FPGA Security – Bitstream Authentication

Milind M. Parelkar, George Mason University, Fairfax, VA

***Abstract*—Safeguarding Intellectual Property on FPGAs is a major challenge for FPGA manufacturers. The challenge stems not from the fact that it is difficult to add security features to the FPGA, but from the commercial point of view. The main question is whether the entire user base will be ready to pay for these added features. Also, there is no consensus among various manufactures as to which security features are absolutely essential for FPGA security. This paper addresses some security scenarios in FPGAs, and tries to point out why currently existing security features are inadequate. The concept of bit stream authentication is introduced and different authentication options are compared. Finally a comparative analysis of hardware implementations of the authentication algorithms is provided for FPGA as well as ASIC implementations.**

***Index Terms*—Bitstream Authentication, FPGA Security, HMAC, Secure Hash Algorithm**

1. Introduction

Over the past few years there have been growing concerns over the security of FPGA designs. Is it possible to steal Intellectual Property (IP) from the FPGA? Can someone reverse engineer or even clone a design should someone capture the corresponding bitstream [7]? These are all valid concerns, and most FPGA manufacturers have incorporated some security features on their FPGAs. Xilinx pioneered with a Triple DES decryption engine on the Virtex II Pro family of FPGAs. The most recent Xilinx family of FPGAs, Virtex-4 uses Advanced Encryption Standard (AES) with a 256-bit key for bit-stream encryption. This key is stored in a battery-backed dedicated RAM. This takes care of some security issues, but there are quite a few issues which cannot be handled only by confidentiality of the bitstream. This paper lists out various factors which have prompted FPGA manufacturers and the research community to look at factors other than bit-stream encryption, especially bit-stream authentication and bit-stream integrity.

1.1 Overview of FPGA Security Issues

This research project was undertaken for Xilinx Inc, CA and continued as a semester-long project for ECE 646 (Cryptography and Network Security) at George Mason University, Fairfax, VA.

A survey of various attack scenarios on FPGAs provides an insight into the need to incorporate security features on FPGAs [12]. Concentrating mainly on remotely reconfigurable FPGAs (since they are most susceptible to spoofing, hacking etc.) let us look at a few attack scenarios in which a design on an FPGA could be compromised.

The simplest passive attack on an FPGA design is eavesdropping. When an unencrypted bitstream is being transmitted to a remote FPGA, the attacker can simply read the bitstream if he has access to the link between the sender and the receiving FPGA. This attack scenario is shown in Fig. 1.

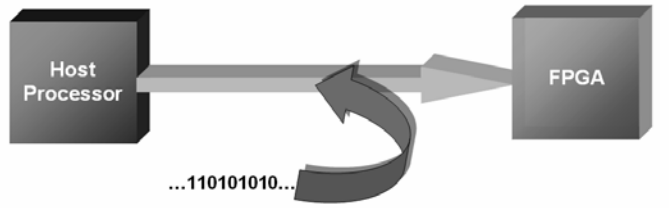

Figure 1: Attack Scenario on FPGAs (Eavesdropping)

If an attacker can get his hands on a complete unencrypted bitstream, cloning the design would be a trivial task, although reverse-engineering might not be so straightforward. Most FPGAs today are equipped to thwart this kind of passive security threat [7]. The solution is to encrypt the bitstream before transmitting it to the remote FPGA. This encryption functionality is an integral part of the vendor-specific tools available for the FPGA designing process. The FPGA has an on-chip decryption engine which allows it to decrypt the incoming bitstream. The keys used for encryption/decryption can be chosen by the user. If a vendor does not provide the ability for a user to select his own keys, beware!! The key is stored on the FPGA in a dedicated battery-backed RAM. Read access to this RAM is disabled once the key is validated and stored into the RAM. The pros and cons of using a battery backed RAM to store decryption keys is a fiercely debated topic. Any attempt to read the contents of the keys on the FPGA causes the contents of the FPGA to be erased. Generally the remote FPGAs are enclosed in some kind of a tamper-proof casing which prevents any physical spoofing attacks. A concern voiced

by some FPGA users is that a tamper proof enclosing still doesn't prevent an attacker from damaging the FPGA. Physically damaging an FPGA (hitting it with a hammer!!!) would possibly lead to Denial of Service (DoS), but that is preferable to the loss of IP in almost all cases.

Passive attacks can be effectively thwarted by bit-stream encryption as shown in Fig. 2. A class of active attacks exist which cannot be handled using bitstream encryption alone. This is where bitstream authentication comes into focus.

The most important thing to note here is that encryption doesn't necessarily provide authentication [1]. A tampered or a fabricated bitstream would decrypt to gibberish. Although this decrypted gibberish would not implement any functionality on the FPGA, it has the potential to damage the FPGA. Damage to the FPGA can result from increased power consumption, swapping input/output modes of I/O pins etc. Some research papers on FPGA Viruses describe particular bitstream combinations which might be potentially harmful to the FPGA [8].

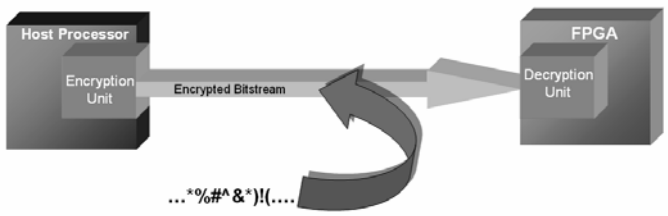

Figure 2: FPGA Bitstream Encryption

In an ideal security scenario, the FPGA should accept only bitstreams from an authenticated source. This would prevent an active attacker from destroying the FPGA remotely using certain malicious bit-stream combinations. If the IP which is currently present on the FPGA is assumed to be the authentic one, then this authentic IP could be used to authenticate all future incoming bit-streams. In order to incorporate this feature, the FPGA would need to have an authentication engine on the chip. Various implementations of such engines have been suggested. One possible implementation would be to have the engine in the fabric of the FPGA itself, a concept similar to the currently existing decryption engines. Other suggested implementations include incorporating the authentication engine on FPGA resources (like Configurable Logic Blocks, Block RAMs etc.) or using a soft-core microprocessor (like Xilinx MicroBlaze) to implement the engine.

Implementation of authentication engine using FPGA resources would allow users to add authentication functionality to older families of FPGAs, with obvious trade-offs. Implementation using a soft-core microprocessor (or even an embedded processor like PowerPC) on an FPGA would be an interesting option as it would require fixed resources for implementation of any authentication algorithm, but with a possible speed trade-off. In this paper, we will look at implementation aspects for an ASIC as well as an FPGA implementation of authentication engine.

2. Bitstream Authentication – An overview

Authentication involves some kind of digital signature or a hash function which can prove the authenticity of the source of a particular message [1]. A hash function by itself does not provide complete authentication, since it does not have a secret key associated with it[1][2]. As long as there is no secret parameter between the sender and the receiver, complete authentication is impossible. Hence, a hash function is always used along with a Message Authentication Code, like an HMAC, in which the message digest is a function of the hash function as well as a secret key.

Let us look at a scenario in which only a hash function is used to sign a bit-stream. The hash of the bit-stream would be calculated and appended to the bit-stream before it is transmitted to the remote FPGA. The authentication engine on the FPGA would calculate the hash of the incoming bit-stream and compare it to the hash value appended to the bit-stream. The goal would be to preserve message integrity and also authenticate the source of the bit-stream. It is quite straightforward to see that this scheme would not work very well. It serves to preserve message integrity, but does not really prevent an attacker from putting his functionality on the FPGA. The attacker could simply fabricate a message, compute its hash and transmit it. The FPGA would accept it as being authentic, since the hash value would be valid. Here, the attacker only needs to know the algorithm used for hashing. It is generally not considered feasible to keep the algorithm secret. Fig. 3 shows the inherent flaw in such a scheme.

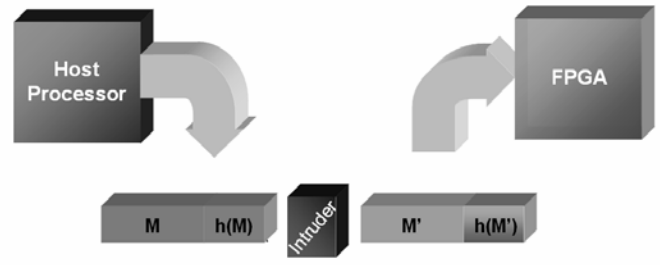

Figure 3: Standalone Hash Functions for Authentication - A Security Flaw

The solution to the attack described above is to use a MAC instead of a simple hash function. Message digest in the case of a MAC is a function of the message as well as a secret key. The secret key is also stored on the FPGA. The authentication engine computes the MAC of the incoming bit-stream and verifies the signature. In this scheme, an attacker cannot fabricate a message and expect it to be authenticated, since he cannot compute the proper MAC without knowledge of the secret key.

3. Authentication Techniques

This section discusses various authentication techniques available and provides a detailed explanation of the options implemented in the course of this project. It would be useful to mention the available options, so as to facilitate a comparative analysis of various techniques. There are a variety of authentication algorithms approved by NIST which are worth comparing with regards to performance.

1. HASH Functions - Secure Hash Algorithms (SHA) – SHA-1, SHA-256, SHA-384, SHA-512
2. Message Authentication Codes (MACs)
3. Message Digest Algorithms (MD5 etc.)
4. AES – OCB (Offset Code Book)

This paper focuses on the first 2 options mentioned in the above list. A brief mention of possible implementation issues with remaining options seems appropriate at this point. Message Digest Algorithms such as MD5 are designed to be fast in software. Hardware implementation of MD5 is extremely inefficient in terms of both timing as well as circuit area. AES-OCB is quite appealing. Simple analysis is presented to reinforce the appeal presented by the OCB mode. As mentioned earlier, a large number of recently developed FPGA families which have the capability for bitstream encryption use 3DES or AES. It would be a fair assumption to say that in the near future, all manufacturers would prefer replacing the existing encryption algorithms with AES. AES-OCB builds around the AES block cipher. Since, the block cipher engine would already be present on the FPGA; it would be extremely efficient to just add the OCB wrapper to it. The only argument against the use of OCB is that it is patented and that NIST has not yet approved it as an authentication standard.

3.1 Hash Functions

Hash function is a compression function which takes an input of arbitrary length and compresses it into a message digest or a fingerprint of a fixed length [1][2]. In this paper, the words 'message digest' and 'fingerprint' are used interchangeably. Another important property is that it is a one-way function. See Fig. 4 below. HASH functions are also referred to as 'trap-door functions' because of this property. One-way property implies that it is computationally easy to calculate the message digest from the original message, but to do the reverse procedure is computationally infeasible. The one way property implies that it is computationally infeasible to fine the original message from the message digest [2].

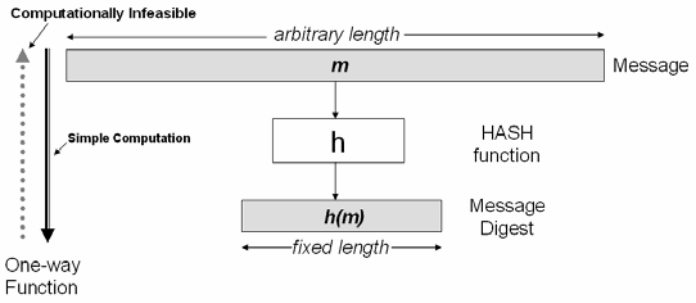

Figure 4: One-way Property of Hash Functions

FIPS 180-2 standard [5] specifies four secure hash algorithms, also known as SHA functions. These are SHA-1, SHA-256, SHA-384 and SHA-512. All the four functions are iterative hash functions which produce a compressed fingerprint from the original message. These functions enable the determination of message integrity since any change in the message, will with a very high probability, produce a different message digest. This property is useful in the generation of Message Authentication Codes (MACs). Secure Hash Algorithms (SHA-1 and SHA-512) have been used to implement HMACs.

3.2 Message Authentication Codes (MACs)

Providing a way to check integrity of information transmitted over an unreliable medium is the prime necessity in the world of open networking and computing. Mechanisms which provide such integrity checks based on a secret key are called Message Authentication Codes (MACs). The FIPS 198 standard [6] defines a MAC which uses a secret key in conjunction with an approved hash function. This generalized mechanism is called an HMAC (Keyed-Hash Message Authentication Code).

HMACs are similar to hash functions in the sense that both are compression functions. The main difference between HMACs and hash functions is the use of a secret key. In case of hash functions, the output is a function of only the message, whereas in case of HMACs, the output is a function of both the message as well as a secret key. The output of these compression functions is referred to as message digest or fingerprint. The key limits the authentication process somewhat in the sense that only a receiver with the knowledge of the secret key can verify the hash. But, the presence of the

key also provides a form of non-repudiation, since only a person in possession of the secret key could have produced the fingerprint in the first place.

3.2.1 Cryptographic Keys used with HMACs

The FIPS 198 standard specifies the following criteria with respect to the size of the key for HMACs – "*The size of the key, K, shall be equal to or greater than L/2, where L is the size of the hash function output. Note that keys greater than L bytes do not significantly increase the function strength. Applications that use keys longer than B-bytes shall first hash the key using H and then use the resultant L-byte string as the HMAC key, K.*" [6] This paper focuses on an implementation of HMAC with SHA-512 used as the hash function. The length of the message digest produced by SHA-512 is 512 bits. Hence, the smallest size of key approved according to the FIPS 198 standard for the implementation under consideration is 256 bits.

3.2.2. Computations in HMACs

Mathematically computations in HMAC can be expressed as

$$MAC(text) = HMAC(K,text)$$
$$= H((K_0 \oplus opad)\,||\,H((K_0 \oplus ipad)||text))$$

In the above equation *H(x)* denotes the hash operation on message *x*. *ipad* and *opad* are constant padding stings. The acronyms stand for 'input pad' and 'output pad' respectively. The length of these padding strings is equal to the length of the derived key K_0. The value of *ipad* is repetitions 0x36 while that of *opad* is repetitions of 0x5A. K_0 is derived from the user key K by computing the hash of the user key K. Compression of the key is done only when the length of the key supplied by the user is larger than the size of the key which can be handled by the HMAC. For all other cases, K_0 is the same as K.

The dataflow diagram for HMAC is shown in the Fig. 6. As can be seen from the diagram, the hash operation is performed twice. The initial hash operation uses the concatenation of message and the key K_0 xored with *ipad* as input. The hash operation produces a message digest of a fixed length depending upon the hash function used. In the case of SHA-512, a 512-bit hash value is generated. This hash value is then used as the input to the next hash operation after concatenation with $K_0 \oplus opad$. An important aspect to note here is that the first hash operation takes an arbitrary length message as input, but the length of the input to the second hashing operation is constant for a given hash function and length of the key. This observation is the basis of some of the implementation aspects of HMACs.

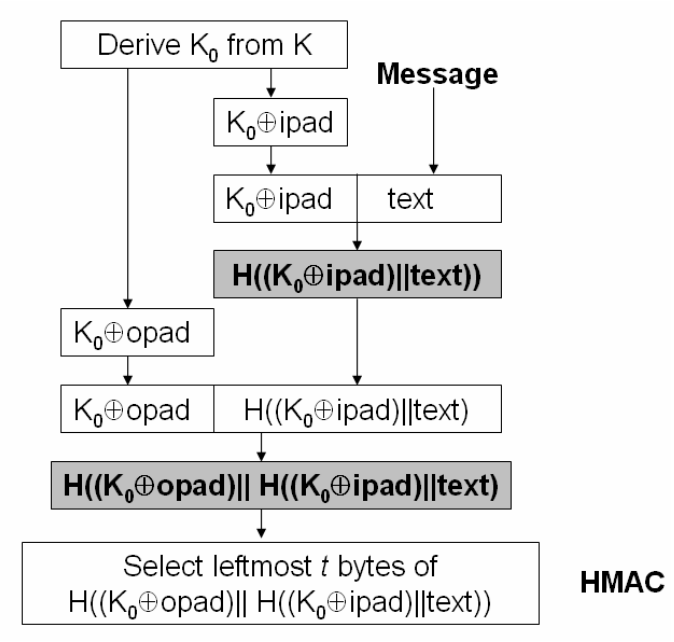

Figure 5: Dataflow Diagram for HMAC

4. Hardware Implementation Issues

The basis for this research project was a continuation of the work started by Tim Grembowski during his Master's Thesis at GMU[3]. His implementation of SHA-512 was used as the starting point. The VHDL implementation was platform specific in the sense that it targeted Xilinx VirtexII FPGAs. Some pre-defined components from the Xilinx UNISIM library were used in order to optimize hardware for the target family. The code was fully revised to further optimize it for an FPGA implementation and it was also made platform independent. Details about code revisions are given in the section for SHA-512 Implementation.

The first undertaking was to build a HMAC wrapper around the revised code for SHA-512. The other task was to implement a different hash function (SHA-1) which could replace the existing SHA-512 core inside the same HMAC wrapper. At this juncture, it is important to specify the factors which influenced the design of the other hash function.

The broader issue is the implementation of HMACs. Secure Hash Algorithms are just sub-entities inside the top-level HMAC entity. The structure of the HMAC remains the same irrespective of the hash function used. Hence, the idea here is to create a general top-level HMAC which can support any Secure Hash Algorithm. In order to realize this, it is important, that the Secure Hash Algorithms should be exactly same when viewed as black-boxes i.e. they should have similar port configurations. Hence, the main design criterion in developing the SHA-1 core was to match the design functionality of SHA-512. This task was a little more involved since it meant ensuring pin compatibility of the

new design with the already existing design in addition to ensuring similar functionality.

A couple of alternative designs of SHA-1 were implemented – one targeting Xilinx VirtexII FPGAs and the other platform independent. Synthesis results for all designs were obtained for both FPGA as well as ASIC implementations.

4.1 FPGA Implementation

An overview of the tools used for synthesis and implementation is given here.

- VHDL Design Entry/Compiler: Aldec Active HDL v6.2
- Synthesis Tool: Synplicity Synplify Pro 7.6.1, Xilinx XST v6.1
- Implementation Tool: Xilinx ISE v6.1

The main design criterion was implementation with minimum circuit area. Since, bit-stream authentication is a kind of an added service for security a large overhead with regards to circuit area would not be favorable. Devices from the Xilinx Virtex II family were targeted. As far as possible, implementations were performed targeting the smallest device which could accommodate the designs. It is non-trivial to note the importance of the latter design criterion. Larger FPGAs with more logic as well as routing resources invariably give better performance results than smaller devices. An example would be helpful in understanding this phenomenon.

Assume that a certain design occupies about 90% of logic resources on a particular FPGA. It is a safe assumption that it would occupy about the same percentage of routing resources on the FPGA. An analysis of routing resources on the FPGA reveals that there are a comparatively small percentage of fast nets. Critical signals, if constrained properly generally use these fast nets. There will always be a case wherein, a critical signal might not find a free fast net. This affects timing adversely. There is also routing congestion to take into consideration. Routing congestion occurs if there are a large number of interconnects between CLBs. Interconnects between adjacent CLBs can be handled efficiently, but if the interconnects are long and run across the length of the chip, then it might lead to localized routing congestions. In the most extreme cases, the design might not be completely routed. It follows from the above analysis that if designs occupy only a small fraction of the chip, then it gives much better performance estimates.

4.2 ASIC Synthesis

For ASIC Synthesis, Synopsys Design Compiler was used. The GUI mode of Design Compiler (also known as Design Analyzer) was the primary tool interface which was used. Here it is important to note that, due to unavailability of some DesignWare licenses, memories (RAM, ROM) could not be synthesized. Hence, all ASIC Synthesis results for area are without the use of memories. Timing results can be safely assumed to be accurate, since memories are not in the critical path in any of the designs. Hence, absence of memories should not affect the timing results in any way.

Baseline results for 90nm as well as 130nm technology are provided in this paper. Synthesis was performed using NCCOM (Normal Case Commercial) operating conditions [14]. A wireload model was used for estimates about interconnect delays. The concept of wireload model is beyond the scope of this paper.

5. Implementation of Secure Hash Algorithm–SHA-1

All Secure Hash Algorithms have similar structure and hence their implementation is also similar in most aspects. In this paper, the design methodology for SHA-1 is presented. Secure Hash Algorithms can be divided into 2 main modules – Message Scheduler and Message Digest Unit. Implementation aspects of both these functional units are described.

5.1 Implementation of Message Scheduler

Message Scheduler takes as input the incoming message stream, performs necessary computations and outputs a single word W_t per clock cycle. This word is used by the message digest unit to compute the hash value. The diagram for message scheduler unit of SHA-1 is shown in Fig. 7. The data path is 32 bits wide, and the unit consists of 16 registers. Hence, the block size which can be handled by the algorithm is 32*16=512 bits. The multiplexer selects between the incoming message words and the words computed by the message scheduler unit. For the first 16 rounds, the multiplexer selects the message word as the input to the register bank. In the remaining rounds, the internally generated words are fed back to the register bank. Accordingly proper control signal is generated for the multiplexer by the control unit depending upon the round number. The generation of the feedback word uses an XOR gaterotation is realized using fixed connections (i.e. rotation operation does not require any combinational logic to implement in hardware).

An interesting optimization is possible in the message scheduler unit for an FPGA implementation targeting Xilinx devices [3] [4]. The register bank in Fig. 7 can be viewed as a series of four shift registers as shown in the right hand portion of the same figure. Instead of using Flip-flops to implement registers; the FPGA implementation uses Lookup Tables (LUTs) to implement the shift register. Xilinx UNISIM library includes an interesting implementation of a variable length shift register (SRL16) using LUTs.

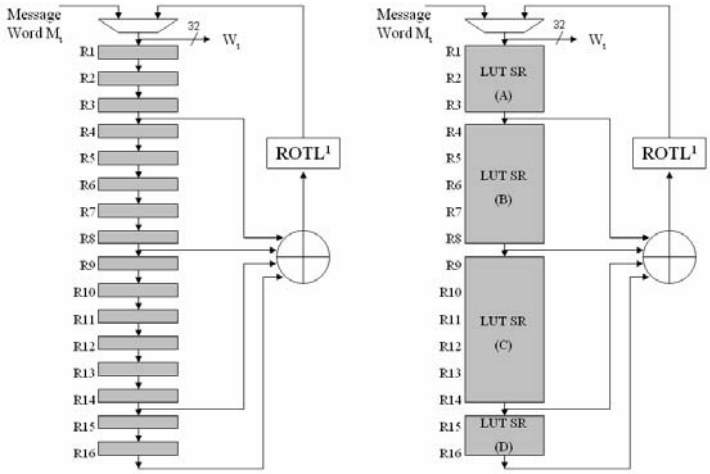

Figure 6: Message Scheduler Unit for SHA-1

The variable length concept is useful in this implementation, since all the four shift registers used have different lengths. Comparative analysis of an FPGA implementation using SRL16s and traditional FFs reveals the disparity between the two approaches. It is worthwhile to mention a few concepts of FPGA architecture in order to appreciate the optimality of implementation using SRL16s.

Logic elements are placed on FPGAs inside Configurable Logic Blocks(CLBs) [15]. Each CLB is further broken down into multiple CLB Slices. Generally there are 2 or 4 CLB Slices per CLB. Programmable routing inside CLBs is much faster as compared to routing between different CLBs. Each CLB Slice includes 2 LUTs and 2 FFs. This implies that the maximum number of FFs inside a CLB, considering 4 CLB slices per CLB, is 4*2=8. So, only 8 FFs can be connected by dedicated fast routing resources. If more than 8 FFs have to be interconnected, then they have to be connected using inter-CLB routing. Although there are some fast routing resources for interconnections between CLBs, these are generally used by critical signals. For the register bank shown in figure, we would need 512 FFs. This would lead to a large number of programmable interconnects, thereby increasing the routing delay.

A large routing delay would lead to a longer critical path and reduced clock frequency. Using SRL16s helps because, the implementation tool handles interconnects between SRL16 blocks using faster routing resources. Also, there are a less number of nets to route for the synthesis tool. Hence, overall timing efficiency is increased due to this optimization.

ASIC implementation of this module uses traditional shift registers. Since, the HDL code for registers contains generics; this is handled differently by Synopsys. Specific commands were used to force Design Compiler to read the code containing generics and save it as a template [13]. This template is later pulled up by the tool during the linking phase and proper values of generics are substituted in the template.

5.2 Implementation of Message Digest Unit

The main components of the Message Digest Unit are the 5 working registers, *a, b, c, d* and *e*. Each of these registers is 32 bits wide. The final value of these registers is the message digest. Hence, in case of SHA-1, the size of message digest is 5*32=160 bits. In every round, the contents of the working registers are updated. The computations performed in each round are explained in [5]. A brief overview of implementation aspects for FPGAs is presented here.

The rotation left operation (ROTL) is performed using fixed connections. The round dependent logic function $f_t(b,c,d)$ uses 3 different combinational logic blocks, Ch, Maj, and Parity, and a 4-to-1 multiplexer to select the output of a particular block depending on the round number. W_t is the output of the Message Scheduler Unit. K_t is a round dependent constant. K_t takes one of 4 possible values depending upon the round number. A simple implementation is to hardwire these constant values to the inputs of a 4-to-1 multiplexer, and use a bit-slice of the round counter output to select a particular constant.

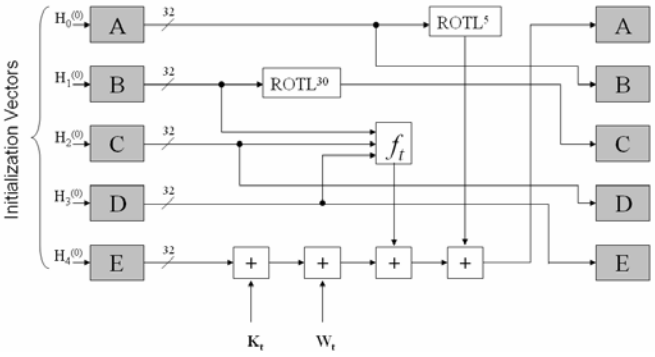

Figure 7: Message Digest Unit for SHA-1

All the operations in the message digest unit mentioned so far do not provide much scope for optimization. The only components which can be optimized to some extent are the adders used in the module. Let us first consider the string of adders connected to register *e* (see Fig. 8). Intuitively, it can be seen that the critical path of the circuit is the one which covers all the adders. Minimizing the length of the critical path is an important design criterion. In this case, there are 5 operands to be added. Various implementations of adders were studied [9], and a comparative analysis influenced the decision to use ripple carry adders. An interesting alternative is to

use a carry save adder tree in order to reduce the number of operands to be added from 5 to 3. 3 operands can then be reduced to 2 by using another carry save adder. To add 2 operands, some kind of fast adder like Brent-Kung or Kogge-Stone could be used. Implementation results show that there is not much gain by using this scheme. Also,
the use of fast adders to improve timing results in an increase in circuit area. Although, timing is critical in this application, it is more important that the circuit be as small as possible. Hence, in order to minimize area, ripple carry adders were used.

In case of an FPGA implementation, the use of ripple carry adders provides an added advantage. It may again be worthwhile to note that there are dedicated FPGA resources for performing fast additions [15]. These include carry chain logic and a dedicated AND gate per CLB slice for performing fast additions. The synthesis tool can recognize certain VHDL constructs as adders, and it uses carry chain logic in order to speed up the ripple addition. This ultimately leads to a tolerable delay of the critical path, albeit a little more than the delay when carry-save adders are used. On the other hand, the circuit area required for ripple carry adders is much less than that required by the scheme using carry save adders.

Another interesting design consideration was the placement of adders required to find the final hash [4]. Please refer to the last set of equations in the hash computation round in [5]. This placement provides a trade-off between number of clock cycles required to compute the output and the period of each clock cycle. If these adders are placed outside the loop, then they are used for addition after 80 rounds have been completed. In this case, the entire iteration of computing the hash takes 81 clock cycles. The number of clock cycles can be reduced to 80 if the adders are placed inside the loop. A control circuit is required with forces one of the inputs of the adders to a zero for all but the last cycle. The length of the critical path increases only slightly in this case. The control circuit uses up a little more area than the first option. Both the options are comparable in terms of performance, and it was arbitrarily chosen to use the adders outside the loop.

5.3 Implementation of other units in SHA-1

Other major units of SHA-1 are the control unit and the preprocessor. The preprocessor was implemented so as to match the features of this implementation with the already available implementation for SHA-512. The input message is passed to the preprocessor, which takes care of proper padding and length field appending. The design of the preprocessor is such that it expects the first word after reset to be the length of the message to be processed. Using this length field, the preprocessor's control unit produces appropriate control signals for internal registers and multiplexers. The implementation of the preprocessor was borrowed from the SHA-512 implementation. Implementation details are explained in [3].

Control unit is implemented as a Finite State Machine. Almost all components of the FSM are implemented as a Mealy machine. For an FPGA implementation, one-hot state encoding was used. General experience shows that one-hot encoding results in a much lesser combinational logic area, since a decoding circuit is not required. Since there are a large number of FFs available on FPGAs, one-hot encoding is favorable. For an ASIC implementation, the encoding scheme could not be forced on Synopsys Design Compiler.

6. Implementation of Secure Hash Algorithm – SHA-512

As stated earlier, already available VHDL code for SHA-512 was used as a starting point in this implementation. The available code was platform specific – targeting Xilinx VirtexII family of FPGAs[3]. In this project two kinds of revisions were made using the existing code. The first revision included further optimization of the VHDL code in order to improve performance. Some portions of the code were written at a very low level of abstraction i.e. using components from FPGA vendor specific libraries. This is an oft used programming practice to describe very small portions of circuits because it allows the programmer to better constrain critical portions of the design. An example here would make the explanation more clear.

Let's say, the circuit is a 2 bit counter with an output for terminal count. It would be favorable if the FFs are placed in the same CLB Slice and the dedicated AND gate is used to detect the condition that the counter has reached "11". It might be useful if the counter is structurally described in HDL using library components which can later be constrained during floor planning. (Floor planning in case of FPGAs deals with the placement of certain logic components in particular locations on the chip). Although it is clearly advantageous in case of small circuits to describe them at a low-level and floor plan them, it is not such a good idea to use the same concept for larger circuits. Floor planning is a tedious and a complex process and might adversely affect the final outcome if not done correctly. Hence, it is always recommended that the HDL code be written at the RTL level. This is the level of abstraction which can be well understood by all synthesis tools.

Also, when the code is written at a higher level of abstraction, there is still some chance for the synthesis tool to perform optimizations like resource sharing, pruning unwanted logic, replicating driver logic for high

fan-out nets etc. If everything is explicitly specified by the user, then there is very little the synthesis tool can do to optimize the design. Hence, portions of the code which were written at a lower level of abstraction were revised to RTL level. Also, control logic was simplified a little in order to reduce combinational logic area by a very small margin.

The second code revision was necessary for an ASIC implementation of the circuit. Using components from vendor libraries makes the code unsuitable for transporting to other environments. Hence, all library specific components were replaced using platform independent coding technique. Shift registers implemented using LUTs were revised to the traditional form. All other components like FFs etc. were written using behavioral RTL code. It is most interesting to note that when the platform independent code was synthesized using FPGA synthesis tool (Synplify Pro) it was intelligent enough to infer the original components from the behavioral code. The only components which were different from the original code were the shift registers. The RTL netlist showed them as being implemented using FFs rather than LUTs.

7. Implementation of HMAC with SHA-512

The top level entity for HMAC SHA-512 is shown in Fig. 9. The table describes the functions and bus widths of all the ports.

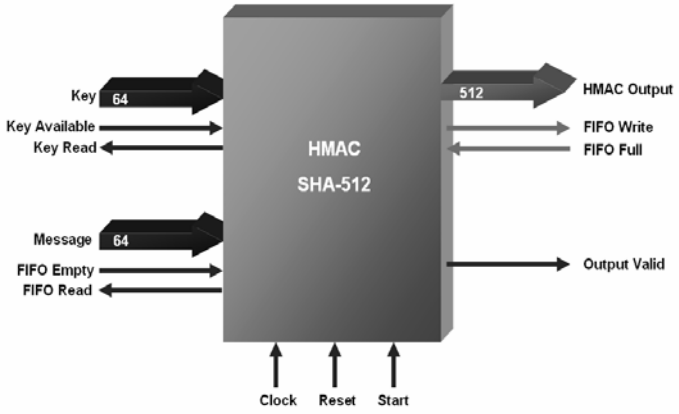

Figure 8: HMAC with SHA-512 - Top Level Entity

Port Name	Width	Mode	Function
Key	64	IN	Secret Key is input through this port 64 bits at a time. Key sizes are limited to multiples of 64 bits.
Key Available	1	IN	Signal is asserted when Secret key is available for reading from the Input Key FIFO.
Key Read	1	OUT	Signal is asserted when the HMAC core reads 64 bit portion of the Secret Key.
Message	64	IN	Message whose MAC is to be computed is input through this port in blocks of 64 bits.
FIFO Empty	1	IN	Signal is asserted when the Input FIFO is empty and does not have any new data to input to the HMAC.
FIFO Read	1	OUT	Signal is asserted when the HMAC core reads a 64 bit block of message.
Clock	1	IN	Master Clock signal to the HMAC core.
Reset	1	IN	Master Reset signal to the HMAC core.
Start	1	IN	This signal is set to HIGH in order to start the MAC computation. It should be asserted only after the secret key has been loaded into the core.
HMAC Output	512	OUT	HMAC computed from the message is output on this port.
Output Valid	1	OUT	Signal indicates that the HMAC Output available is valid when asserted.
FIFO Write	1	OUT	Signal is asserted when the HMAC Output is valid and Output FIFO is ready to accept data.
FIFO Full	1	IN	Signal is asserted when Output FIFO is full and cannot accept further data.

Table 1: HMAC-SHA-512 - Pin Functions

A block diagram of the internal features of the top-level entity is as shown in Fig. 10. In this case, the HMAC-SHA-512 has been designed to support a key size of 256 bits. It can be extended to support sizes of 512 and 1024 bits with minor changes in the HDL code. The 256 bit key is read in terms of 64 bit words. This is done in order to minimize the number of I/O pins on the top-level entity. Minimization of I/O pins is useful, especially in the case of FPGA implementations. A study of available FPGA packages reveals that only FPGAs

with large areas (i.e. large amount of resources) have more number of I/O pins. It would be a waste of FPGA resources to utilize a larger chip just to accommodate a large number of I/O pins. Also, cost of these devices is proportional to the size, and hence cost would also increase for larger devices.

The key is internally stored in a 256 bit register. 4 clock cycles are required to fill the 256 bit register, since only 64 bits are read at a time. A modulo 4 counter is used to enable one out of 4 register slices during each clock cycle. After the key is read into the core, the HMAC Calculation begins only when the START input is asserted. As long as START is de-asserted, the HMAC operation is stalled. Padding signals are generated by XORing the 256-bit key with the 256-bit value of *ipad* and *opad*. *ipad* and *opad* signals are repetitions of the string 0x5C and 0x36 respectively. Both these operations are performed in parallel.

After the padding operation is performed, there is a particular order in which the data words should be input to the SHA-512 core. Here, it is worthwhile to look at the way in which the SHA-512 core accepts data. The SHA-512 core is designed to accept input data in terms of 64-bit words. The first 64-bit word accepted by the SHA-512 core after reset indicates the length of the message. This data word triggers internal control circuit to generate proper control signals to perform message padding without user intervention. The method of padding messages in Secure Hash Algorithms is described in [5].

Since, the SHA-512 core is expecting the length of the message as the first word, the HMAC wrapper should be designed to provide the length to the SHA-512 core. It follows from the earlier discussion about HMACs that the hashing operation has to be performed twice. The first hashing operation is performed on the concatenation of the incoming message and *ipad* xored with the key K_0, whereas the second hashing operation is performed on the concatenation of the hash produced by the first operation and *opad* xored with the key K_0. Hence, for the first operation, the length of the message to be hashed is equal to sum of the length of the incoming message and the length of *ipad*. As stated earlier, *ipad* has a length of 256 bits or 32 bytes, because of the 256-bit key. Hence, the first word sent to the SHA-512 core is the length of the message in bytes plus 32. In the second case, the length of the message to be hashed is equal to the 512 bits (64 bytes) of the first hash operation plus 256 bits (32 bytes) of padding due to *opad*. After the length has been input to the SHA-512 core, the padding strings are the next in line. Since, the padding strings are 256 bits wide and the input data path of the SHA-512 core is only 64 bits wide, it takes the next 4 clock cycles to input the padding string into the SHA-512 core. After the padding string, message is sent to the SHA-512 core. This is done using the multiplexers shown in the figure. Specifically note the order of inputs for the multiplexers. The select lines for the multiplexers are outputs of separate counters. Let's look at the multiplexer to which the combination of *ipad* and key is connected. The length is input through the input 0 (the topmost input) of the multiplexer. Inputs 1 through 4 are used to input the padding string. Input 5 is used to input the message. The counter which controls the select lines of this multiplexer is designed as an up-counter which counts from 0 to 5 and then stalls till it is reset again. Differences in the design of the second multiplexer-counter module are trivial. There is another multiplexer in the input data-path to the SHA-512 core which selects the proper message stream depending on whether it is the first hash operation or the second.

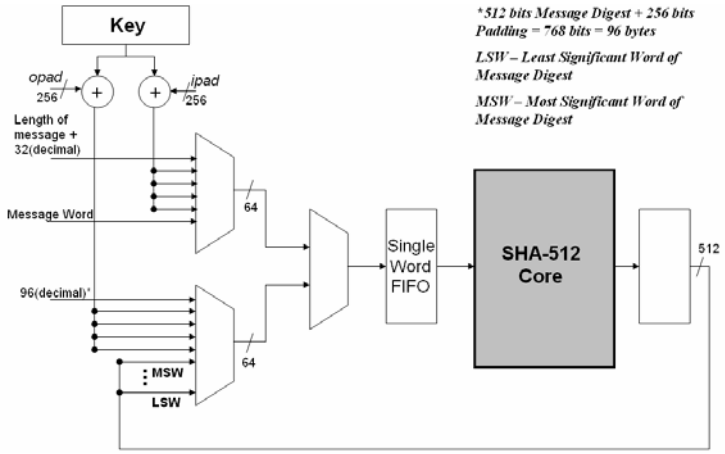

Figure 9: Implementation Aspects of HMAC with SHA-512

Here, it is worthwhile to reiterate some specifics about the functionality of the SHA-512 core. The core reads input data from an input FIFO and writes results to the output FIFO. Since, the HMAC wrapper covers the SHA-512 core, it no longer has direct access to the input FIFO. Hence, a single word FIFO is simulated by introducing a register in the data-path. This serves another useful purpose of breaking up the critical path, thereby improving timing.

Since, input FIFO was just mentioned, let us look at how some of the handshaking signals for the core are derived. Out of the 2 hash operations, the first operation gets data from outside the entity, but the second hash operation utilizes internally generated data for performing computations. It follows that the FIFO_EMPTY for the first operation is replicated from the FIFO_EMPTY signal which is input to the HMAC. For the second hash operation, since all data to be processed is already available, the FIFO_EMPTY signal

is redundant. Hence, the FIFO_EMPTY signal is set to '0' for the second hash operation.

The output of the SHA-512 core is stored in a register, the output of which is fed back to the second multiplexer-counter module.

The most important part of the HMAC wrapper is the control unit. It is extremely involved since it has to handshake with the control unit of the SHA-512 core. Separate parts of the control unit generate control signals for the multiplexers and enables for registers. Another part controls the interface with the SHA-512 core. Still another generates the output handshaking signals for writing to the output FIFO and indicating the end of operation.

8. Implementation Results

This section explains the results obtained for FPGA implementation as well as ASIC synthesis.

8.1 Results of FPGA Implementation

In this section, results of FPGA implementation of all the designs will be presented. A comparative study of results of platform specific designs and platform independent designs is provided. Platform specific designs are those targeting the Xilinx VirtexII family of devices.

8.1.1 FPGA Implementation results for SHA-1

	Design Version	
	FPGA Specific	Platform Independent
Target Device	2v250fg456	2v250fg456
Occupied CLB Slices	509	482
Occupied LUTs	806	774
Occupied FFs	239	396
Minimum Clock Period	11.373 ns	11.864 ns
Maximum Clock Frequency	87.92 MHz	84.28 MHz

8.1.2 FPGA Implementation results for SHA-512

	Design Version	
	FPGA Specific	Platform Independent
Target Device	2v250fg456	2v250fg456
Occupied CLB Slices	1534	1534
Occupied LUTs	2693	2916
Occupied FFs	2424	2435
Minimum Clock Period	14.839 ns	15.216 ns
Maximum Clock Frequency	67.38 MHz	65.72 MHz

8.1.3 *FPGA Implementation results for HMAC SHA-512*

	Design Version	
	FPGA Specific	Platform Independent
Target Device	2v500fg456	2v500fg456
Occupied CLB Slices	3070	3070
Occupied LUTs	4317	4521
Occupied FFs	3737	3747
Minimum Clock Period	14.643 ns	14.382 ns
Maximum Clock Frequency	68.29 MHz	69.53 MHz

Performance results are compared using graphs later in this section. It is interesting to note here, that Platform independent design is actually a little smaller than the FPGA specific design in the table in Section 8.1.1. An analysis of the synthesis results revealed that the tool used the resource sharing principle to share some logic and then pruned the redundant combinational logic.

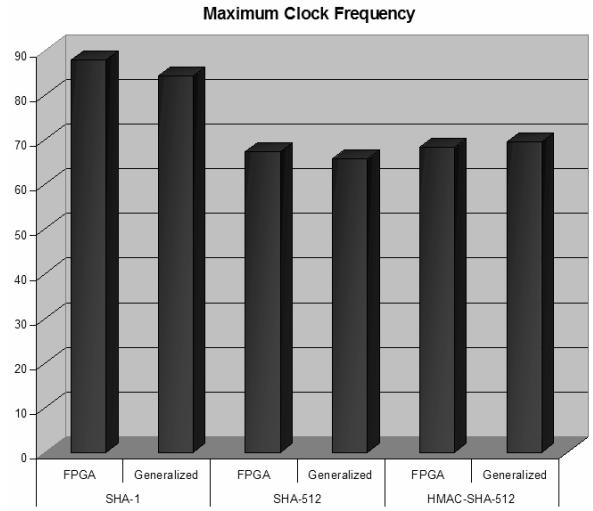

Figure 10: Comparison of Maximum Clock Frequency for FPGA Implementations

An interesting observation from the above tables reiterates the fact explained earlier about coding at the RTL level rather than at a low-level of abstraction for sizeable circuits. Table in section 8.1.3 shows an extreme kind of result wherein, both the implemented versions have exactly the same resource utilization in terms of occupied CLB Slices, but the platform independent version is slightly faster than the FPGA specific one. A simple explanation for this is that the FPGA synthesis tool has much more scope of optimizing the circuit when it is platform independent than when it is described at a low-level of abstraction.

A thorough analysis reveals that the synthesis tool has performed logic replication on high fan-out nets, thereby increasing the number of LUTs used. This leads to better timing performance as can be seen from the table in section 8.1.3.

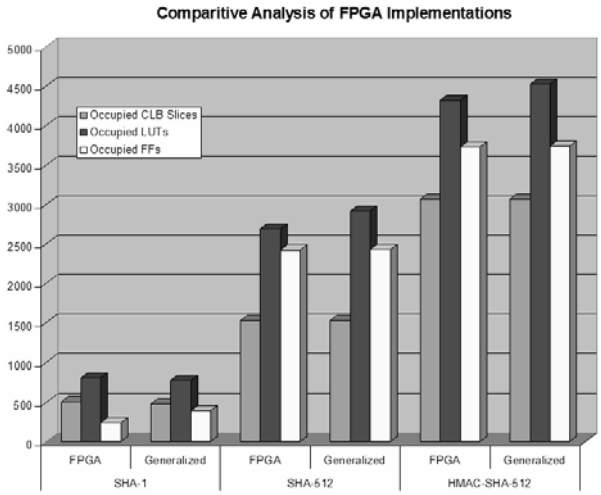

Figure 11: Comparison of Resource Utilization for FPGA Implementations

8.1.4 Throughput

Throughput is the amount of data processed per clock cycle. The units of throughput are Gbps.

Throughput
= (Block Size/No. of Clock Cycles) * Clock Frequency
Block Size for SHA-1 is 512 bits and for SHA-512 and HMAC-SHA-512 it is 1024 bits. This gives a throughput of about 0.5 Gbps for SHA-1 and HMAC-SHA-512 and about 0.9 Gbps for SHA-512.

The results are graphically represented in Fig. 12 below.

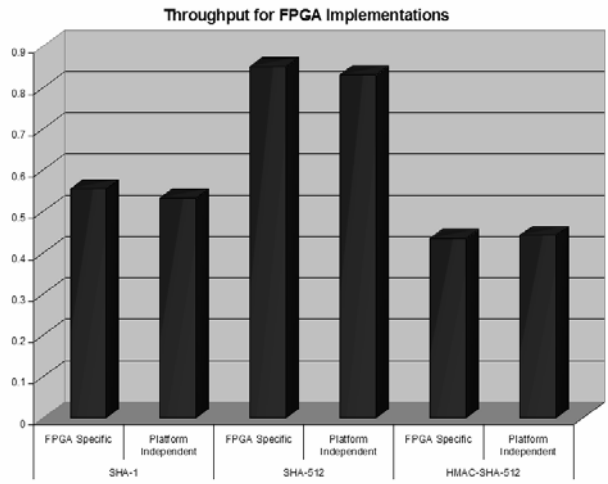

Figure 12: Throughput for FPGA Implementation

8.2 Results of ASIC Synthesis

8.2.1 Synopsys Synthesis results for SHA-1

	Target Technology	
	90 nm	130 nm
Combinational Area	9707	19935
Non-Combinational Area	18659	41824
Total Cell Area	28366	61759
Clock Period Requested	10 ns	10 ns
Clock Period Obtained	7.49 ns	10 ns

In all results related to Synopsys Synthesis, the units for area are *square microns*.

8.2.2 Synopsys Synthesis results for SHA-512

	Target Technology	
	90 nm	130 nm
Combinational Area	33459	82828
Non-Combinational Area	62145	127072
Total Cell Area	95604	209905
Clock Period Requested	10 ns	10 ns
Clock Period Obtained	6.86 ns	7.87 ns

8.2.3 Synopsys Synthesis results for HMAC with SHA-512

	Target Technology	
	90 nm	130 nm
Combinational Area	45337	108988
Non-Combinational Area	93521	187772
Total Cell Area	138860	296758
Clock Period Requested	10 ns	10 ns
Clock Period Obtained	9.88 ns	9.98 ns

Tables in sections 8.2.1 and 8.2.2 show the relationship between resource utilization and amount of security of a particular hash function. Considering results for 90 nm technology for SHA-1 and SHA-512, we find that the ratio of resource utilization (Total Cell Area) is approximately equal to the ratio of the amount of security of the 2 functions. The ratio of total cell areas is about 0.29, while the ratio of amount of security is 160/512≈0.3.

Another interesting feature about Synopsys relates to meeting the user timing requirements. Presenting numerical values here would confuse the matter. Hence, only the gist of the matter is explained. In all the above tables, the requested clock frequency is 10 ns. For a comparative analysis, it is only fair that all the designs be synthesized under similar conditions. But the tool has the ability to meet any reasonable timing estimates.

Synthesis was performed after decreasing the requested clock period to 7.5 ns. Interestingly enough, the synthesis tool could match that timing with a slightly increased cell area. The increase in cell area occurs because of pipelining and replication of logic with high fan-out nets in order to decrease the load on drivers, thereby decreasing point-to-point delays.

Even more interesting was the fact that when requested clock frequency was set to 20 ns, the tool tried to match it with a small reduction in circuit area.

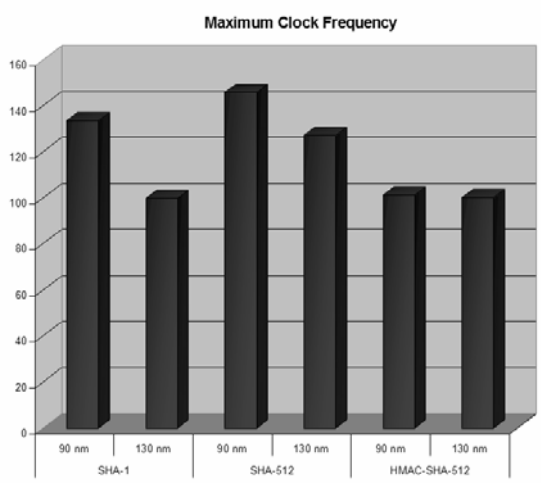

Figure 13: Comparison of Maximum Clock Frequency for ASIC Synthesis

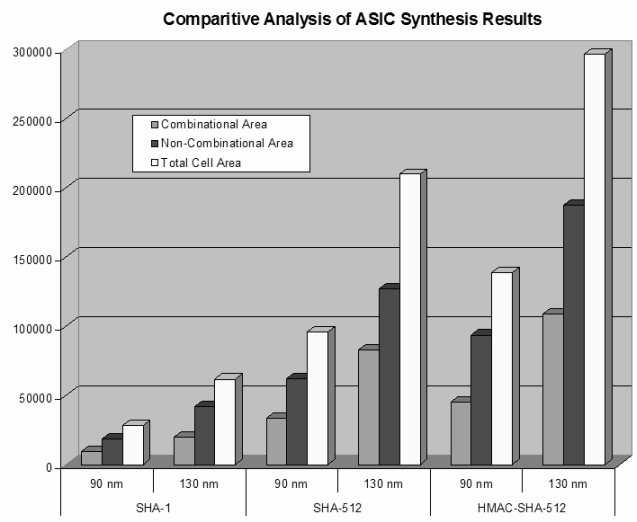

Figure 14: Comparison of Resource Utilization for ASIC Synthesis

8.2.4. Throughput

Throughput results after ASIC Synthesis are much higher than those of FPGA Implementations. The values are calculated using the formula stated in section 8.1.4.

The results are shown graphically in Fig. 15.

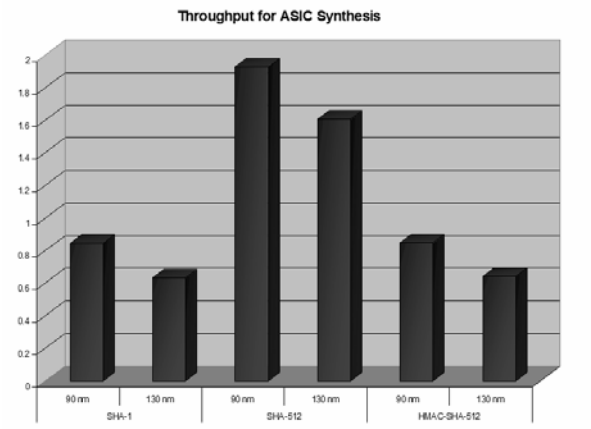

Figure 15: Throughput for ASIC Synthesis

9. Conclusion

The overall analysis of implementation results indicates that an authentication engine can be implemented on FPGAs with a reasonable overhead. Implementations using FPGA resources with minimal overheads can be used to add security features to already existing devices. ASIC synthesis results show that a possible addition to the FPGA fabric for future families would not increase the cost by a large margin. Implementation on the FPGA fabric would be recommended since, it is the only implementation which is really secure. In case of implementation using FPGA resources or soft-core processors, the integrity of the authentication core cannot be guaranteed. Hence, it is not advisable for high-security applications. Also, ASIC implementations have a very high throughput (≈ 1 Gbps) which would prevent the authentication engine from being a bottleneck in the system.

10. Acknowledgment

I would like to thank Tim Grembowski for giving me access to VHDL codes developed by him during the course of his Master's Thesis at GMU and also for patiently explaining his design methodology.

11. References

[1] *Cryptography and Network Security: Principles and Practice*, 3rd ed., William Stallings, Prentice Hall

[2] *Applied Cryptography – Protocols, Algorithms and Source Codes in C*, Bruce Schneier, John Wiley and Sons pp. 429 – One Way Hash Functions

[3] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J.Flidr, T. Lehman, B. Schott, *"Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512"* Proc. Information Security Conference, Sao Paulo, Brazil

[4] *FPGA implementation of SHA-1 Secure Hash standard*, Roar Lien – Master's Thesis, GMU

[5] ***FIPS 180-2*** *Secure Hash Standard – Specifications of SHA functions and source of test vectors* -

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[6] ***FIPS 198***, *The Keyed-Hash Message Authentication Code (HMAC) Standard,* Crypto ToolKit, http://csrc.nist.gov/CryptoToolkit/tkmsgauth.html

[7] *Is Your FPGA Design Secure?* – XCell Journal Online - *http://www.xilinx.com/publications/xcellonline/xcell_47/xc_secure47.htm*

[8] *FPGA Viruses* - Ilija Hadzic, Sanjay Udani and Jonathan M Smith - Distributed Systems Laboratory, University of Pennsylvania - *ww.cis.upenn.edu/~boosters/fpgavirus.ps*

[9] *Computer Arithmetic*, B. Parhami. Oxford University Press, 2000.

[10] *Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware*, Pawel Chodowiec, MS CpE Candidate, Master's Thesis, March 2002, GMU

[11] *Security Scenarios,* Actel Corporation *www.**actel**.com/documents/SecurityScenarios.pdf*

[12] Synopsys Documentation available at http://cpe02.gmu.edu/synopsysdocs/

[13] TSMC Library Documentation available at http://cpe02.gmu.edu/synopsysdocs/tsmc/

[14] *Virtex Datasheet* - available at *http://direct.xilinx.com/bvdocs/publications/ds003.pdf*