An Efficient Hardware Accelerator for HS1-SIV Encryption Algorithm

Maththaiya Durai  
Dept. of Electrical and Electronic Engineering  
California State University, Sacramento  
Sacramento, CA 95819 USA  
maththaivadurai@csus.edu

Behnam S. Arad  
Department of Computer Science  
California State University, Sacramento  
Sacramento, CA 95819-6021 USA  
arad@csus.edu

Abstract

Data security is a major concern for everyone in today’s informational world. Encryption is the process of encoding messages or information in such a way that only authorized parties can read it. It is one of the major information security solutions. Hash Stream1-Synthetic Initialization Vector (HS1-SIV) is a recently developed and fast encryption algorithm. In this paper, we present a hardware accelerator for the HS1-SIV encryption algorithm. Our implementation relied on parallelism and pipelining to increase message encryption throughput. The hardware realization of HS1-SIV encryption algorithm involved designing a hardware data path and control unit, modeling the data path in System Verilog hardware description language, validating and synthesizing it using a 90nm hardware cell library. The proposed design was thoroughly verified using a System Verilog layered test bench. The proposed pipelined model is very efficient and can encrypt messages at the rate of 457 Gigabytes per second.

Keywords: Encryption, hardware accelerator, System Verilog, pipelining, simulation, synthesis.

1. Introduction

Storage, processing, and communication of data in electronic form has significantly increased over the past twenty five years. Availability of data in electronic form has attracted nefarious elements. Many high profile data hacks in the recent years highlight the need for data security and throw light on how invaluable data can be. One approach to achieve data security efficiently is through encryption. It is the process of encoding messages or information in such a way that only authorized parties can access it.

To ensure security and confidentiality the message is encrypted using an encryption algorithm (cipher). The encrypted message (cipher text) yields the original message only upon decryption. Encryption key (pseudo random number) generated by an algorithm forms the crux of the encryption process. An authorized recipient who has the key provided by the originator can easily decrypt the message while an unauthorized interceptor cannot. Due to its effectiveness, encryption has been long employed by government agencies and military to ensure the secrecy of communication.

HS1-SIV is one of the faster authenticated encryptions. It requires a single key for both encryption and decryption, which is independent of the message and the cipher itself. There are three variations of HS1-SIV algorithm (HS1-SIV-LO, HS1-SIV, and HS1-SIV-HI) depending on the number of bytes used in the hashing part of algorithm, the size of the synthetic IV, the number of internal rounds used by the stream cipher, and the collision level of the hashing algorithm [5] (specification).

In this paper, we propose a hardware model for implementing the HS1-SIV algorithm using the System Verilog hardware description language. The model relies on pipelining to greatly improve the throughput of the design. The model was thoroughly verified using a layered test bench. The validation process continued until 100% functional coverage was achieved. Then, the verified model was synthesized using the Synopsys Design-Compiler tool to get an estimate of the number of gates, area and timing of the hardware model.

The rest of the paper is organized into six sections. Section 2 covers an overview of the HS1-SIV encryption algorithm. Section 3 discusses the design and architectural view of the hardware implementation of the HS1-SIV encryption algorithm. Section 4 describes the modeling of the HS1-SIV algorithm using System Verilog. Section 5 covers the verification of the hardware model. Section 6 covers the synthesis of the hardware model using the Synopsys Design Compiler synthesis tool. Finally, Conclusions are stated in Section 7.

2. HS1-SIV Encryption Algorithm

2.1 Overview

This section introduces the concept of HS1-SIV encryption algorithm. HS1-SIV is an authenticated-encryption algorithm developed by Krovetz in 2014. It is
designed to exploit 32-bit multiplication and Single Instruction Stream Multiple Data Stream (SIMD) processing, which are well-supported on almost all current CPUs [5] (specification).

The HS1-SIV algorithm is a symmetric cipher. In symmetric ciphers, a single secret key is used for both the encryption and decryption, whereas in asymmetric ciphers, there are two sets of keys known as private and public keys. The plaintext is encrypted using the public key and can only be decrypted using the private key [9] (book).

HS1-SIV uses a new pseudo random function called HS1 to provide authenticated encryption via Rogaway and Shrimpton’s SIV mode [10] (proceedings). HS1-SIV maintains full integrity and confidentiality over the message. HS1-SIV is designed to have the features such as competitive speed on multiple architectures, provable security, general-purpose PRF, scalable and nonce misuse resistant [5] (specification). HS1 uses a universal hash function to accept arbitrary strings and a stream cipher to produce its output. SIV, as defined in [10] (proceedings), uses a block-cipher-based PRF to create a synthetic IV (an SIV) from given associated data and plaintext.

“HS1-SIV uses HS1 to instantiate SIV mode. If A is the associated data, M is the plaintext, and N is HS1’s IV, then the SIV is defined as the first 16 bytes of HS1(A||M, N) and cipher text C is defined as all but the first 16 bytes of HS1(A, M, N) XOR’ed with M. The SIV and cipher text are bundled together to create the final cipher text. If (A, M, N) is repeated, then an observer knows this fact because the scheme is deterministic and the SIV will be identical, but no security degradation otherwise occurs. Supplying N as a nonce thus improves security by masking repeated encryptions.

HS1 operates by pairing an almost-universal hash function with a stream cipher. When given an input (IV) pair, HS1 uses the hash function to hash the input, it then XOR’s this hash result with the stream cipher’s key and uses the HS1 IV as the stream cipher’s IV. The stream cipher produces as many bytes as desired. As long as a (hash result, IV) pair is never repeated, and the stream cipher is secure against related-key attacks, the stream cipher will produce independent pseudorandom output streams. We introduce a new hash HS1-Hash which we use for the almost-universal phase of HS1, and Bernstein’s CHACHA is used as the stream cipher” [5] (specification).

2.2 HS1-SIV Process

An authenticated-encryption scheme is a shared-key encryption scheme which provides both privacy and authenticity. The encryption algorithm takes a key, a plaintext, an associated data and a nonce, and returns a cipher text and an SIV. The decryption algorithm takes a key, a cipher text, an SIV, an associated data and a nonce, and returns either a plaintext or an invalid indicator.

```
k = HS1-subkeygen [b, t, r, ℓ] (K)
M' = A||M || |A|| toStr(8, jMj)
T = HS1 [b, t, r] (k, M', N, ℓ)
C = M_ HS1 [b, t, r] (k, T, N, 64 + jMj) [64, jMj]
Inputs: (K, M, A, N)
Output: (T, C)
```

Where K, a non-empty string of up to 32 bytes
M, a string shorter than 264 bytes
A, a string shorter than 264 bytes
N, a 12-byte string
(T, C), strings of ℓ and jMj bytes, respectively

Figure 1 : HS1-SIV Process, Based on [5]

“HS1-SIV has three variations depending on the parameters b, t, r, and ℓ. Parameter b specifies the number of bytes used in part of hashing algorithm (larger b tends to produce higher throughput on longer messages). Parameter t selects the collision level of the hashing algorithm (higher t produces higher security and lower throughput). Parameter r specifies the number of internal rounds used by the stream cipher (higher r produces higher security and lower throughput). Parameter ℓ specifies the byte-length of synthetic IV used (higher ℓ improves security and increases cipher text lengths by ℓ bytes). The following table names parameter sets.” [5](specification) [11] (proceedings)

<table>
<thead>
<tr>
<th>Name</th>
<th>b</th>
<th>t</th>
<th>r</th>
<th>ℓ</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS1-siv-lo</td>
<td>64</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>HS1-siv</td>
<td>64</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>HS1-siv-hi</td>
<td>64</td>
<td>6</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

Next we describe 4 stages of HS1-SIV algorithm.

2.2.1 SubKey Generation

The first stage of HS1-SIV, Subkeygen generates the sub keys from the input key of up to 32 bytes. HS1-SIV uses CHACHA12, a stream cipher to produce pseudorandom string. This phase produces three keys denoted by ks, kn, kp which are used in the later hash stages.

```
T= Chacha[r] (K, 0, N, 0^)
kS= T[0, |C|]
kN= toInts(4, T[|C|, |NH|])
kp= map(mod 2^60, toInts(8, T[|C|+ |NH|, |P| ]))
```

Figure 2 : Subkey Generation, Based on [5]
where $|C| = 32$, $|NH| = b + 16(t - 1)$, $|P| = 8t$, $y = |C| + |NH| + |P|$, $N = \text{toStr}(12, b^{248} + t^{240} + r^{232} + \ell^{216} + |K|)$, toInts($n$, $S$) is the vector of integers obtained by breaking $S$ into $n$-byte chunks and Little-endian interpreting each chunk as an unsigned integer, and $||$ represents concatenation.

### 2.2.1.1 CHACHA Stream Cipher

“CHACHA stream cipher takes four inputs, a 32-byte key, an initial counter value, a 12 byte Iv, and a plaintext. CHACHA produces the cipher text which is as long as the plaintext. CHACHA builds a 4×4 matrix, and transforms the matrix through 12 rounds, and adds the result to the original matrix to obtain a 16-word (64-byte) output block.

CHACHA, uses 4 additions and 4 exclusive-or’s (XOR’s) and 4 rotation operations to update 4 32-bit state words. However, CHACHA applies the operations in a different order, and in particular updates each word twice rather than once. CHACHA updates $a$, $b$, $c$, $d$ as follows:

\[
\begin{align*}
a &+ b; & d &^\leftarrow a; \quad d \lll 16; \\
c &+ d; & b &^\leftarrow c; \quad b \lll 12; \\
a &+ b; & d &^\leftarrow a; \quad d \lll 8; \\
c &+ d; & b &^\leftarrow c; \quad b \lll 7;
\end{align*}
\]

Figure 3 : CHACHA Quarter Round, Based on [5]

CHACHA first round modifies first, fourth, third, second, first, fourth, third along columns, and the second round modifies first, fourth, third, second, first, fourth, third, second along southeast diagonals: The four quarter-round words are always in top-to-bottom order in the matrix, to improve diffusion slightly.” [1] [2] (proceedings) [7] (online)

| QUARTERROUND (X0, X4, X8, X12) |
| QUARTERROUND (X1, X5, X9, X13) |
| QUARTERROUND (X2, X6, X10, X14) |
| QUARTERROUND (X3, X7, X11, X15) |
| QUARTERROUND (X0, X5, X10, X15) |
| QUARTERROUND (X1, X6, X11, X12) |
| QUARTERROUND (X2, X7, X8, X13) |
| QUARTERROUND (X3, X4, X9, X14) |

Figure 4 : CHACHA Round Order, Based on [5]

### 2.2.2 Message Padding

In this phase, the plaintext ($M$), associated data ($A$), length of plaintext data and length of associated data are concatenated to produce the plaintext prime. The later stages of hash operate on the produced plaintext prime. Figure 5 summarizes this phase.

\[
M' = A || M || |A| || |M|
\]

Figure 5 : Message Padding, Based on [5]

### 2.2.3 SIV generation

The next stage in HS1-SIV is SIV generation based on HS1-PRF. HS1-pseudo random function is a composition of HS1-hash, an almost universal hash function, with CHACHA, a stream cipher.

\[
A_i = \text{HS1-Hash} (kn[4i, b/4], kp[i], M) \quad \text{for each } 0 < i < t \\
Y = \text{Chacha}[r](\text{pad}(32, A_0 || A_1 . . || A_{t-1}) ^ ks), 0, N, 0^y)
\]

where $ks$ - subkey string of 32 bytes, $kn$ - a vector of $b/4 + 4(t-1)$ integers, $kp$ - a vector of $t$ integers, $M$ - an input string of any length.

Figure 6 : SIV Generation, Based on [5]

#### 2.2.3.1 HS1-Hash

First stage in SIV generation is breaking the plaintext prime into a vector of intermediate strings using HS1-hash. It is a composition of two hashes, NH hash and Poly hash. Similar to the techniques used in UMAC and VMAC, the NH hash is used to reduce the input by a fixed ratio to an intermediate string which is then hashed to a fixed size by a polynomial evaluation. To reduce the chance of collision, this hashing procedure is repeated $t$ times, with different keys, for a higher collision probability [5] (specification).

\[
\begin{align*}
ai &= (\text{NH}(kn, mi) + |Mi|\text{mod} 16) \text{mod} 260, \text{for } 1 < i < n \\
h &= (kp + a1*kp_{t-1} + a2*kp_{t-2} + . . . + a0)\text{mod} (261 - 1) \\
Y &= \text{toStr}(8, h)
\end{align*}
\]

where $n$ = max(⌊|M|/b⌋ + 1), $|Mi| = b \text{for each } 1 < i < n$, $mi$ = toInts(4, $\text{pad}(16, Mi)$) for each $1 < i < n$, $Y$, an 8 byte (if $r < 4$) or 4 byte (if $r = 4$) string.

Figure 7 : HS1-Hash, Based on [5]

### 2.2.4 HS1-SIV-PRF

The last stage in HS1-SIV applies HS1 pseudo random function to the synthetic IV produced by SIV generation stage and produces the cipher text. In this stage, the SIV produced by the previous stage is used as a plaintext. HS1_hash is used to break the plaintext into vector of intermediate strings and applied to CHACHA stream cipher to produce the output string. The output string of CHACHA...
is XOR'ed with the input message and concatenated with synthetic IV to produce the final cipher text.

3. Hardware Realization of HS1-SIV

In this section, we propose a hardware realization of the HS1-SIV algorithm. In the subsequent sections we show that the pipelined design is fully functional and synthesizable. This means that the Register Transfer Level (RTL) description of the designs can be converted to an optimized gate-level net list using a logic synthesis tool. We developed a non-pipelined implementation primarily to assess the efficiency of the pipelined implementation. That implementation is not described here due to space limitations. It is described in detail in [3] (report).

To keep the scope of the project, we assumed that each incoming message 512-bits wide. If the message packet is of variable size, then the design will require FIFO logic to store the entire message before the hashing process. The block size of HS1-SIV design inputs are shown in Table 2:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext</td>
<td>512 bits</td>
</tr>
<tr>
<td>Associated data</td>
<td>512 bits</td>
</tr>
<tr>
<td>Key</td>
<td>256 bits</td>
</tr>
<tr>
<td>Nonce</td>
<td>96 bits</td>
</tr>
</tbody>
</table>

Table 2 : Inputs of HS1-SIV

In the following subsections we introduce a pipeline implementation of the HS1-SIV algorithm. The design hierarchy of HS1-SIV pipelined design is shown in Figure 8.

3.1 Subkeygen stage

The main task of subkeygen stage is to generate sub keys from the input key using CHACHA12 stage and keys_gen stage. CHACHA12 stage takes the length of pseudo random string, input key and initial counter value as inputs and produces the pseudo random string of as long as needed. Keys_gen stage takes the produced pseudo random string and breaks into three sub keys.

A CHACHA12 stage produces 512-bit pseudo string as its output. The inputs and outputs of this stage are shown in Figure 9.

Each quarter round in the CHACHA12 performs twelve rounds of CHACHA transformations. A quarter round stages takes four 32-bit inputs and does the rotation of the four inputs and produces four 32-bit outputs. This stage uses 4 additions, 4 XOR’s and 4 rotations to update 4 32-bit state words.

Figure 8 : Pipeline HS1-SIV Design
3.2 Message Padding stage

This stage concatenates the plaintext, associated data, and two 64-bit parameterized inputs and produces 1536-bit plaintext prime string as output.

3.3 HS1_prf3 stage

$HS1_{prf3}$, pseudo-random function is a composition of $HS1$-hash and CHACHA12 stream cipher. $HS1$-hash is itself a composition of two hash functions: NH hash and polynomial evaluation hash. CHACHA12 takes the produced string from HS1-hash function as inputs and produces pseudorandom string as output. The HS1_prf3 stage block diagram is shown in Figure 10.

3.4 HS1_prf stage

$HS1_{prf}$ and $HS1_{prf3}$ stages are similar, except the length of the input plaintext and output cipher text. The 128-bit cipher text produced by $HS1_{prf3}$ stage is synthetic IV (SIV). The SIV is used as input plaintext for $HS1_{prf}$ stage, which produces 512 bits cipher text. CHACHA12 takes the produced string from $HS1$-hash function as inputs and produces a 512 bits pseudo-random string as output.

3.5 HS1_siv_prf stage

$HS1_{siv_prf}$ stage concatenates both SIV and cipher text and produces the 640 bits final cipher text. The entire hashing process is performed for each sets of data received by the first stage.

4. Modeling of HS1-SIV

All the blocks in the design hierarchy are modeled in behavioral style of System Verilog. This style models the functionality of a digital circuit at highest level of abstraction. It captures the functionality of a design at an algorithmic level.

The design methodology undertaken in this project is the bottom-up methodology [8] (book). In this approach, the leaf components in the design hierarchy are developed first and the higher-level components are constructed by instantiating and inter-connecting the subcomponents.

We utilized advance features of System Verilog such as new data types, streaming operators, and the new procedural blocks. Other features that are related to verification are described in detail in Section 5.

System Verilog offers many new data types to be used in a design. Some of them are logic, enum and struct. Logic is a 1-bit 4-state variable, like the Verilog reg type. It can be declared as any vector size. Enum type is an enumerated net or variable with a labeled set of variables, similar to C enum type, but with additional syntax and semantics for modeling hardware. Struct type is a collection of variables that can be referred to individually or collectively, similar to the C struct type [14] (book).

The streaming operators perform packing of bit-stream types into a sequence of bits in a user-specified order. The streaming operator $<<$ or $>>$ determines the order in which blocks of data are streamed: $>>$ causes blocks of data to be streamed in left-to-right order, while $<<$ causes blocks of data to be streamed in right-to-left order [6] (online).

The specialized always_comb, always_latch, and always_ff procedural blocks indicate the design intent. The software tools do not need to infer the designer’s intent. If the content of a specialized procedural block does not match the rules for that type of logic, software tools can issue the warnings [14] (book). The complete System Verilog code for the pipelined implementations is provided in [3] (report).

Here is summary of some modeling choices we made in this work:

Figure 9 : CHACHA12 Stage

Figure 10 : HS1_prf3 Stage
• The block in subkeygen is modeled as a System Verilog module which instantiates CHACHA12 and keys_gen module.
• CHACHA12 stage requires 12 rounds of CHACHA transformation. Each round of is modeled as a module.
• Each quarter round of CHACHA transformation is modeled as a CHACHA12_qtr module and is instantiated four times within CHACHA12 round module to produce one round of CHACHA operations.
• The HS1_prf3 stage is modeled as System Verilog module which instantiates NH_hash3, Poly_hash3 and CHACHA12_h1 modules.
• CHACHA12_h1 module is functionally similar to CHACHA12 module, even though the number of input elements and output elements are lesser.
• The HS1_prf module is functionally similar to HS1_prf3 module, the variation being the size of input plaintext and output cipher text.

5. HS1-SIV Verification

5.1 Overview

This section describes the layered System Verilog test bench developed to verify the functionality of the proposed HS1-SIV design. The simulation was done using Synopsys® VCS simulation tool. The test bench fully validated the design by constructing random messages, keys and nonce, passing them to the model, and comparing the actual HS1-SIV output to the expected result. A scoreboard block was used to generate the expected value for each test vector.

5.2 Layered Test bench Verification

The layered test bench comprises of multiple layers to take full advantage of code reuse and automation [15] (online). Figure 11 shows the block diagram of verification infrastructure used for the proposed implementation of HS1-SIV. Top is the verification wrapper that includes the program, interface, design under test (DUT), and clock generation blocks needed for validation of the DUT.

5.3 Interface

A System Verilog interface encapsulates the communication between blocks, allowing a smooth refinement from abstract system-level through successive steps down to lower RTL and structural levels of the design.

Interfaces also facilitate design re-use [12] (online). In this project, we used two interfaces, namely interface_top and interface_dut. The interface_top was used to establish a communication between the programs representing the test bench and the HS1_SIV top module. The interface_dut was used to set up a communication between the modules in top module. Two modports were used in the interface_top, one for the program and one for the DUT. Seven modports were used in the interface_dut, one for each module in top module.

5.4 Threads and IPC

System Verilog constructs such as fork join_none and fork join_any can be used to dynamically create new threads, in addition to the standard fork_join. These threads communicate and synchronize using events, semaphores, mailboxes, and classic @ event control and wait statements. System Verilog enables design of a powerful and flexible test bench environment, as Object Oriented Programming (OOP) objects are created and destroyed, they can run in independent threads [15] (online).

Each object gets a transaction from an upstream object, performs some operations, and then passes the transaction to a downstream object. The channel allows its driver and receiver to operate asynchronously. In this project, standard fork/join was used to create threads, and mailboxes were used to provide inter process communication between the threads. A mailbox is just a FIFO, with a source and sink. The source puts a value into the mailbox, and the sink gets values from the mailbox. It can have a maximum size or can be limited.

5.5 Program

In System Verilog, the test bench can be developed as a program block, which supports multiple implicit timing
regions for sampling test bench results, scheduling the design events, observing System Verilog assertions, taking reactive steps in the test bench [4] (online).

The test bench described in this section consists of a single program. It uses the Object Oriented Programming feature of System Verilog to build random test vectors dynamically. The random test vectors were generated using “rand” system built-in task inside a “Class”. The randomization method used in the test bench is “Constraint Based”, which programs the simulator to limit the selection of randomized value to a specific value-set pool.

5.6 The Environment

In this work, a verification environment was properly initialized and synchronized using mailbox, avoiding race condition between the design and the test bench. The verification environment automates the generation of input stimuli, and reuses existing models and other infrastructure. The test bench (program) verifies the design until functional coverage reaches 100%. The verification procedure involves generating stimuli randomly, passing them to the design through interface_top and verifying the correctness of the results obtained.

The generator, agent, driver, monitor, checker and scoreboard are all classes, modeled as transactor. They are instantiated inside the Environment class. The program blocks instantiates the environment class.

5.7 Validation

Scoreboard block has the functions to generate the expected results of the HS1-SIV design. The Scoreboard then forwards the results to the checker class to verify whether the actual results from DUT and expected results from Scoreboard matches, and sets up the pass/fail flag accordingly. This verifies the HS1-SIV design output correctness and proper functional working.

5.8 Functional Coverage

Functional coverage is a measure of how an RTL model matches the specification. In a System Verilog test bench, the coverage is calculated by sampling the values of variables and expressions. These sample locations are known as cover points. Multiple cover points that are sampled at the same time, such as when a transaction completes, are placed together in a cover group. A cover group is similar to a class, it contains cover points, options, formal arguments, and an optional trigger. Sample function was used to trigger the cover group [15] (online).

A cover group encapsulates all the cover points for the randomized plaintext, associated data, key and nonce, for gathering coverage. These randomized signals defined as the cover points are necessary to verify HS1-SIV design, by measuring the functional coverage. We defined our cover points based on plaintext, associated data, and keys to measure functional coverage of our verification. We were able to achieve 100% functional coverage which reflects the thoroughness of the verification process and robustness of the model.

6. Logic Synthesis

In this section, the synthesis of the HS1-SIV hardware model is described. One of the objectives of this project was to develop a synthesizable model of the HS1-SIV algorithm. Logic synthesis is the process of converting a high-level description of a design into an optimized gate-level representation, given a standard cell library and certain design constraints. The Synopsys® Design Compiler is the synthesis tool used in this project.

Logic synthesis tools take the HDL model of a design, information on cells from a technology library, and the design constraints. A technology library can have simple cells, such as basic logic gates like AND, OR, and NOR, or macro cells, such as adders, multiplexers, and special flip-flops. The synthesis tool converts the HDL design into an optimized gate-level net list satisfying the given design constraints such as timing, area, testability, and power, using the cells from the technology library [8] (book).

The proposed HS1-SIV model was synthesized towards a 90 nm technology library. We synthesized both the pipelined and non-pipelined implementations. The timing results show that the pipelined design can operate at 71MHz. This is based on the synthesis tool being configured for the high synthesis effort. We expect the design to be able to operate at a higher frequency if it is synthesized using a more recent technology library.

The HS1_hash function for the proposed HS1-SIV design with 512-bit block size, take 30 clock cycles to complete. This is the bulk of the processing for a short message. There are number of multiplication operations in the hash function used. Modified Booth Algorithm is used for faster multiplication, but more sophisticated algorithm can improve speed but take up more hardware area. Each message block uses 114 clock cycles to become cipher text. Further parallel processing can improve this speed.

The synthesis results for HS1_SIV are tabulated in Table 3. It shows the clock frequency, the timing slack, and the cell area generated after synthesis.
Table 3: Synthesis Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Clock period</th>
<th>Timing slack</th>
<th>Area report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-pipeline</td>
<td>85ns</td>
<td>0.3 ns</td>
<td>452481.50 um²</td>
</tr>
<tr>
<td>Pipeline</td>
<td>14ns</td>
<td>0.13 ns</td>
<td>1200432.50 um²</td>
</tr>
</tbody>
</table>

Speed up of the pipeline design is the ratio of total time taken by the non-pipeline design to encrypt \( N \) 512-bit messages to that of the pipeline design:

\[
\text{Speed up} = \frac{20 \times 85 \times N}{114 + N} \times 14,
\]

where 85ns is the non-pipeline design clock period, 20 is the number of steps in the non-pipelined design, 14ns is the clock period the pipeline design, and 114 stages is the number of its stages. This leads to speed up of 121 for large messages. More importantly, the pipeline design can encrypt a 512-bit message per cycle after the first 114 cycles where the pipelined is filled. It can encrypt messages at a very efficient the rate of 457 Gigabytes per second.

7. Conclusion

We proposed an efficient hardware accelerator for a recently developed and fast encryption algorithm called HS1-SIV. The proposed pipeline design breaks each round of the encryption process into several stages to achieve a high encryption rate. This hardware concurrency increases the message encryption throughput and makes the hardware model suitable for time-critical encryption applications.

The pipeline design was thoroughly validated using a layered test bench in System Verilog layered hardware description and verification language, which utilized several unique features of the language including Interface, Program, and also OOP concept. The test bench included Functional Coverage to assess the verification progress of the design features ensuring the design is fully validated.

The model was synthesized using Synopsys Design Compiler tool based on a 90nm technology library. Our synthesis results showed that the pipelined design can achieve encryption bandwidth of 457 Gigabytes per second.

Acknowledgement

In this work we utilized EDA tools set donated to California State University, Sacramento by Synopsys® Inc.

References