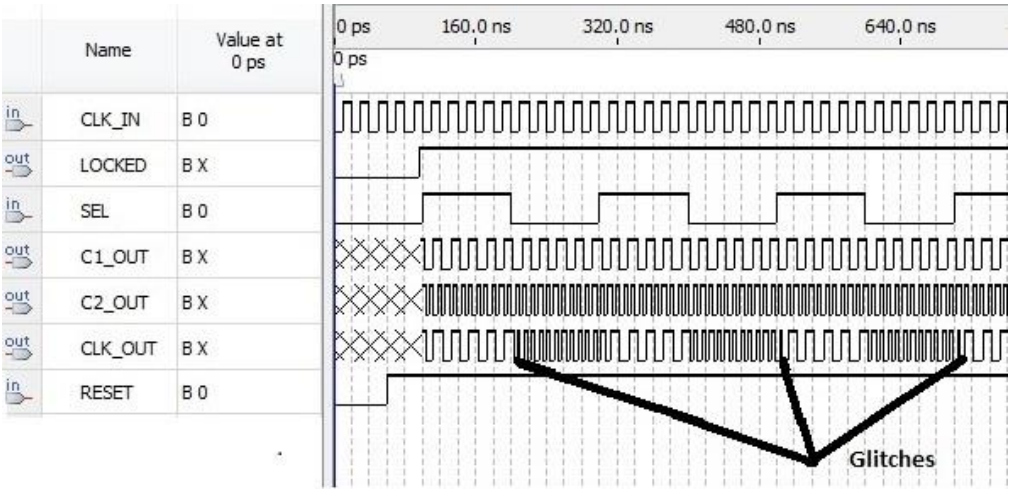


Figure 3.6 Waveform simulation showing presence of glitch during clock switching

3.4 Methodology and Experimental Setup

This section provides an overview of the FPGA board used for implementing SHA-256 and the other software, hardware requirements needed for the experiment. The targeted device along with the system architecture is also discussed.

3.4.1 Targeted Device – Altera DE2 Board

One of the main objectives of this thesis is to incorporate the entire system on a single FPGA, specifically the Circuit under Test, Glitch Generator and the Interface. For this an Altera DE2 board was selected. The block diagram of DE2 board is shown in Figure 3.7. The Altera DE2 board is built around a Cyclone II programmable logic device and it provides a flexible platform for developing and testing embedded system applications. The Altera Quartus II design tool allows the user to reconfigure the Cyclone II FPGA device to implement different designs. The components which are designed and implemented on the FPGA include the following:

- Phase locked loop (PLL) for generating clock signals
- SHA-256 core -custom VHDL component

- Component for timing and the glitch
- Nios II embedded processor to communicate with the SHA-256

The Altera Megawizard tool was used to design and create the PLL, and Nios II interface whereas other elements used in the design were specified using VHDL. A photograph that shows the layout of a DE2 board is given on Figure 3.9. There are 50 MHz and 27 MHz oscillators on the board that can be used as possible clock sources. The board supports an external clock input as well. An external power supply source provides the power supply connector with 9.0 V. A USB Blaster port is provided for configuring the board. The JTAG interface of FPGA chip can be used to communicate with the board which connects to the USB port on the host PC. This interface will transfer the hardware design from the PC onto the Cyclone II device using the hardware configuring process. It can also be used to communicate with the Nios II processor configured on the FPGA, through downloading software to run on the Nios II processor as shown in Figure 3.8.

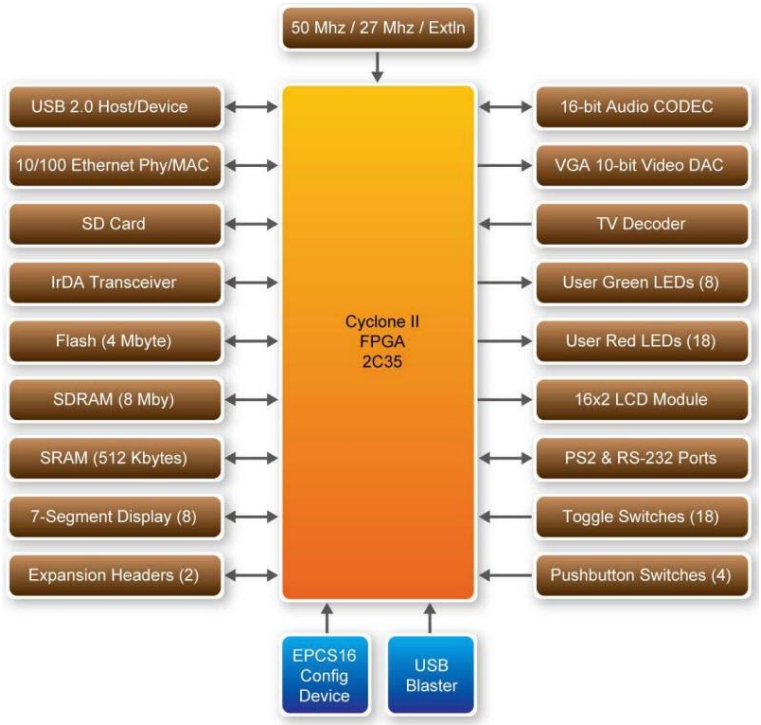


Figure 3.7 Block diagram of Altera DE2 board [48]

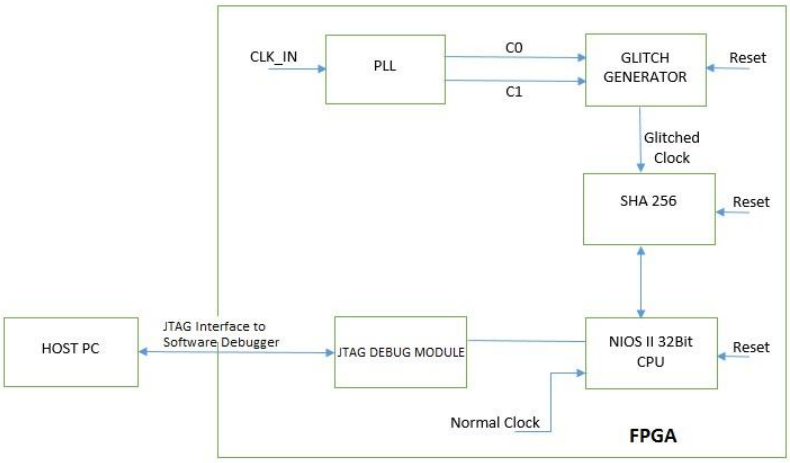


Figure 3.8 System Architecture

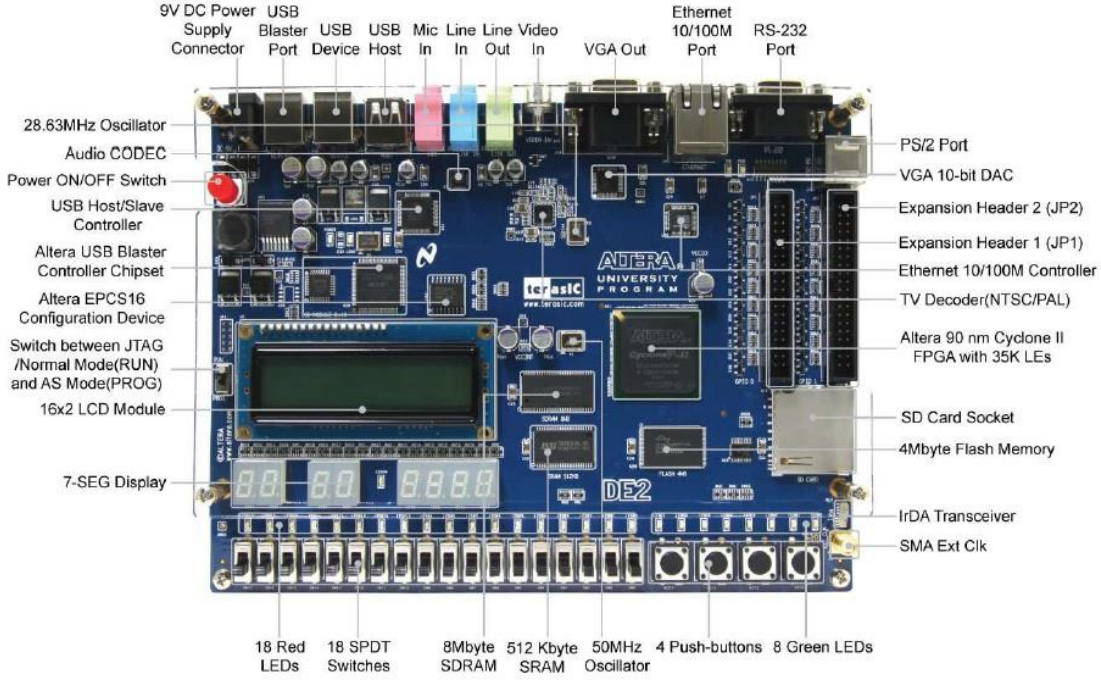


Figure 3.9 Altera DE2 board layout [48]

3.4.2 Methodology

The FPGA-based fault injection testbed was designed such that the message input to SHA-256 could be controlled and outputs from SHA-256 after a specific round could be observed. Furthermore fault injection was controllable, or in other words the specific round for the fault injection could be controlled. In order to have the controllability and observability, it was necessary not only to have the SHA-256 and clock glitch circuitry on the FPGA, but also the Nios II processor. NIST test vectors [8] were utilized to verify the designed architectures.

As outlined in Section 2.3, SHA-256 will accept a 512 bit block input message and produce a 256 bit output hash. In the SHA-256 proposed architecture, a ready (rdy) signal is produced which is used to communicate with the Nios II interface. The input message block is fed to the input ports of SHA-256 through the Nios II interface. The *rdy* signal is initially 0 and after writing the final hash values to its output ports, the SHA-256 module will make the *rdy* signal 1. Nios II waits until *rdy* is 1 in order to read the results from its pio pins which are driven by the output ports of SHA-256. A C program is used to send input messages to and read hash results from the SHA-256 core through the Nios II interface and the interconnections between them are made through the Avalon bus. The software is downloaded onto the RAM on the DE2 board and is run on the Nios II processor. Note that the SHA-256 core uses glitched clock and the Nios II processor operates on a normal 50 MHz clock. After the *rdy* signal is made 1, the SHA core becomes inactive. Thus the Nios II processor can retrieve the data from the 8 output ports of the SHA core. These output ports can be made to read the final hash value or the values of working registers for specific rounds (a *to h*). This was done by modifying the VHDL code and downloading it to the board. The number of registers connected to the output port is always 8 so that the top level entity in SHA-256 design remains unchanged.

As discussed earlier the glitch generator has components to keep track of the clock cycles and switch the clock signals in the rounds were a glitch is desired.

3.5 Summary

In this chapter the methodology and experimental set up for injecting clock glitch faults into an FPGA based implementation of SHA-256 is discussed. The glitch generator was designed using the PLL available on Cyclone II device and a multiplexer. The architecture used for implementing SHA-256 will be discussed in the next chapter along with simulation and synthesis results.

Chapter 4

Experiment and Results

This chapter presents the two architectures considered for implementing SHA-256 on FPGA and their testing and verification followed by the area, frequency analysis. It is followed by the fault injection on SHA-256 using an FPGA based clock glitch circuit, experimental results, analysis and a brief comparison to previous works.

4.1 Memory-Based Architecture

The memory-based hardware design for the SHA-256 core consists of logical operations, 32 bit adders and memory units to store the constants (K_t) as well as the output of message scheduler (W_t). The SHA-256 architecture can be divided into four main units: Message Scheduler, Constant Memory, Compressor and Intermediate hash computation. The first 16 words M_t of the message are stored in the first 16 locations of the memory unit for message scheduler. It will take 16 clocks to do so and from clock 17 to 64 the corresponding values of W_t are calculated and stored into memory using equation given in Step 2 (Section 2.3.1). The initialization of the registers *a to h* can be done during reset. Once the message scheduling and initialization is completed, the compressor will use registers *a to h*, and W_t *and* K_t to determine the new values of *a to h*. The SHA-256 algorithm performs 4 rounds of compression and this can be controlled internally by using an iteration counter. After 64 rounds of iteration the intermediate hash computation is performed as given in Step 4 (Section 2.3.1).In the case of a single block message the hash computation is completed at this point but for a multi-block message a new execution cycle will begin.

This architecture was designed and compiled on Altera Quartus II 10.1 platform with the targeted device *EP2C35F672C6*. It was simulated using Altera Vector Waveform Simulation and results were verified using NIST test vectors as shown in Figure 4.1. However the final verification in the hardware was not successful as there were errors in the SHA-256 output. After significant debugging it was determined that the value of *W* from clock cycles 17 to 64 was corrupted. To calculate each value of W_t four previous values of *Wt* are required. As long as *W* is being implemented in memory, this cannot be achieved in single clock. It was determined that this was the likely error in the hardware design, despite successful VHDL simulation of the architecture. In order to overcome this a second design was considered based on registers and will be described in the next section 4.2.

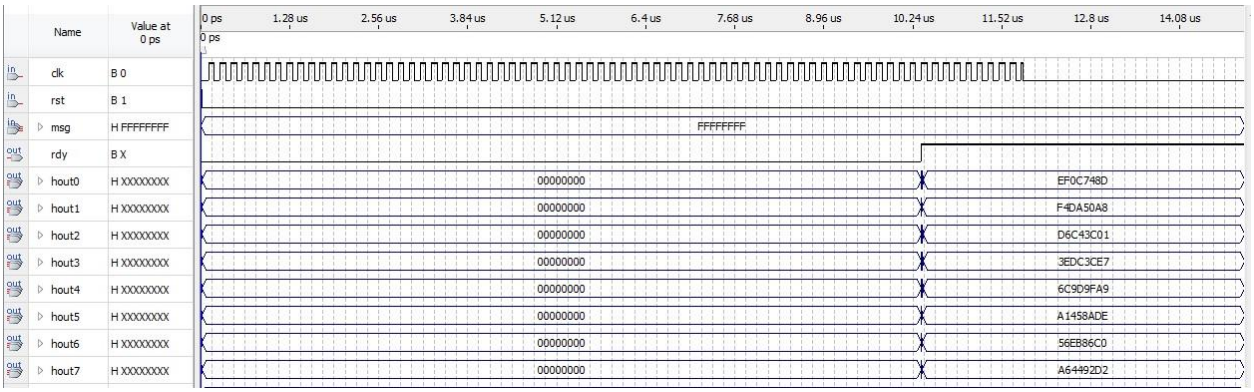


Figure 4.1 SHA-256 waveform Simulation

4.2 Pipelined Register-Based Architecture

The second proposed design for SHA-256 consists of logical operations, 32 bit adders, memory unit for constants and shift registers for the message scheduling. The total number of registers required is 1048. The array of SHA-256 constants will be synthesized as a memory block which in turn requires 2048 bits. The architecture can be divided into the following blocks as shown in Fig. 4.2: Message scheduler, constant memory, compressor and Intermediate hash computation. The message scheduler uses 16 shift registers number 0 to 15 as shown in Figure 4.2. These can be initialized either in serial or in parallel. Parallel initialization requires us to store the values

internally and then load them directly into registers W_0 to W_{15}. In the case of serial initialization the first 16 words M_t of the message can be shifted into the registers over 16 clock cycles. It will take one clock to initialize the working variables and initial hash values. This can be done at reset. Once the initialization is complete, the hash compressor will use registers *a* to *h*, W_t and K_t to calculate new values for *a* to *h* as shown in Figure 4.2.

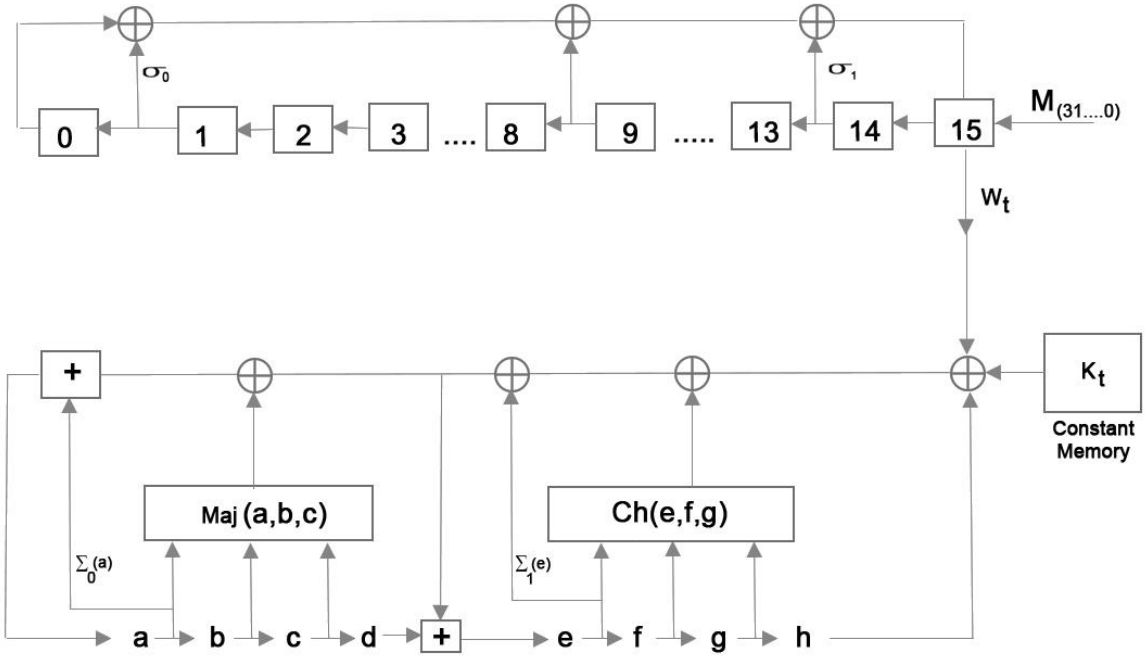


Figure 4.2 SHA-256 Architecture – message scheduling and hash computation

The algorithm will run for 64 iterations and can be monitored by an iteration counter. After 64 iterations the intermediate hash computation is performed by adding the content of registers *a to h* with the initial hash values as given in step 4 of SHA-256 algorithm given in Section 2.3.1. In the case of a single block message the hash computation is completed at this point but for a multi-block message a new execution cycle will begin. Note that in HMAC-SHA-256 the first message block is the secret key, hence at least 2 message blocks will be computed within the SHA-256. This architecture was designed and compiled on Altera Quartus II 10.1 platform with the targeted device *EP2C35F672C6*. The verification was done in two stages. For the behavioral

simulation the Altera Quartus II Vector waveform simulation was used and it was tested using the NIST test vectors and verified result. The circuit was synthesized and the final verification was done by running the program on the DE2 board. The Nios II soft core processor interface was designed to communicate with SHA-256 core. A C software program was written to send the input and read the output from the SHA-256 core.

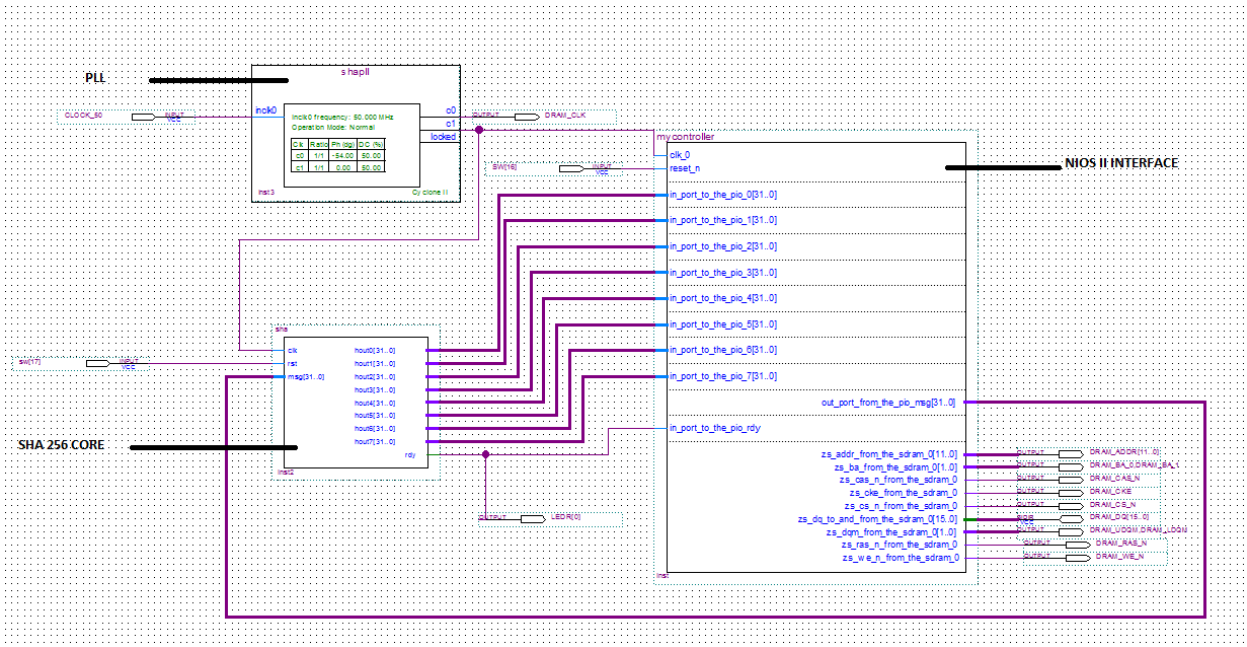


Figure 4.3 Schematic design file showing SHA-256 and Nios II Interface

4.2.1 Area and Frequency Analysis

The proposed architecture for SHA-256 was described in VHDL. An assortment of hardware such as Nios II processor, PLL for generating clock signals and the input output peripherals (PIO) for Nios II were added using the Altera SOPC builder. It was implemented on Altera Cyclone II *EP2C35F672C6* FPGA using the Altera Quartus II 10.1 tools. The input clock for the PLL was chosen as 50 MHz. The design was tested using different clocks and the maximum clock frequency was found to be 114.29 MHz which was obtained by selecting the clock multiplication factor as 16 and division factor to be 7. The next possible frequency that can be

produced by the PLL was 120 MHz however the SHA-256 core was not compatible with this frequency.

The implementation area measured as the number of total Logic Elements used (LE) for the proposed SHA-256 partial implementation is given in Table 4.1. The logic element utilization was found to be 5175 out of the 33,216 available logic elements on the targeted device. The throughput is defined as the number of message bits processed per amount of time, measured in bits per second (bps). Throughput can be computed as *Throughput = (B*F)/c*, where B is the message block size in bits, F is the operating clock frequency and c is the number of clock cycles needed to compute the hash value.

Logic Elements (# equivalent gates)	5175 out of 33,216
Total Registers	2783
Maximum Frequency	114.29 MHz
Throughput	914.29 Mbps

Table 4.1 SHA-256 partial implementation Area and Frequency Analysis

4.3 Setup Time Violation on SHA-256 Using Clock Glitches

This section describes the initial exploration of clock glitching. In particular it was observed that several parameters had to be carefully determined in order to produce a viable single clock glitch applicable for injecting faults into SHA-256.

The SHA-256 registers were latched on the positive clock edge and hence it was required to have a glitch with time between the rising edges smaller than that of a normal clock. The maximum operating frequency of SHA -256 core, 114.29 MHz corresponds to a time period of 8.75 ns. The

next available frequency of the PLL, 120 MHz corresponds to 8.3 ns which was found to be incompatible with the SHA- 256 core. This clearly means that the sum of critical path and setup time is in the range of 8.3 ns to 8.75 ns. It is therefore important to generate a clock glitch which is shorter than 8 3ns to have successful fault injection.

As discussed in section 3.4 the clock glitch circuit is connected to the SHA-256 core. The clock switching is controlled by the select input of the multiplexer. As the multiplexer output changes, there will be sudden change in the frequency of the clock which in turn produces glitch in the clock line of SHA-256. This is expected to result in a data misread in the rounds between 55 and 60. The complete circuit showing the SHA-256 core, the Nios II interface, PLL, the glitch generator circuitry is given in Figure 4.4.

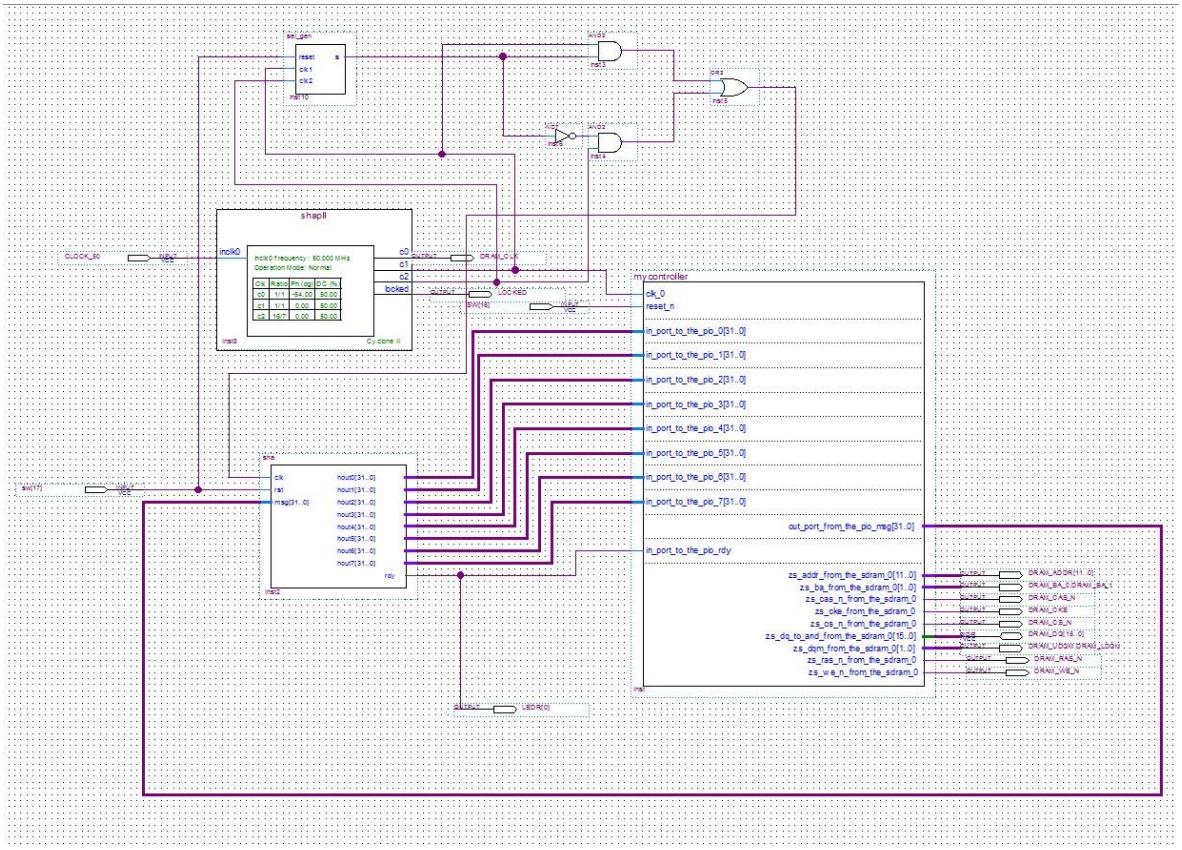


Figure 4.4 CUT Schematic Design File

The output waveform of timing simulation (Figure 4.5) shows the glitch in output clock (*CLK_OUT*) during clock switching. It should be noted that the simulation given in Figure 4.5 is a timing simulation of the VHDL design and not the device simulation. This is why the final hash output is not showing any effect of the glitch.

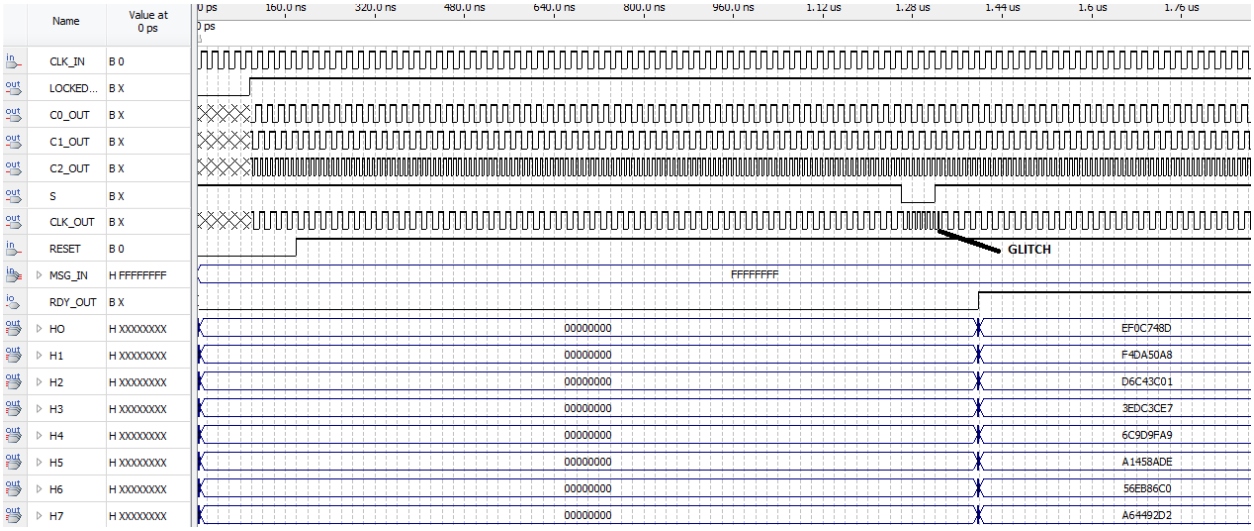


Figure 4.5 Waveform simulation showing the presence of glitch

4.3.1 Varying Parameters of Clock

The PLL available on the FPGA produces three clocks *C0, C1, C2* out of which *C1* and *C2* are used in the glitch circuit. The parameters of these two clocks can be varied using the *ALTPLL* Megawizard plugin manager available on the Altera Quartus 10.1 tool. The external SDRAM interface for Nios II requires a special clock which was met by specifying the parameters of clock *C0* to be 50 MHz, -3ns delay. The parameters of *C1* were kept fixed (50 MHz, 50% duty cycle, 0ns delay) and that of *C2* was varied. The output frequency, duty cycle and delay of *C2* was varied using the *ALTPLL* Megawizard. The two clock signals *C1* and *C2* with different frequencies and delay can be fed into a multiplexer and is switched depending upon the select (*S*) signal. The range of frequencies selected for *C2* was between 50 MHz and 114.29 MHz (the maximum frequency of SHA-256 core). Different combinations of frequencies from this range were chosen and were tested using the timing simulation to verify the presence of a clock glitch.

The frequencies between 50 MHz and 80 MHz were discarded for testing on hardware as they were producing glitched clock cycles whose period were almost equal to or even higher than the normal clock frequency. It was noted that the shape of the glitch changed as the delay of *C2* was varied. The delay or phase shift of the PLL output clock with reference to the input clock can be varied using the *ALTPLL* Megawizard. As the frequency of *C2* approached 80 MHz, the CLK_*OUT* signal was observed to have clock cycle with period less than the normal clock during clock switching. All the clocks that were selected for testing on the FPGA are given in Table 4.2.

Output Frequency	Duty Cycle	Delay
80 MHz	50%	0ns to 5ns
90 MHz	50%	0ns to 5ns
100 MHz	50%	0ns to 3ns
110 MHz	50%	0ns to 3ns

Table 4.2 Clock Parameters

In the following waveforms *CLK_IN* is the input clock to the PLL and signals labelled *LOCKED, C1_OUT, C2_OUT* respectively are the locked output, clock 1 and clock 2 produced by the PLL. The switching is controlled by the select input (*S*) to the multiplexer which is generated by *SEL_GEN* component as discussed before. The test was repeated for different frequency and delay values of C2 with clock switching done between round 55 and 60 and the presence of glitch was confirmed (circled in the waveforms) as given in Figures 4.6 ,4.7 and 4.8.

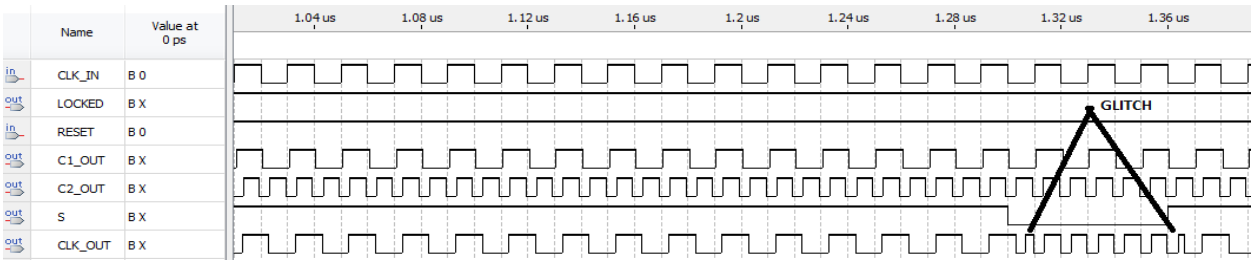


Figure 4.6 C2 with 100 MHz and 1ns delay

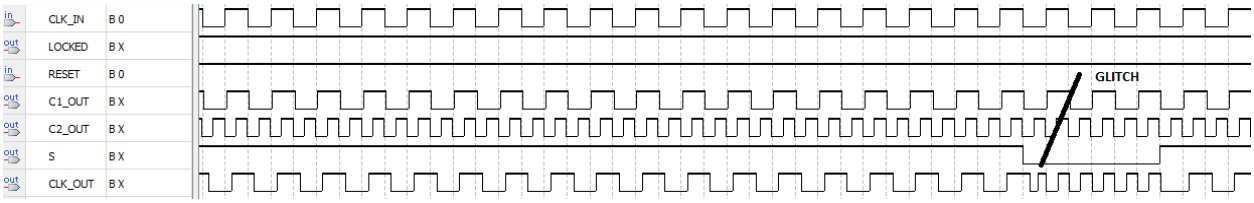


Figure 4.7 C2 with 100 MHz and 2.5 ns delay

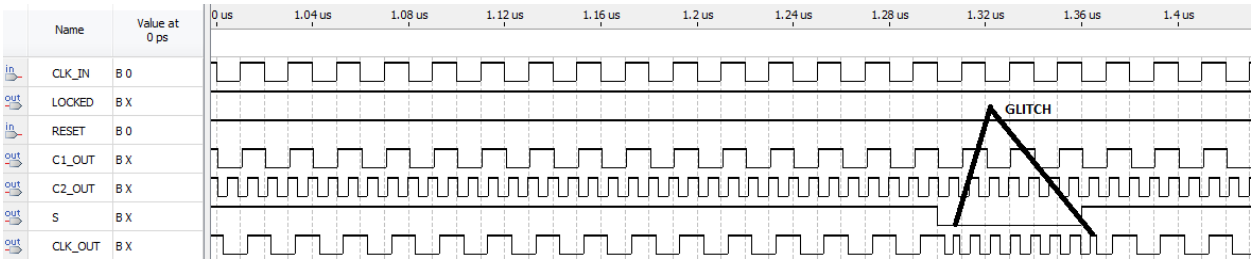


Figure 4.8 C2 with 110 MHz and 2.75ns delay

After testing various frequency and delay values for *C2* by using timing simulation as described above, it was decided to test it on hardware to confirm whether or not these glitches can induce faults.

4.3.2 Testing the Glitch on FPGA Implemented SHA-256

The glitch circuit with parameters for creating glitches, from the previous section, was examined in conjunction with SHA-256 in order to determine if faults would be generated. The glitch generator circuit along with SHA-256 core was downloaded on to the Altera DE2 FPGA board and the C program was made to run on the Nios II soft core processor to obtain final hash output. The clock switching was performed between rounds 55 and 60 as described earlier.

The experiment was first run without glitch at 50 MHz and 114.29 MHz clock to verify that the circuit could run at this speed. This was done to make sure that the error in the final hash is due to the glitch and not because of an incompatible clock. An incompatible clock can result in metastability condition and a corrupted output.

Fault injection was successful by changing frequency of the circuit from 50 MHz to 110 MHz and also from 50 MHz to 100 MHz between rounds 55 and 60 as discussed before. The value of registers at the end of round 58 and 60 were checked. The tables below shows the register value samples for different tests for the input block *(0xFFFFFFFF)*.

Registers	Expected result	Obtained result	Number of bits in error
a	*64D09224*	*25A10307*	*12*
b	*DD734AC5*	*CBB30864*	*10*
c	*6E2F65B1*	*CF0BEDF0*	*9*
d	*16788381*	*0E30CB01*	*7*
e	*BBE50132*	*6AFD02C7*	*13*
f	*384AC793*	*B05A469B*	*7*
g	*94856D88*	*1001E90C*	*8*
h	*B5DD28B7*	*3DC5203F*	*7*

Table 4.3 Test one C2 100 MHz: register contents at end of round 58

Registers	Expected result	Obtained result	Number of bits in error
a	*64D09224*	*E0B80610*	*11*
b	*DD734AC5*	*9C315E8D*	*8*
c	*6E2F65B1*	*EE674135*	*7*
d	*16788381*	*13FA02C3*	*8*
e	*BBE50132*	*830739B2*	*11*
f	*384AC793*	*78488F11*	*8*
g	*94856D88*	*BCCD25A4*	*9*
h	*B5DD28B7*	*AD9D00C7*	*8*

Table 4.4 Test two C2 110 MHz: register contents at end of round 58

Registers	Expected result	Obtained result	Number of bits in error
a	*998C47AD*	*D413CCCE*	*18*
b	*45AEF1AB*	*76CF5D0A*	*14*
c	*5E2580BE*	*78DDE6E4*	*16*
d	*64D09224*	*4A9FEA74*	*15*
e	*4A63C5B9*	*A21CA4FE*	*18*
f	*52F36C3D*	*36FAD138*	*13*
g	*3DCB2DFC*	*3F07B356*	*14*
h	*BBE50132*	*B7C19A32*	*9*

Table 4.5 Test three C2 100 MHz: register contents at end of round 60

Registers	Expected result	Obtained result	Number of bits in error
a	*998C47AD*	*D92A8810*	*17*
b	*45AEF1AB*	*14A4F0F3*	*9*
c	*5E2580BE*	*2735006F*	*11*
d	*64D09224*	*E7B07C4A*	*10*
e	*4A63C5B9*	*79AED409*	*14*
f	*52F36C3D*	*40E754DE*	*12*
g	*3DCB2DFC*	*2DFA1CCD*	*10*
h	*BBE50132*	*3BD712B4*	*10*

Table 4.6 Test four C2 110 MHz: register contents at the end of round 60

The change in clock source is expected to produce a glitch and as a result the generated clock will be shorter than the sum of max delay and set up time. This will result in the set up time violation resulting in the latching of an incorrect value by the combinational logic elements and

the error propagates to next rounds. The register contents at the end of round 58 and 60 as shown in the above tables validates this assumption.

The select signal going from 1→ 0→ 1 will result in clock switching from C1 → C2 → C1 and this produces two glitches and the errors in final hash output are due to multiple glitches. However the future work would expand this idea to design fault model based on single glitch. The experiment was repeated multiple times for a chosen *C2* (frequency, delay) and the generated fault values were found to be random.

After testing the hardware for varying values of *C2* it was noticed that the glitch produced during clock shift may or may not cause an error in final output. It was observed that there was no error in the final output for certain clock glitches but for some other glitches there was error in the final output. After analyzing the glitch waveforms it was found that the period and width of those clock glitches that couldn't produce any error was significantly larger than the one that produced the error. Even though the timing simulation results for clock switching from 50 MHz to 80 MHz and from 50 MHz 90 MHz had a glitch cycle, it didn't produce any error in the output while testing on hardware. The period of the glitch in these cases was found to be higher than 8 ns. It should be noted that the maximum operating frequency of SHA-256 core was found to be 114.29 MHz (period of 8.75 ns).

The glitches produced by the 100 MHz and 110 MHz had period mostly around 6.13 ns which is less than the period of maximum clock supported by the design.

From the synthesis results it was observed that the clock glitch circuit was successful in injecting fault for some of the glitches and caused error in final output. The period of the glitch is an important factor in producing error. It can be assumed that for all the values of glitch period less than a particular minimum value, it is not possible for the hardware to treat it as a clock and will not produce any error in the hash computation. Similarly for all glitch periods higher than a particular maximum value, the hardware treats it as a normal clock and will not result in a set up time violation. The main observations are listed below:

- The results of timing simulation of the glitch circuit shows that the clock switchover can produce glitches in the clock line. The period and width of the glitch depends on the

period of the individual clocks, their phase difference and the select signal that triggers the clock switching.

- The resulting glitches may or may not produce error in the output and the error injecting capability of a glitch is dependent on its period and width.
- In the case of a glitch generator using the onboard PLL, the glitch cycle can be generated only after the PLL has locked onto the reference clock as indicated by the *locked* signal.
- The hardware experiments confirmed that the glitch generator circuit can inject fault in the SHA-256 implementation.
- It can be assumed that fault injection happens in an interval of glitch width. The knowledge of this interval is very important for the attacker to precisely control the fault injection.

The position of faulty bytes or bits is a matter of further research. We cannot apply the idea of set up time violation to all bytes as different bytes will have different path delays. It requires a much more advanced experimental set up to analyze the nature of bytes that are more susceptible to set up time violation. The experiment was repeated with two different set of input blocks. As mentioned above certain bytes are more prone to faults which may be attributed to their path delay. Any particular bit that is fed as an input to a register gets there on its own logical path and has a different propagation time.

A change in the input block to the SHA-256 core will result in changes in propagation times and the probability of inducing a fault. The difference in propagation time can also be due to the difference in operations that the bytes undergoes.

In the case of SHA-256 compression function the registers *a* and *e* undergoes a different operation as compared to the other registers. Those bytes that are being used by the register *a* and *e* are supposed to have a higher delay as compared to the other registers (since they pass through more logical computations). This could be the reason why the registers *a* and *e* had more bits in error than other registers. The FPGA level architecture showing the logic element on the Cyclone II device after the place and route is given in Figure 4.8.

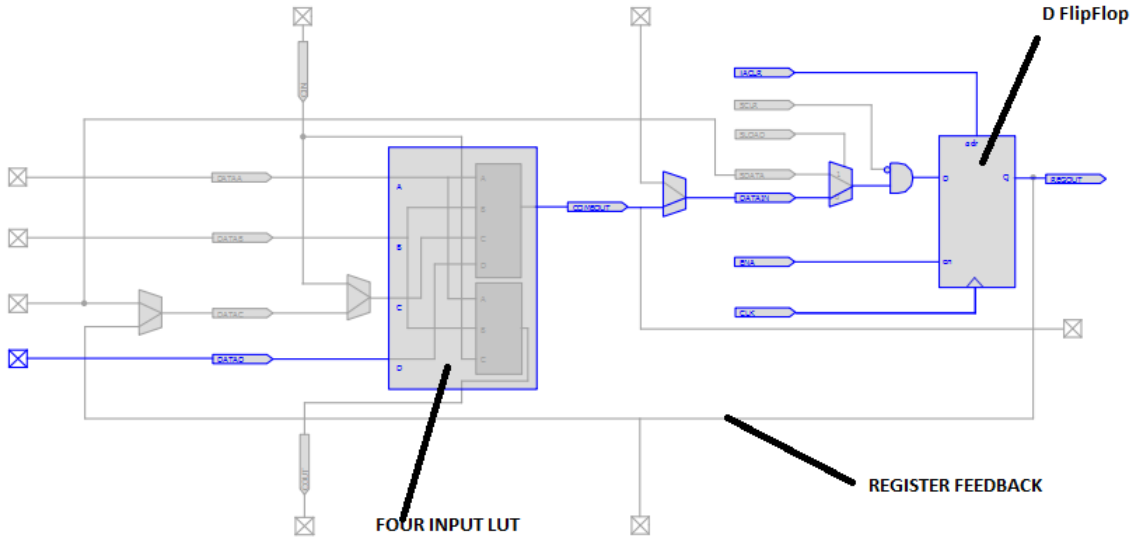


Figure 4.9 FPGA level architecture showing the logic element on Cyclone II device

In order to generate the glitches, the frequency, phase and duty cycle of the clock signals must be controlled very precisely. The use of an onboard Phase Locked Loop has a lot of limitation when it comes to higher accuracy and precision. The Phase locked loops on Cyclone II FPGAs don't support the dynamic phase configuration feature as compared to Stratix IV, Cyclone III family of devices. This feature will allow the user to dynamically adjust the phases of the PLL output clocks. A fine phase shift resolution of up to $1/8^{th}$ of the VCO frequency can be achieved using dynamic phase stepping in real time. Another important factor is the speed of clock switching – the use of high speed switches is expected to improve the performance. The idea to precisely control the clock parameters and high speed switching can be incorporated to set up an efficient clock glitch generator based on an FPGA, like the one in [28].

The main objective of this thesis was to research the vulnerability of FPGA based implementation of SHA-256 towards a simple fault attack using clock glitches within a testbed and hence exploring more sophisticated glitch generation circuit is beyond the scope of this work.

4.4 Comparison to Previous Works

The study of fault attacks on SHA-256 is a relatively new subject and the published research work on this topic is limited. The fault attack on hardware implementation of SHA-512 discussed in [41] uses glitches in power supply for the physical injection of fault. It is based on flipping control bits to reduce the number of rounds in SHA-512. Most of the fault injection attack against FPGA implementation of AES uses the method suggested by Piret and Quisquarter in [26]. The work discussed in [27] uses the glitch generator module and AES code from the DPA contest. The characteristics of the glitch that generated the fault and the effect of faults were also studied. As compared to the previous research, the work done in this thesis aims at incorporating the SHA-256 core and other circuitry for fault injection on a single FPGA board and to test its vulnerability towards clock glitch fault injection.

4.5 Summary

In this chapter the synthesis results for two different architectures for implementing SHA-256 in hardware are presented. The second register-based architecture was used to test the viability of a clock based fault injection on SHA-256. An FPGA based clock glitch generator was designed to inject the glitch that examined the feasibility of such fault injections in practical. From the experimental results it was concluded that fault injection is possible only for certain glitches. It is concluded that the period of the glitch is the key factor that decides the fault injection.

Chapter 5

Discussion and Conclusions

This section provides a brief summary of the thesis work and conclusion of the obtained results and future work.

5.1 Summary and Discussion

In this thesis the viability of a setup time violation attack using clock glitches on SHA-256 implemented in a FPGA was experimented. For this purpose a SHA-256 core was first designed and implemented on the DE2 board. Two slightly different architectures were considered for implementing SHA-256 and one of them was selected. It was tested with different inputs to verify correct functioning.

The clock glitch generator circuit was designed using the phase locked loop available on the DE2 board to produce the required clock signals and a multiplexer to switch between the clock signals. The select input for the multiplexer was generated using a component with an internal counter that keeps track of the SHA-256 operation rounds. The output of this circuit was tested and verified by means of timing simulation and the results confirmed the presence of glitches in the output clock line at the time of clock switching.

The effect of clock glitch on the SHA-256 architecture was experimented by testing the complete circuit on the DE2 board. The experiment was repeated for varying clock parameters and it confirmed that the onboard glitch generator could successfully inject faults in the SHA-256. The glitches are expected to produce set up time violation in the flip-flops of the FPGA and the

results clearly show the vulnerability of FPGA implementations of SHA-256 towards set up time violation attacks using clock glitches.

It was observed that the parameter of the glitches, mainly its period and width are the factors that determine whether fault injection is successful or not. The shape of the glitch was changed when the phase shifts (delay) of the clocks used were varied. This can be further explored to parameterize the shape of a glitch. The fault occurrence requires the attacker to have precise control over these factors. The use of an onboard PLL has its own limitations and better results can be expected with the use of PLL that allows dynamic configuration.

The main focus of this thesis was to study the viability of fault injection using clock glitches on a FPGA implementing SHA-256. Hence the nature and effect of the fault was not researched. Moreover a much more sophisticated experimental set up is required in order to study further detailed effects of fault injection on the FPGA. The clock glitch characterization and the resulting faults on SHA-256 can be researched further. It is important to note that the change in input block also changes the probability of injecting errors. This is probably due to the difference in path delay of the individual bytes.

To the best of our knowledge this is the first FPGA-based testbed with an onboard clock glitch generator using PLL to test the viability of fault injection on SHA-256 using clock glitches. The glitch generator implemented on the same FPGA provides a uniform testbed and it allows provides a low cost environment to study the viability of these kind of attacks on cryptographic systems. This proposed research is unlike previous research. For example the glitch generator discussed in [25] which is applied against AES, uses external equipment such as pulse generators and oscilloscopes to conduct the attack. On-chip glitch-clock generator for fault injection discussed in [28] is applied on an RSA cryptosystem and it uses safe error attack on RSA. The glitch characterization presented in [27] is based on the idea in [28] and have investigated the glitch characteristics and effects on AES S-box.

5.2 Conclusion and Future Work

In this thesis the viability of fault attack using clock glitches against SHA-256 was studied. The basic principle of fault injection and different methods to conduct such attacks in practice were

outlined in Chapter 2. The SHA-256 and HMAC algorithms were also introduced in this chapter and a survey of their hardware implementations was also done. Chapter 3 introduces the fault injection method using clock glitches and the proposed circuit to produce glitches. The targeted FPGA board to implement the SHA-256 was also introduced. The simulation results of the two architectures considered for implementing SHA-256 was presented in Chapter 4 along with the area and frequency analysis of the selected architecture. It was followed by the simulation and the implementation results of clock glitch circuit and the main observations.

A uniform hardware testbed to study the viability of fault injection attack using clock glitches is presented in this thesis. The entire circuit is based on a single FPGA without the need of an external pulse generator and oscilloscope. In this work the nature and characteristics of the faults are not explored. We have only examined whether or not the glitches produced error in the output of hash computation. Future work would involve expanding the circuit to have precise control over the glitch parameters namely the period and width. This way glitch characterization can be done and it will be very useful to design efficient fault models.

The uniform hardware set up presented in this work can be used to test the vulnerabilities of other cryptosystems towards similar attacks. So the future work would involve implementing and testing other cryptosystems as well.

Bibliography

[1] A. J. Menezes, S. A. Vanstone and P. C. van Oorschot, Handbook of Applied Cryptography, CRC Press, 1996.

[2] A. Barenghi, L. Breveglieri, I. Koren and D. Naccache, "Fault Injection Attacks on Cryptographic Devices : Theory , Practice , and Countermeasures," *Proceeings of the IEEE,* vol. 100, no. 11, pp. 3056-3076, 2012.

[3] F. Koeune and F.-X. Standaert, "A Tutorial on Physical Security and Side-Channel Attacks," in *Foundations of Security Analysis and Design III*, LNCS, 2005, pp. 78-108.

[4] P. Kocher, "Timing attacks on implementations of Diffie-Hellman , RSA, DSS and other systems," in *Advances in Cryptology-CRYPTO'96*, Santa Barbara, California, 1996.

[5] P. Kocher , J. Jaffe and B. Jun, "Differential power analysis," in *Advances in Cryptology CRYPTO'99*, 1999.

[6] D. Boneh, R. A. DeMillo and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology — EUROCRYPT* , 1997.

[7] "The keyed-hash message authentication code (HMAC)," NIST,FIPS PUB 198, March 2002.

[8] "Secure hash standard (SHS)," NIST,FIPS PUB 180-4, March 2012.

[9] X. Wang and H. Yu, "How to break MD5 and other hash functions," *EUROCRYPT,* pp. 19-35, 2005.

[10] X. Wang and H. Yu, "Finding collisions in the full SHA-1," in *Proceedings of CRYPTO*, 2005.

[11] "Google Online Security Blog," 05 September 2014. [Online]. Available: http://googleonlinesecurity.blogspot.ca/2014/09/gradually-sunsetting-sha-1.html.

[12] T. N. Institute, "Implementing Cryptography using an FPGA," [Online]. Available: https://www.tyndall.ie/content/implementing-cryptography-using-fpga. [Accessed 2014].

[13] T. Wollinger and C. Paar, "How Secure Are FPGAs in Cryptographic Applications?," in *Field Programmable Logic and Application*, LNCS, 2003, pp. 91-100.

[14] T. Wollinger , J. Guajardo and C. Paar , "Security on FPGAs: State-of-the-art implementations and attacks," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 3, no. 3, pp. 534-574, 2004.

[15] D. Boneh, R. DeMillo and R. Lipton, "On the importance of checking cryptographic protocols for faults," in *EUROCRYPT*, 1997.

[16] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *Security Protocols*, LNCS, 1998, pp. 115-124.

[17] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology-CRYPTO '97*, LNCS, 1997, pp. 513-525.

[18] F. Amiel, C. Clavier and M. Tunstall, "Fault analysis of DPA-resistant algorithms," in *Fault Diagnosis and Tolerance in Cryptography*, LNCS, 2006, pp. 223-236.

[19] N. Selmane, S. Guilley and J.-. L. Danger, ""Practical Setup Time Violation Attacks on AES," in *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on* , 2009.

[20] M. H. a. J.-M. Schmidt, "The Temperature Side Channel and Heating Fault Attacks," in *Smart Card Research and Advanced Applications*, LNCS, 2014, pp. 219-235.

[21] A. Dehbaoui, J. M. Dutertre, B. Robisson, P. Orsatelli, P. Maurine and A. Tria, "Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system," *Journal of Cryptology ePrint Archive Report 2012/123*, 2012.

[22] A. Dehbaoui, J.-M. Dutertre, B. Robisson and A. Tria, "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on* , Leuven, 2012.

[23] J.-M. Schmidt, M. Hutter and T. Plos, "Optical Fault Attacks on AES: A Threat in Violet," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on* , Lausanne, 2009.

[24] C. H. Kim and J.-J. Quisquater, "Faults, Injection Methods, and Fault Attacks," *IEEE , Design and Test of Computers*, vol. 24, no. 6, pp. 544-545, 2007.

[25] T. Fukunaga and J. Takahashi, "Practical Fault Attack on a Cryptographic LSI with ISO/IEC 18033-3 Block Ciphers," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, Lausanne, 2009.

[26] P. Gilles and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD," *LNCS*, pp. 162-181, 2003.

[27] M. Dadjou, "Analysis and Design of Clock-glitch Fault Injection within an FPGA," University of Waterloo, 2013.

[28] S. Endo, T. Sugawra, N. Homma, A. Takafumi and A. Satoh, "An on-chip glitchy-clock

generator for testing fault injection attacks," *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 265-270, December 2011.

[29] T. Korak, M. Hutter, B. Ege and B. Lejla, "Clock Glitch Attacks in the Presence of Heating," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2014.

[30] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robinsson and A. Tria, "When clocks fail: on critical paths and clock faults," in *Smart Card Research and Advanced Application*, Berlin Heidelberg, Springer, 2010, pp. 182-193.

[31] R. McEvoy, F. Crowe, C. Murphy and W. Marnane, "Optimisation of the SHA-2 family of hash functions on FPGAs," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, March 2006.

[32] K. Ting, S. Yuen, K. Lee and P. Leong, "An FPGA based SHA-256 processor," in *Proceedings of the Reconfigurable Computing is Going Mainstream , 12th International Conference on Field-Programmable Logic and Applications*, 2002.

[33] M. Juliato and C. Gebotys, "Tailoring a Reconfigurable Platform to SHA-256 and HMAC through Custom Instructions and Peripherals," in *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on* , Quintana Roo , 2009.

[34] L. Bai and S. Li, "VLSI implementation of high-speed SHA-256," in *IEEE 8th International Conference on ASIC*, 2009.

[35] C. Gebotys, "Data Integrity and Message Authentication," in *Security in Embedded Devices*, Springer, 2010, pp. 143-152.

[36] L. Dadda, M. Macchetti and J. Owen, "The design of a high speed ASIC unit for the hash function SHA-256 (384, 512)," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, 2004.

[37] H. Michail, G. Athanasiou, A. Kritikakou and C. Goutis, "Ultra high speed SHA-256 hashing cryptographic module for IPSec hardware/software codesign," in *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on* , Athens, 2010.

[38] L. Hemme and L. Hoffman, "Differential Fault Analaysis on the SHA 1 Compression Function," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on* , Nara, 2011.

[39] R. Li, C. Li and C. Gong, "Differential Fault Analysis on SHACAL-1," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on* , Lausanne , 2009.

[40] H. Choukri and M. Tunstall, "Round reduction using faults," *Fault Diagnosis and Tolerance*

in Cryptography (FDTC), vol. 5, pp. 13-24, 2005.

[41] A. Shoufan, "A fault attack on a hardware-based implementation of the secure hash algorithm SHA-512," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on* , Cancun , 2013.

[42] K. Jeong , Y. Lee, J. Sung and S. Hong, "Security Analysis of HMAC/NMAC by Using Fault Injection," *Journal of Applied Mathematics*, Vol. 2013, Article ID 101907, 6 pages, 2013.

[43] P. Maistri, "Countermeasures against Fault Attacks: The Good, the Bad, and the Ugly," in *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, Athens, 2011.

[44] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 492-505, 2003.

[45] M. Juliato and C. Gebotys, "SEU-resistant SHA-256 design for security in satellites," in *Signal Processing for Space Communications, 2008. SPSC 2008. 10th International Workshop on*, Rhodes Island, 2008.

[46] M. Juliato, C. Gebotys and R. Elbaz, "Efficient fault tolerant SHA-2 hash functions for space applications," in *Aerospace conference, 2009 IEEE*, Big Sky, MT, 2009.

[47] Altera, "PLLs in Cyclone II Devices," February 2007. [Online]. Available: http://www.altera.com/devices/fpga/cyclone2/features/cy2-pll_features.html.

[48] DE2 Development and Education Board User Manual, Altera Corporation, 2012.

[49] Altera, "My First FPGA Design Tutorial," July 2008. [Online]. Available: http://www.altera.com/literature/tt/tt_my_first_fpga.pdf.

[50] Cyclone II Device Handbook, Altera Corporation .

[51] Nios II Processor Reference Handbook, Altera Corporation , 2011.

APPENDICES

APPENDIX A

The Nios II processor, the Phase locked loop was specified using the Altera **SPOC** builder and the steps are listed below. The following instructions based on the tutorial available in the Altera website [49]

Hardware and Software requirements

- A PC running on Windows XP , Vista or 7 operating systems
- Nios II software development tools version 10.1
- Quartus II software version 10.1

Create Quartus II project

To start the Quartus II software, choose **all programs – Altera – Quartus**

To create project do the following steps

1. Choose New Project wizard (File Menu).
2. Click Next in the introduction.
3. Specify your working directory, name of the project.
4. Click Next, and click yes if you are asked to create the new directory.
5. Click Next.
6. Select Cyclone II in the family drop down menu and in the filter box select FBGA, 672 pins and speed 6. Now choose EP2C35F672C6 from the list of available devices.
7. Click Next.
8. Leave all EDA tools unchecked and click Finish.

Create Nios II System Module

The steps to create the top level block design file (.bdf) is explained below.

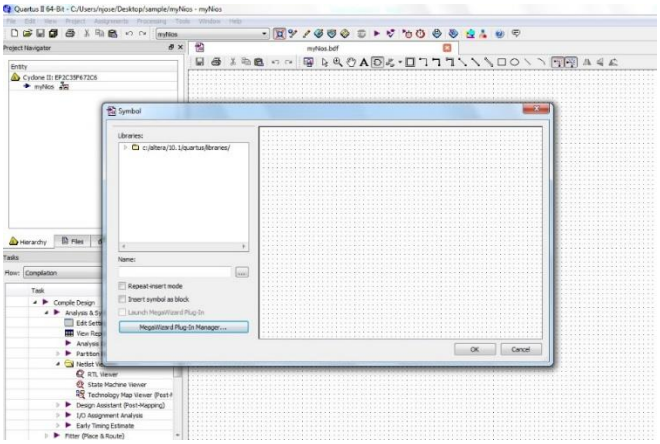
Create a new .bdf

1. Choose New (File Menu).
2. Select Block diagram/Schematic File in the Device Design Files tab.
3. Click OK
4. Choose Save As from File Menu and in the File Name type the name as myNios.
5. Click Save.

SOPC Builder

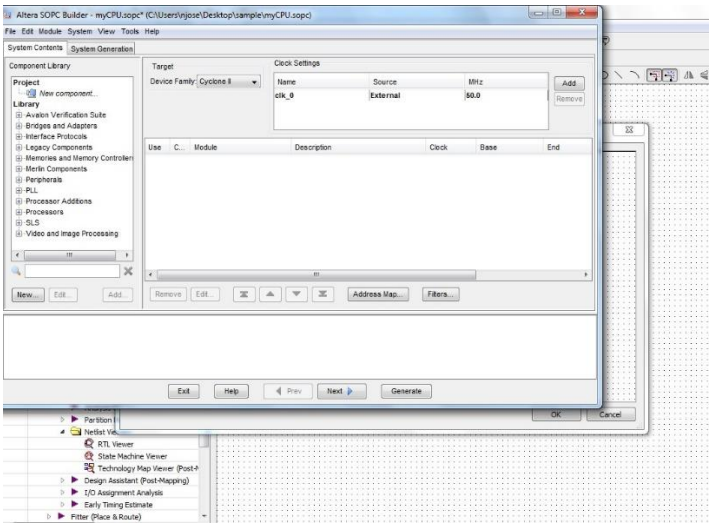
The steps to start SOPC builder are as follows:

1. On the block editor window double click on the empty space. The Symbol dialog box will appear as shown below.



2. Click on Megawizard Plug-In Manager. Select create.
3. Select Create Custom Megafunction variation from **which action do you want to perform**?
4. Under installed Plug-Ins select the Altera SOPC builder.
5. Specify the device family as Cyclone II, output file to be VHDL and give the name as myCPU.

6. Click Next and this will load the Altera SOPC builder and the System Contents tab will be displayed as shown below.



Now the Nios II CPU and peripherals can be added to the system. To add these follow the steps below:

Nios II

To add Nios 32-bit CPU, select Nios II processor under Processors. Click **Add** and the wizard displays. Specify the following options.

- Nios II Selector Guide : Nios II/s
- Hardware Multiply: None
- Hardware Divide: Unchecked

Click Next. The Caches and Tightly Coupled Memories tab will appear.

Specify the following settings:

- Instruction Cache: 4 Kbytes
- Enable Bursts (Burst Size: 32 Bytes): Unchecked
- Include Tightly Coupled Instruction Master Port(s): Unchecked

  Click Finish.

Error messages appear in the SOPC Builder Messages window. These messages are normal and will be fixed later.

System ID

Do the following to add the system ID peripheral:

In the list of available components, select Peripherals and then Debug and Performance and then System ID Peripheral. Click Add and once you return to the SOPC Builder main page, you will need to rename the **sysid_0** to **sysid**.

JTAG UART

The JTAG UART provides a convenient way to communicate character data with the Nios II processor through the USBBlaster download cable. To add the JTAG UART peripheral, **uart_0**, do the following:

Under Communication select JTAG UART and click Add. The Avalon UART – uart_0 wizard displays. Do not change the default settings Click Finish. You are returned to the Altera SOPC Builder window.

External SDRAM Interface

To add the external SDRAM peripheral, sdram_0, perform the following steps:

Select the SDRAM controller from the component library and click Add and the SDRAM controller. In the Presets drop down box select (Custom). Under the Memory Profile tab set the Data Width to 16 bits .All the other settings can be left the default values. Click Finish to return to the Altera SOPC Builder window.

Double click on the cpu_0 name. This will re-open the Nios II Processor Wizard. Select sdram_0 for both of the vectors (Reset and Exception).

This will eliminate the two of the errors we had earlier.

Parallel I/O (pio)

The pios are required to connect the Nios II to the other components like SHA core, PLL etc. To add the pios follow the steps below:

Select the pio from the Component Library and click Add. In the Avalon PIO – pio_0 wizard specify the options such as width based on the requirements.

You will need to add 8 pios with the following specification.

Width: 32 bits, Direction: Input

After adding the pios rename them as pio_0, pio_1...pio_7.

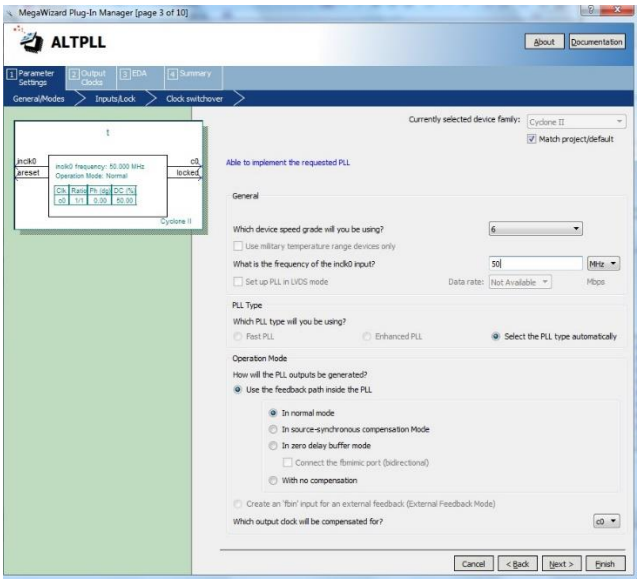
Now add a similar 1 bit input pio and rename it as pio_rdy. Similarly add a 32 bit output pio and rename it as pio_msg.

After adding all of the above components, on the System Menu select Auto-Assign Base Addresses and Auto-Assign IRQs. Now click Generate in the SOPC builder. Save your design when asked by clicking yes. Once the system generation was successful message appears clock Ok to exit. The myCPU symbol attached to the cursor can be dropped on the block editor window and Save it.

Phase Locked Loop

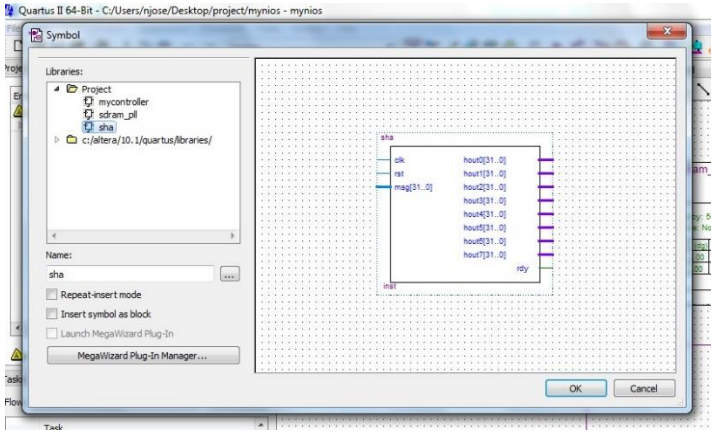
To build the PLL the ALTPLL wizard is to be used. Open the Mega Wizard Plug-in Manager as mentioned earlier. From the I/O folder select ALTPLL, output file as VHDL and name as shaPLL.

On page 3 of the wizard select 50 MHz as shown in the figure below.



On page 4 deselect the boxes on optional input and on page 5 leave the default settings. On page 6 specify the Clock C0 to be 50 MHz, -3ns delay. On page 6 add clock C1 50 MHz and specify you required parameters and similarly on page 7 for clock C2. Leave the default settings as such on page 9 and click Finish. This will return you to the block diagram file and drop the PLL symbol to the .bdf.

Now open a new VHDL file and copy paste the SHA-256 code given in Appendix B. Save the file and add it to the current project. Choose File--Create and Update--Create Symbols for current file. Go to the block diagram window. Double click on empty space and from the Project select SHA and insert the symbol on the .bdf file as shown below.

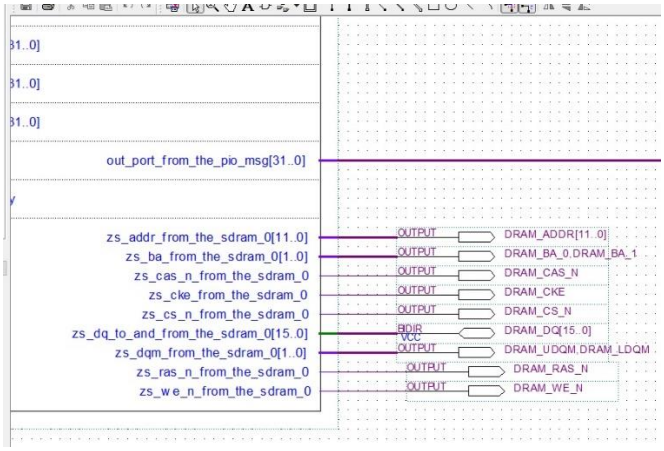


Add Pins & Primitives

The pins can be added as described below.

Click the Symbol Tool button on the Block Editor Toolbar. The Symbol dialog box will appear and in the primitive folder expand the pin folder. Add 3 input pins, 8 output pins and 1 bidirectional pin.

Connect one input pin to the input clock to the PLL (inclk0) and name it as CLOCK_50, another one to rst pin of SHA and name it as SW[17] and the third one to resent_n of myCPU and name it as SW[16]. Connect the output pins and bidirectional pin to the myCPU and rename them as shown in figure below.



The complete circuit and its connection were discussed in the Chapter 4. After connecting the components the pin assignments can be done using the .csv file for the DE2 board provided on the Altera Website. Download this file on to the project folder. Open Import Assignments dialog box under the Assignments Menu on Quartus tool. Browse the .csv file and click OK. Now open the Device dialog box on the Assignments menu. Select Device & Pin Option –Unused Pins— Reserve all unused pins and select as input tri-stated.

Compile Design

Save the .bdf file and choose Start Compilation from the processing menu. After successful compilation the Cyclone II device can be configured. Details on configuring the device can be found on the Altera DE2 handbook.

Testing Nios II system

To test the system a new C/C++ Application has to be created. For this start the Nios II software build tools. Select the workspace which is same as the Quartus project directory. Choose File/New/Nios II Application and BSP from Template to open the new project wizard. Click the ... button beside the SOPC Information File name: text box. The Open window should appear. Browse to the location of your project folder and select .sopcinfo file. Click Open. Name your project and select any template from the list provided (hello word for e.g.). Click Finish. This will crate the application program and the board support package. You can copy paste your program in the .c file on the application folder. Save the project, right click on bsp folder and select Nios II/Generate BSP. Click Project/Build All in the main menu. Once it is successfully compiled you can run the program on the DE2 board by choosing Run As/Nios II hardware.

APPENDIX B

The VHDL design for SHA-256 is given below.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sha is
port (
clk : in std_logic;
rst : in std_logic;
msg : in std_logic_vector(31 downto 0); --input
hout0 : out std_logic_vector(31 downto 0);
hout1 : out std_logic_vector(31 downto 0);
hout2 : out std_logic_vector(31 downto 0);
hout3 : out std_logic_vector(31 downto 0);
hout4 : out std_logic_vector(31 downto 0);
hout5 : out std_logic_vector(31 downto 0);
hout6 : out std_logic_vector(31 downto 0);
hout7 : out std_logic_vector(31 downto 0);


rdy : inout std_logic
);
end entity sha;
architecture rtl of sha is


        --define all functions
        --sigma0(x)
        function e0(x: unsigned(31 downto 0)) return unsigned is
        begin
                return (x(1 downto 0) & x(31 downto 2)) xor (x(12 downto 0) & x(31 downto 13)) xor
(x(21 downto 0) & x(31 downto 22));
        end e0;

        --sigma1(x)
        function e1(x: unsigned(31 downto 0)) return unsigned is
        begin
                return (x(5 downto 0) & x(31 downto 6)) xor (x(10 downto 0) & x(31 downto 11)) xor
(x(24 downto 0) & x(31 downto 25));
        end e1;
```

```
	-- s0 & s1 are functions used in message expansion
	function s0(x: unsigned(31 downto 0)) return unsigned is
		variable y : unsigned(31 downto 0);
	begin
		y(31 downto 29) := x(6 downto 4) xor x(17 downto 15);
		y(28 downto 0) := (x(3 downto 0) & x(31 downto 7)) xor (x(14 downto 0) & x(31 downto
18)) xor x(31 downto 3);
		return y;
	end s0;

	function s1(x: unsigned(31 downto 0)) return unsigned is
		variable y : unsigned(31 downto 0);
	begin
		y(31 downto 22) := x(16 downto 7) xor x(18 downto 9);
		y(21 downto 0) := (x(6 downto 0) & x(31 downto 17)) xor (x(8 downto 0) & x(31 downto
19)) xor x(31 downto 10);
		return y;
	end s1;

	function ch(x: unsigned(31 downto 0); y: unsigned(31 downto 0); z: unsigned(31 downto 0))
return unsigned is
	begin
		return (x and y) xor (not(x) and z);
	end ch;

	function maj(x: unsigned(31 downto 0); y: unsigned(31 downto 0); z: unsigned(31 downto 0))
return unsigned is
	begin
		return (x and y) xor (x and z) xor (y and z);
	end maj;

	function fw(x :unsigned(31 downto 0); y : unsigned(31 downto 0); u: unsigned(31 downto 0); v:
unsigned (31 downto 0)) return unsigned is
	begin
		return (s1(x)+u+s0(y)+v);
		end fw;

	type word_array is array(0 to 63) of unsigned(31 downto 0);
	constant k : word_array := ( X"428a2f98", X"71374491", X"b5c0fbcf", X"e9b5dba5",
X"3956c25b", X"59f111f1", X"923f82a4", X"ab1c5ed5",
				X"d807aa98", X"12835b01", X"243185be", X"550c7dc3", X"72be5d74",
X"80deb1fe", X"9bdc06a7", X"c19bf174",
				X"e49b69c1", X"efbe4786", X"0fc19dc6", X"240ca1cc", X"2de92c6f",
X"4a7484aa", X"5cb0a9dc", X"76f988da",
				X"983e5152", X"a831c66d", X"b00327c8", X"bf597fc7", X"c6e00bf3",
X"d5a79147", X"06ca6351", X"14292967",
				X"27b70a85", X"2e1b2138", X"4d2c6dfc", X"53380d13", X"650a7354",
X"766a0abb", X"81c2c92e", X"92722c85",
```

```
                    X"a2bfe8a1", X"a81a664b", X"c24b8b70", X"c76c51a3", X"d192e819",
X"d6990624", X"f40e3585", X"106aa070",
                    X"19a4c116", X"1e376c08", X"2748774c", X"34b0bcb5", X"391c0cb3",
X"4ed8aa4a", X"5b9cca4f", X"682e6ff3",
                    X"748f82ee", X"78a5636f", X"84c87814", X"8cc70208", X"90befffa",
X"a4506ceb", X"bef9a3f7", X"c67178f2" );


    signal a, b, c, d, e, f, g, h : unsigned (31 downto 0);

    signal w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15 : std_logic_vector(31
downto 0);


begin

    process (clk, rst)

        variable t : integer := 0;

        constant initialh0 : unsigned (31 downto 0) := X"6a09e667";
        constant initialh1 : unsigned (31 downto 0) := X"bb67ae85";
        constant initialh2 : unsigned (31 downto 0) := X"3c6ef372";
        constant initialh3 : unsigned (31 downto 0) := X"a54ff53a";
        constant initialh4 : unsigned (31 downto 0) := X"510e527f";
        constant initialh5 : unsigned (31 downto 0) := X"9b05688c";
        constant initialh6 : unsigned (31 downto 0) := X"1f83d9ab";
        constant initialh7 : unsigned (31 downto 0) := X"5be0cd19";


begin
        if (rst = '0') then
            rdy<='0';
            a<=initialh0;
            b<=initialh1;
            c<=initialh2;
            d<=initialh3;
            e<=initialh4;
            f<=initialh5;
            g<=initialh6;
            h<=initialh7;


        elsif(rising_edge(clk) and rdy='0') then
            if (t < 64 ) then
            if (t < 16) then

                w15<=msg;
```

```
                    e<=d + h + e1(e)+ ch(e,f,g) + unsigned(msg)+k(t);
                    a<=h + e1(e)+ ch(e,f,g) + unsigned(msg)+k(t)+ e0(a) + maj(a,b,c);

                    else
                    w15<=
std_logic_vector(fw(unsigned(w14),unsigned(w1),unsigned(w9),unsigned(w0)));
                    e<=d + h + e1(e)+ ch(e,f,g) +
fw(unsigned(w14),unsigned(w1),unsigned(w9),unsigned(w0)) +k(t);
                    a<=h + e1(e)+ ch(e,f,g) +
fw(unsigned(w14),unsigned(w1),unsigned(w9),unsigned(w0)) +k(t)+ e0(a) + maj(a,b,c);
                    end if;
                        w0<=w1;
                        w1<=w2;
                        w2<=w3;
                        w3<=w4;
                        w4<=w5;
                        w5<=w6;
                        w6<=w7;
                        w7<=w8;
                        w8<=w9;
                        w9<=w10;
                        w10<=w11;
                        w11<=w12;
                        w12<=w13;
                        w13<=w14;
                        w14<=w15;

                        h<=g;
                        g<=f;
                        f<=e;
                        d<=c;
                        c<=b;
                        b<=a;


                        t := t + 1;


                else


                        hout0 <= std_logic_vector(a + initialh0);
                        hout1 <= std_logic_vector(b + initialh1);
                        hout2 <= std_logic_vector(c + initialh2);
                        hout3 <= std_logic_vector(d + initialh3);
                        hout4 <= std_logic_vector(e + initialh4);
                        hout5 <= std_logic_vector(f + initialh5);
```

```
                                    hout6 <= std_logic_vector(g + initialh6);
                                    hout7 <= std_logic_vector(h + initialh7);


                                   rdy <= '1';

                            end if;
                        end if;
                end process;
end rtl;
```