Verification of SHA Crypto Processor Core

Dhimmar Ketankumar J.[1] Pankhaniya Ritesh M.[2]
[1, 2] VLSI & Embedded System Design
[1, 2] Gujarat Technological University, Ahmedabad

Abstract---This paper verifies the Secure Hash Algorithm (SHA) Crypto Processor Core by applying different types of test cases as input. The test cases are given as randomized which will more efficient to catch the bug from design compare to direct test. As there are so many versions of SHA algorithm like SHA-1, SHA-2(160, 256, 384 and 512) and also now SHA-3, this paper verifies the 256 bit SHA algorithm. The verification of core is done using System Verilog language. The tool used for verification is questasim 6.6d by Mentor Graphics.

Keywords: Secure hash Algorithm, Test Bench, Test plan, Test case, Randomization, Regression, Coverage.

I. INTRODUCTION

As the complexity of Integrated Circuits (specifically ASIC and SoC) increases, and as their sizes keep reducing, the task of testing the chip gets more and more challenging. Engineers need to come up with better and different methodologies to ensure what goes to the factory for manufacturing is actually what they intended to deliver. Verification occurs at various stages in the ASIC development cycle. How much is enough at each stage is a problem that needs to be addressed on a case to case basis. A sound knowledge of various techniques and awareness of capabilities and limitations of each technique goes a long way in making decisions about when, where and what.

To ensure that a bug free product reaches the customer is a complex activity and poses multiple challenges. Coverage, legacy code, repeatability are issues that need to be tackled. Ensuring that the coverage is at an acceptable level is important. Code coverage is run to find out if all the possibilities of a written code are exercised in a test suite. Simulators from cadence (ius), Synopsys (vcs) and mentor (modelsim) have their own code coverage analyzers. Functional coverage means to find out if each feature listed in the specification for an ASIC/SoC is verified. It is essential that the functional specification document has an individual numbered paragraph for each feature so that traceability is easier. Functional coverage is an activity that needs planning, reviews and careful test case designing.

While choosing the verification flow for a certain ASIC, team needs to look at what is available in terms of resources as well as time, understand the end user requirement, and make a decision on which technique to employ at what stage.

This paper is organized as follows. The overview of SHA-256 algorithm is explained in section 2. The verification flow is explained in section 3. The test bench environment is explained in section 4. The most important part of verification i.e. coverage is explained in section 5. The simulation result is shown in section 6 and section 7 makes a conclusion of the whole work.

II. ALGORITHM OVERVIEW

The algorithm can be described in two stages: pre-processing and hash computation. Pre-processing involves padding a message, parsing the padded message into m-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a message schedule from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest. SHA-256 is a cryptographic hash function with digest length of 256bits. A message is processed by blocks of $512 = 16 \times 32$ bits, each block requiring 64 rounds. [3]

The algorithm uses the following functions and constants:

1) $ch(x, y, z) = (x \wedge y) \oplus (x \wedge z)$,
2) $maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$,
3) $\Sigma0(x) = rotr(x,2) \oplus rotr(x,13) \oplus rotr(x,22)$,
4) $\Sigma1(x) = rotr(x,6) \oplus rotr(x,11) \oplus rotr(x,25)$,
5) $\sigma0(x) = rotr(x,7) \oplus rotr(x,18) \oplus shr(x,3)$,
6) $\sigma1(x) = rotr(x,17) \oplus rotr(x,19) \oplus shr(x,10)$,

To ensure that the messagehas length multiple of 512 bits:

- first, a bit 1 is appended,
- next, k bits 0 are appended, with k being the smallest positive integer such that $l + 1 + k \equiv 448 \bmod 512$, where l is the length in bits of the initial message,
- finally, the length $l < 2^{64}$ of the initial message is represented with exactly 64 bits, and these bits are added at the end of the message.

The message shall always be padded, even if the initial length is already a multiple of 512.

For each block $m \in \{0,1\}^{512}$, 64 words of 32 bits each are constructed as follows:

- the first 16 are obtained by splitting m in 32-bit blocks

$$m = w1 \| w2 \| \cdots \| w15 \| w16. \quad (1)$$

- the remaining 48 are obtained with the formula:

$$w_i = \sigma1(w_{i-2}) + w_{i-7} + \sigma0(w_{i-15}) + w_{i-16},\ 17 \le i \le 64. \quad (2)$$

- First, eight variables are set to their initial values, given by the first 32 bits of the fractional part of the square roots of the first 8 prime numbers:[8]

$$\begin{aligned} &H_1^{(0)} = 0x6a09e667,\ H_5^{(0)} = 0x510e527f \\ &H_2^{(0)} = 0xbb67ae85,\ H_6^{(0)} = 0x9b05688c \quad (3) \\ &H_3^{(0)} = 0x3c6ef372,\ H_7^{(0)} = 0x1f83d9ab \\ &H_4^{(0)} = 0xa54ff53a,\ H_8^{(0)} = 0x5be0cd19 \end{aligned}$$

- Next, the blocks $m^{(1)}, m^{(2)}, \ldots, m^{(N)}$ are processed one at a time:
  For t = 1 to N

– construct the 64 blocks W_i from M(t), as explained above set

$(a,b,c,d,e,f,g,h) = (H_1^{(t-1)}, H_2^{(t-1)}, H_3^{(t-1)}, H_4^{(t-1)}, H_5^{(t-1)}, H_6^{(t-1)}, H_7^{(t-1)}, H_8^{(t-1)})$

- do 64 rounds consisting of:

$$\begin{aligned} &T1 = h + \Sigma 1(e) + Ch(e, f, g) + Ki + Wi \\ &T2 = \Sigma 0(a) + M\,aj(a, b, c) \\ &h = g, \quad d = c \\ &g = f, \quad c = b \\ &f = e, \quad b = a \\ &e = d + T1, \quad a = T1 + T \end{aligned} \tag{4}$$

where, K_i is given by [3]

- compute the new value of $H_j^{(t)}$

$$\begin{aligned} &H_1^{(t)} = H_1^{(t-1)} + a, \quad H_5^{(t)} = H_5^{(t-1)} + e \\ &H_2^{(t)} = H_2^{(t-1)} + b, \quad H_6^{(t)} = H_6^{(t-1)} + f \\ &H_3^{(t)} = H_3^{(t-1)} + c, \quad H_7^{(t)} = H_7^{(t-1)} + g \\ &H_4^{(t)} = H_4^{(t-1)} + d, \quad H_8^{(t)} = H_8^{(t-1)} + h \end{aligned} \tag{5}$$

End for

- The hash of the message is the concatenation of the variables H_i^N after the last block has been processed.

$H = H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)} \| H_8^{(N)}$

III. VERIFICATION FLOW

Verification flow starts with understanding the specification of the chip/block under verification. Once the specification is understood, a document of Test plan is prepared which covers all the possible input scenario of the design. Once this document is done to a level where 70-80 percent functionality is covered, then verification plan document is prepared. Verification plan covers introduction, overview of design under test (DUT), test plan, details of verificationenvironment, verification complete goal and references. After that, test bench environment is created.Once this is done, the whole verification environment is implemented in terms of code using EDA tools. Then we write all possible test cases using system Verilog and then do a simulation of it.

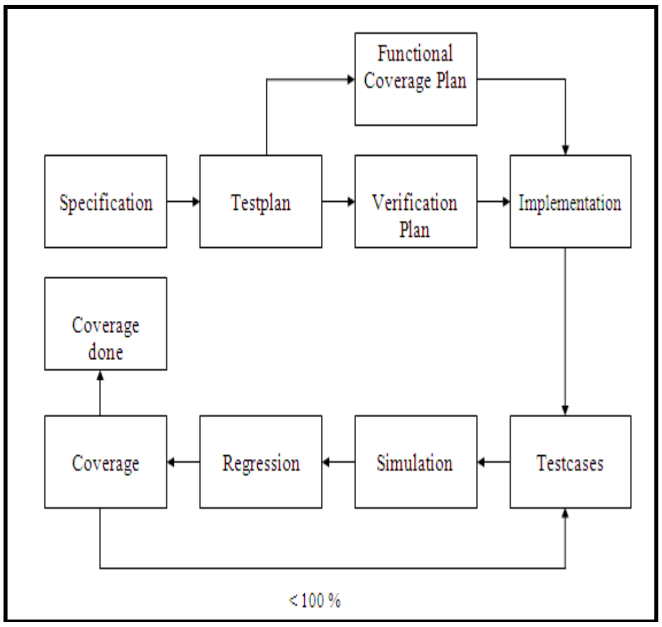


Fig. 1: Verification Flow

In regression test, we can run a test suite on our design at any. The idea is that we can retest everything whenever something changes or periodically as a matter of course. This is sometimes referred to the "Always Working" model, where the design is kept in an always working state. After this, coverage is done which is also explained in detail in below section. If coverage is < 100%, then again we have to write cases and simulate all the cases and so on. This iteration goes on until we achieve our desired verification goal.

IV. TEST BENCH ENVIRONMENT

Environment contains the instances of the entire verification component and component connectivity is also done. Steps required for execution of each component are done in this. The purpose of a Test bench is to determine the correctness of the design under test (DUT). The following steps accomplish this.

– Generate stimulus
– Apply stimulus to the DUT
– Capture the response
– Check for correctness
– Measure progress against the overall verification goals

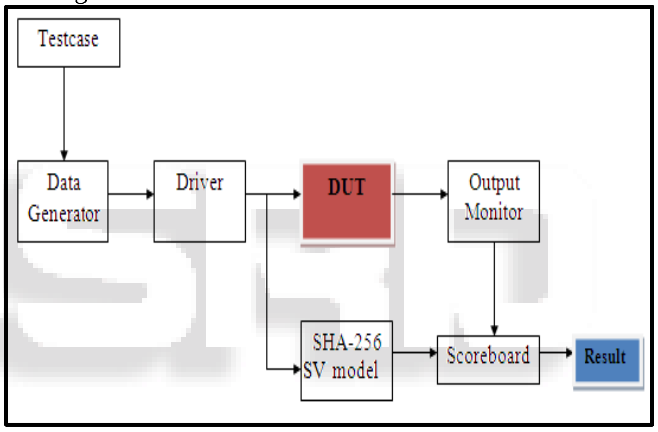


Fig. 2: Test Bench Environment

A. *Test case:*

Test cases are identified from the design specification: a simple task for simple cases. Normally requirement in test cases becomes a test case. Anything that specification mentions with "Can do", "will have" becomes a test case. Corner test cases normally take a lot of thinking to be identified.

B. *Data Generator:*

Data generator generates the data based on the test constraints. This layer also provides necessary information to coverage model about the stimulus generated. Stimulus generated in generator is high level like Packet is with good CRC. This high level stimulus is converted into low-level data using packing. This low level data is just a array of bits or bytes Creates test scenarios and tests for the functionality and identifies the transaction through the interface.

C. *Driver:*

The drivers translate the operations produced by the generator into the actual inputs for the design under verification. Generators create inputs at a high level of abstraction namely, as transactions like read write operation. The drivers convert this input into actual design inputs, as defined in the specification of the designs interface. If the generator generates read operation, then read task is called, in that, the DUT input pin "read write" is asserted

D. Monitor:

Monitor reports the protocol violation and identifies all the transactions. Monitors are two types, Passive and active. Passive monitors do not drive any signals. Active monitors can drive the DUT signals. Sometimes this is also referred as receiver. Monitor converts the state of the design and its outputs to a transaction abstraction level so it can be stored in a 'score-boards' database to be checked later on. Monitor converts the pin level activities in to high level.

E. Scoreboard:

The monitor only monitors the inter interface protocol. It doesn't check the whether the data is same as expected data or not, as interface has nothing to do with the data. Scoreboard basically checks if the output coming out of the DUT is correct or wrong.

V. COVERAGE

Every design verification technique requires coverage metrics to gauge progress, assess effectiveness, and help determine when the design is robust enough for tape-out. At every step of the way and with every bug-finding technology and tool, verification engineers assess coverage results and make critical decisions on what to do next.

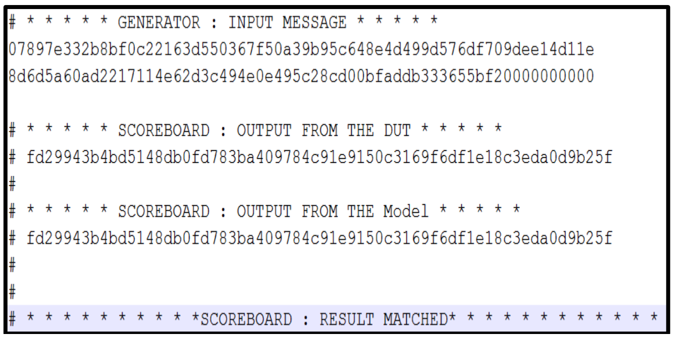
In fact, for the verification of large, complex system-on-chip (SoC) designs, coverage metrics and the responses to them guide the entire flow. The term "coverage-driven verification" describes a methodology built around coverage metrics as the primary way to manage verification.

There are mainly two types of coverage. Code coverage and function coverage. To check whether the test bench has satisfactory exercised the design or not? Coverage is used. It will measure the efficiency of your verification implementation. Again there are many types of code coverage like line coverage, branch coverage, toggle coverage, FSM coverage etc.

Functional coverage is code that observes execution of a test plan. As such, it is code we write to track whether important values, sets of values, or sequences of values that correspond to design or interface requirements, features, or boundary conditions have been exercised. Functional coverage is important to any verification approach since it is one of the factors used to determine when testing is done. Specifically, 100% functional coverage indicates that all items in the test plan have been tested. Combine this with 100% code coverage and it indicates that testing is done.

VI. SIMULATION RESULT

The figure 3 shows the randomized input generated in the generator block. This generated input is padded with 1 and 0's and then the hash computation unit generates hash value. As shown in above figure, the output from the Design under Test (DUT) and the output from the model is compared in scoreboard block. The scoreboard compares the output coming from the model and DUT and it shows the "RESULT MATCHED", if the result matches otherwise it shows "RESULT MISMATCHED".

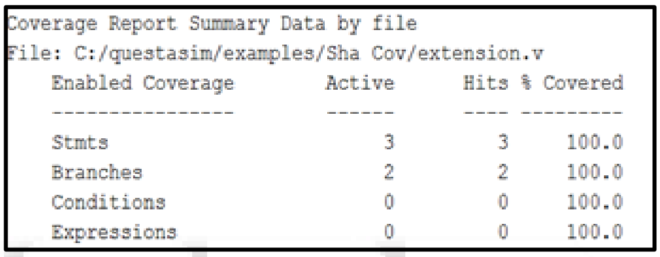


```
# * * * * * GENERATOR : INPUT MESSAGE * * * * *
07897e332b8bf0c22163d550367f50a39b95c648e4d499d576df709dee14d11e
8d6d5a60ad2217114e62d3c494e0e495c28cd00bfaddb333655bf20000000000

# * * * * * SCOREBOARD : OUTPUT FROM THE DUT * * * * *
# fd29943b4bd5148db0fd783ba409784c91e9150c3169f6df1e18c3eda0d9b25f
#
# * * * * * SCOREBOARD : OUTPUT FROM THE Model * * * * *
# fd29943b4bd5148db0fd783ba409784c91e9150c3169f6df1e18c3eda0d9b25f
#
#
# * * * * * * * * * *SCOREBOARD : RESULT MATCHED* * * * * * * * * * *
```

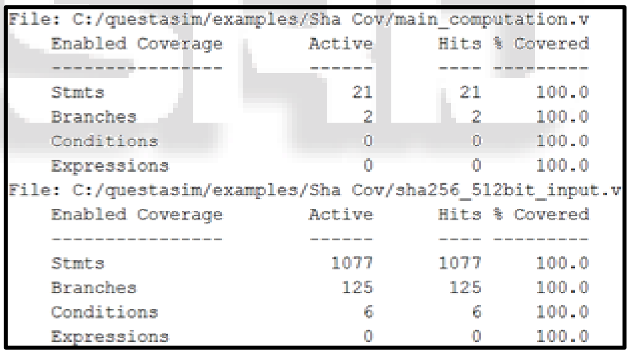
Fig. 3: Scoreboard Result

The figure 4 shows the coverage report, which is generated from tool, in text format which is 100%. This is a code coverage report which includes statements coverage, branch coverage, conditions coverage and expression coverage. Depending up on the design, one can enable/disable the coverage types in tool itself.



```
Coverage Report Summary Data by file
File: C:/questasim/examples/Sha Cov/extension.v
    Enabled Coverage           Active       Hits % Covered
    ----------------           ------       ---- ---------
    Stmts                           3          3     100.0
    Branches                        2          2     100.0
    Conditions                      0          0     100.0
    Expressions                     0          0     100.0
```

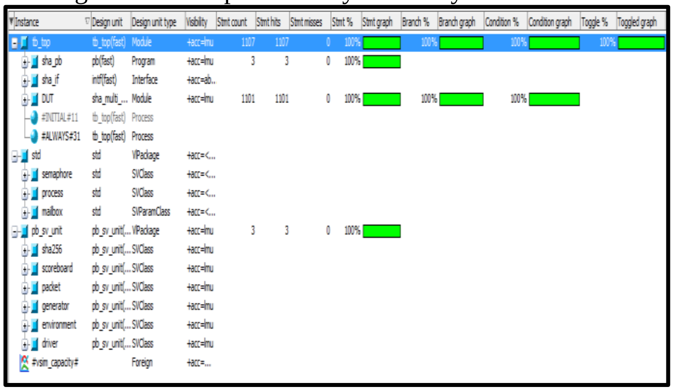
Fig. 4: Coverage Report (Text)



```
File: C:/questasim/examples/Sha Cov/main_computation.v
    Enabled Coverage           Active       Hits % Covered
    ----------------           ------       ---- ---------
    Stmts                          21         21     100.0
    Branches                        2          2     100.0
    Conditions                      0          0     100.0
    Expressions                     0          0     100.0
File: C:/questasim/examples/Sha Cov/sha256_512bit_input.v
    Enabled Coverage           Active       Hits % Covered
    ----------------           ------       ---- ---------
    Stmts                        1077       1077     100.0
    Branches                      125        125     100.0
    Conditions                      6          6     100.0
    Expressions                     0          0     100.0
```

Fig. 5: Coverage Report (Text)

The figure 6 shows the graphical representation of coverage which is 100%. In this figure, toggle coverage is also shown which shows which bits in the RTL have toggled. Toggle coverage is used for power analysis mainly.



Instance	Design unit	Design unit type	Visibility	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt graph	Branch %	Branch graph	Condition %	Condition graph	Toggle %	Toggled graph
tb_top	tb_top(fast)	Module	+acc=lmu	1107	1107	0	100%	🟩	100%	🟩	100%	🟩	100%	🟩
sha_pb	pb(fast)	Program	+acc=lmu	3	3	0	100%	🟩						
sha_if	intf(fast)	Interface	+acc=ab...											
DUT	sha_multi_...	Module	+acc=lmu	1101	1101	0	100%	🟩	100%	🟩	100%	🟩		
#INITIAL#11	tb_top(fast)	Process												
#ALWAYS#31	tb_top(fast)	Process												
std	std	VPackage	+acc=<...											
semaphore	std	SVClass	+acc=<...											
process	std	SVClass	+acc=<...											
mailbox	std	SVParamClass	+acc=<...											
pb_sv_unit	pb_sv_unit(...	VPackage	+acc=lmu	3	3	0	100%	🟩						
sha256	pb_sv_unit(...	SVClass	+acc=lmu											
scoreboard	pb_sv_unit(...	SVClass	+acc=lmu											
packet	pb_sv_unit(...	SVClass	+acc=lmu											
generator	pb_sv_unit(...	SVClass	+acc=lmu											
environment	pb_sv_unit(...	SVClass	+acc=lmu											
driver	pb_sv_unit(...	SVClass	+acc=lmu											
#vsim_capacity#		Foreign	+acc=...											

Figure 6: Coverage Report(Graphically)

The figure 7 shows the waveform of DUT which shows the output when ready signal is asserted. Randomized input (shown in figure 3) is being given to DUT which processes it in the multiple blocks of 512 bits. The number of 512 blocks depends up on the length of message. In this example, two blocks of 512 bits are processed one after another.

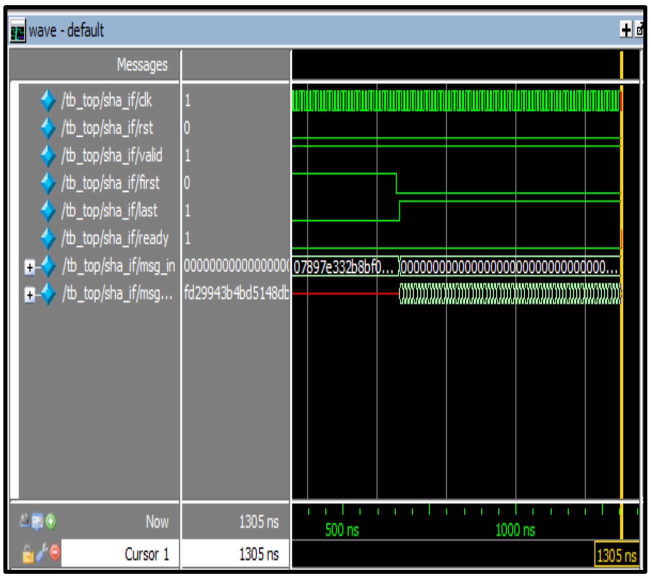


Fig. 7: DUT Waveform

VII. CONCLUSION

In this paper, Randomized test cases are applied to the DUT which gives 100% coverage. The percentage of coverage achieved in this core is shown in simulation results by graph as well as in text format. So, this core has been verified sufficiently with randomized test cases on the Questasim tool.

REFERENCES

[1] National Institute of Standards and Technology, “Secure hash standard, “Federal Information Processing Standards Publications FIPS PUB 180, May, 1993.
[2] National Institute of Standards and Technology, ”Secure hash standard, “Federal Information Processing Standards Publications FIPS PUB 180-1,1995.
[3] National Institute of Standards and Technology, ”Secure hash standard, “Federal Information Processing Standards Publications FIPS PUB 180-2”, 2001.
[4] National Institute of Standards and Technology, ”Secure hash standard, “Federal Information Processing Standards Publications FIPS PUB 180-2”, 2002.
[5] Crowe, F.M. ; Murphy, C.C. ; Marnane, William P., “Optimisation of the SHA-2 Family of Hash Functions on FPGAs”2006,Volume :00
[6] Sklavos, N.; Electr. &Comput. Eng. Dept., Patras Univ., Greece ;Koufopavlou, O., “On the hardware implementations of the SHA-2(256,384,512) hash functions”,2003,Volume :5
[7] Wanzhong Sun ; Inf. Eng. Univ., Zhengzhou ; Hongpeng Guo ; Huilei He ; Zibin Dai, “Design and Optimized Implementation of the SHA-2(256, 384, 512) Hash Algorithms”,2007 pp 858-861
[8] Ling Bai ; Inst. of Microelectron., Tsinghua Univ., Beijing, China ; Shuguo Li, “VLSI Implementation ofHigh-speed SHA-256”, 2009,pp 131-134
[9] Chu-Hsing Lin; Dept. of Comput. Sci., Tunghai Univ., Taichung, Taiwan ; Chen-Yu Lee;Yi-ShiungYeh;Hung-ShengChien,“Generalized Secure Hash Algorithm: SHA-X”,2011,pp 1-4
[10] Sklavos, N., “Towards to SHA-3 Hashing Standard for Secure Communications: On the Hardware Evaluation Development”, 2012,Volume :10,Issue:1
[11] Samir Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, 2nd Edition, Pearson Publisher,2008,Part 2 Advance Verilog Topics, pp 249-274
[12] SwapnajitMittra, Principles Of Verilog Pli, 1st Edition, Kluwer Academic Publishers, 1999.
[13] Verilog code, “https://github.com/rnz/verilog- sha256”