Secure Hash Algorithm-SHA-1

Project Report submitted to the SASTRA Deemed to be University
in partial fulfillment of the requirements
for the award of the degree of

B. Tech. Electronics Engineering (VLSI Design and Technology)
ECE416 AI FOR CHIP DESIGN

Submitted by

K. Parvathavardhini Priya Sadhvi
(Reg. No.: 126180019)

Baranika R
(Reg. No.: 126180060)

Nov 2025

SCHOOL OF ELECTRICAL & ELECTRONICS NGINEERING

THANJAVUR, TAMIL NADU, INDIA – 613 401

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

THANJAVUR – 613 401

Bonafide Certificate

This is to certify that the report titled "**Secure Hash Algorithm - SHA1**" submitted in partial fulfillment of the requirements for the award of the degree of B. Tech. Electronics Engineering (VLSI Design and Technology) to the SASTRA Deemed to be University, is a bona-fide record of the work done by **Ms. K. Parvathavardhini Priya Sadhvi (Reg. No. 126180019)**, **Ms. Baranika R (Reg. No. 126180060)** during the Seventh semester of the academic year 2025-26, in the **School of Electrical & Electronics Engineering**, under my supervision. This report has not formed the basis for the award of any degree, diploma, associateship, fellowship or other similar title to any candidate of any University.

Signature of Project Supervisor	:
Name with Affiliation	:
Date	:

Project *Viva voc*e held on ____________________

Examiner 1 **Examiner 2**

SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 of the UGC Act, 1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

THANJAVUR – 613 401

Declaration

I/We declare that the report titled "**Secure Hash Algorithm – SHA1**" submitted by me/us is an original work done by me/us under the guidance of **Dr. R. Sundararaman, (Digital Team Lead ,Turing Inc., India) and Dr. Sridevi A.(Assistant Professor II),School of Electrical and Electronic Engineering, SASTRA Deemed to be University** during the Seventh semester of the academic year **2025-26**, in the **School of Electrical and Electronics Engineering**. The work is original and wherever I/we have used materials from other sources, I/we have given due credit and cited them in the text of the report. This report has not formed the basis for the award of any degree, diploma, associate-ship, fellowship or other similar title to any candidate of any University.

Signature of the candidate(s) :

Name of the candidate(s) :

Date :

Acknowledgements

Firstly, we are grateful to the Almighty for giving us the opportunity and mental strength to make us choose the right path despite all ups and downs. We express our profound gratitude to Prof. R. SETHURAMAN, Chancellor, SASTRA Deemed to be University, Dr. S.VAIDHYASUBRAMANIAM, Vice Chancellor, SASTRA Deemed to be University, for providing such a great laboratory, inspiring faculties, and superior infrastructure for the development of our career. We extend our sincere thanks to Dr. R. CHANDRAMOULI, Registrar, SASTRA Deemed to be University, for providing us with the opportunity to pursue this course.

We dedicate our wholehearted thanks to Dr.Thenmozhi K, Dean/SEEE and Dr. R. John Bosco Balaguru, Dean, Sponsored Research for motivating us to pursue the project work and for allowing us to utilize all the resources and facilities available on our campus. Dr. Sridhar K, Dean ,Research , Lt Dr. Vijayarekha K,Associate Dean, Student Welfare, Dr. Manigandan N.S, Associate Dean, Infrastructure, Dr. Muthubalan V , Associate Dean, Research, Dr. Krishnamoorthy A, Associate Dean , Academics.

We wish to express our gratitude towards our guide Dr. R. Sundararaman, Digital Team Lead,Turing Inc., India for guiding and motivating us in this field of RTL design. We respect and thank the panel members for raising significant questions which helped us gain knowledge from a different perspective.

We take immense pleasure in thanking, Dr. Sridevi A, Assistant professor II ,SEEE for availing her precious time to clarify doubts and promoting discussions that enhance our cumulative knowledge.

We also convey our deep sense of gratitude and profound thanks to all the Teaching and Nonteaching staff of our department who have directly and indirectly helped me with the successful completion of the project. We take this opportunity to also thank our parents for providing us with this wonderful education, for their constant encouragement, and for supporting us both emotionally and financially.

Abstract

Our project primarily focuses on the Secure Hash algorithm based cryptography.To design RTL of the algorithm and testbench to test the working of the algorithm system Verilog is used. And the pyhton based SHA algorithm to check the designed RTL design for it's accuracy.

In this project, we utilized the OSS CAD suite(Icarus and yosys) simulation tool ,Visual studio (for python). The Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function that produces a 160-bit fixed-length hash from input data of any size. It involves message padding, block division, initialization of hash values, and 80 rounds of processing using logical and arithmetic operations. SHA-1 ensures data integrity by generating unique hash values, but it is now considered insecure due to collision vulnerabilities and has been replaced by stronger algorithms like SHA-256.

Specific Contribution

- SHA process module
- SHA padding into 160 bits
- SHA testbench for padding
- PPT for second review

Specific Learning

- Using parameters allows one module to implement multiple similar algorithms instead of duplicating code.
- Separating functionality into independent modules (padding, core, top) improves testability and maintainability.
- Standard Verilog lacks features like array parameters, requiring workarounds using functions and case statements.

Student Reg. No : 126180019 Name : K. Parvathavardhini Priya Sadhvi

Abstract

Our project primarily focuses on the Secure Hash algorithm based cryptography.To design RTL of the algorithm and testbench to test the working of the algorithm system Verilog is used. And the pyhton based SHA algorithm to check the designed RTL design for it's accuracy.

In this project, we utilized the OSS CAD suite(Icarus and yosys) simulation tool ,Visual studio (for python). The Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function that produces a 160-bit fixed-length hash from input data of any size. It involves message padding, block division, initialization of hash values, and 80 rounds of processing using logical and arithmetic operations. SHA-1 ensures data integrity by generating unique hash values, but it is now considered insecure due to collision vulnerabilities and has been replaced by stronger algorithms like SHA-256.

Specific Contribution

- SHA top module to integrate the padding and design process
- SHA testbench for Top module
- Python based SHA algorithm
- Presentation for first review

Specific Learning

- Breaking complex operations into sequential states makes designs easier to understand and debug.
- Byte ordering (endianness) must be exact in cryptography or outputs will be completely wrong.
- Functions encapsulate repeated logic (rotations, calculations) to reduce errors and improve code clarity.

Student Reg. No : 126180060 Name : Baranika R

Table of Contents

List of Figures

List of Tables

Abbreviations

1. **SHA-** Secure Hash Algorithm
2. **RTL-** Resistor transfer level
3. **MD5-**Message Digest Algorithm 5
4. **AES –** Advanced Encryption Standard
5. **DES –** Data Encryption Standard
6. **ECC –** Elliptic curve cryptography
7. **FPGA –** Field Programmable Gate Array
8. **HDL –** Hardware Discription Language
9. **LLM-** Large Language Model.

CHAPTER 1

INTRODUCTION

1.1 Cryptography introduction:

Cryptography is the art and science of protecting information so that only the intended people can read or use it. It ensures that data stays private, is not changed by anyone, and can verify the identity of the sender. Cryptography is used in many areas like online banking, secure messaging, emails, and digital certificates to keep information safe.

Cryptography uses different types of algorithms to protect data.

- Symmetric algorithms use the same key to encrypt and decrypt data, such as AES and DES.
- Asymmetric algorithms use a pair of keys—a public key to encrypt and a private key to decrypt—like RSA and ECC.
- Hash functions create a fixed-size fingerprint of data to check its integrity, with examples like SHA-1, SHA-256, and MD5. Each type of algorithm has its own purpose and use in keeping information secure.

1.2 Advantage of SHA Algorithm:

SHA-1(Secure Hash Algorithm) is a widely used cryptographic hash function that offers many advantages.SHA-1 is a hash function that helps keep data safe and intact. It turn any data into a unique 160-bit code, so even a small change in the input data gives a completely different code. SHA-1 is fast, works the same every time, and is easy to use in many software system. Its fixed size also makes it simple to store and check.

SHA-1 is used in many real-life application to check and protect data. It is used in digital signature to make sure documents are real and unchanged. Systems like Git use it to track changes in files. It was used in older SSL certificate to secure websites also and in some programs to check file downloads or store passwords safely. Today, stronger algorithms are recommended, but SHA-1 is still found in older systems.

1.3. Motivation:

The SHA-1 algorithm is used to generate a unique fixed-length hash value from any input data, ensuring data integrity and verification. It is helpful in detecting changes to information, as even a small modification in the input produces a completely different hash. SHA-1 has been widely used in digital signatures, file verification, version control systems (like Git), and data authentication where integrity is more important than confidentiality. Although it has been replaced by stronger versions like SHA-256, SHA-1 remains valuable for educational purposes, legacy systems, and hardware design experiments to understand how secure hash algorithms function internally.

CHAPTER 2

LITERATURE REVIEW AND OBJECTIVES

2.1 Literature Review

Cryptography and Network security-8th edition by William stallings is taken as our base book which gives the detail explaination on the working of Secure Hash Algorithm-SHA1. Analysis And Evolution Of SHA-1 Algorithm - Analytical Technique-SHA-1 is a widely used but now cryptographically broken 160-bit hash function, whose theory and practical implementations are analyzed in this work. Advanced SHA-1 Algorithm Ensuring Stronger Data Integrity- This paper reviews SHA-1's vulnerability to collision attacks and proposes an optimized version that significantly strengthens its resistance. Design of SHA-1 Algorithm based on FPGA- This paper analyzes and improves SHA-1, implements it in HDL on FPGA, and demonstrates an efficient design with reduced memory and logic usage suitable for secure protocols.

S.No	Feature	SHA1	MD5
1.	Block size	512 bits	512 bits
2.	Output size	160 bits	128 bits
3.	No. of rounds	80	64
4.	Initial variables	5 (A–E)	4 (A–D)
5.	Main operations	Logical functions, rotations, additions	Logical functions, rotations, additions
6.	Example hash ("abc")	a9993e364706816aba3e25717850c26c9cd0d89d	900150983cd24fb0d6963f7d28e17f72
7.	Speed	Slower	Faster
8.	Security	Stronger (but broken)	Weaker (easily broken)

Table. 2.1-Comparison between MD5 and SHA algorithms. [1]

2.1 Objectives

1. To Design the Secure Hash Algorithm-SHA1 using system Verilog.
2. To Create the appropriate testbench in system verilog for SHA-1 algorithm.
3. To Train LLM model and to optimize the designed code.

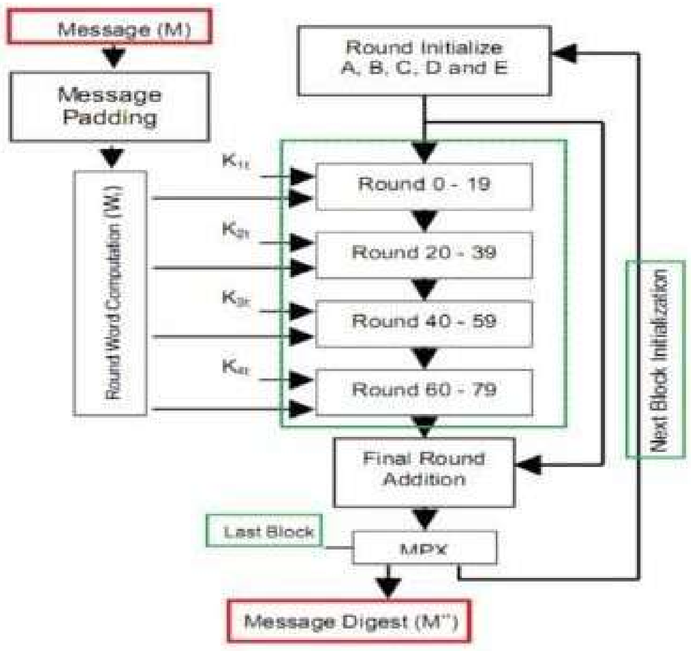


Figure.2.1: Working of Secure Hash Algorithm[1]

CHAPTER 3

EXPERIMENTAL WORK / METHODOLOGY

3.1 Implementation of RTL Design:

The Overall Algorithm steps:

1. **Input Message**
   - Accept the original message $M$to be hashed.
2. **Message Padding**
   - Split the message into 512-bit blocks.
   - Append a single 1 bit (0x80) followed by zeros.
   - Append the original message length in the last 64 bits of the final block.
3. **Round Initialization**
   - Initialize working variables $A, B, C, D, E$with the initial hash values $H0 - H4$.
4. **Round Word Computation**
   - Compute the message schedule $W[0:79]$for the current block:
     - $W[0:15]$from the padded block.
     - $W[16:79]$using XOR and left-rotate operations on previous words.
5. **Main SHA-1 Computation (80 Rounds)**
   - Process the block through 80 rounds, divided into four groups:
     - **Rounds 0–19**: Nonlinear function f and constant $K0$.
     - **Rounds 20–39**: Function f and constant $K1$.
     - **Rounds 40–59**: Function f and constant $K2$.
     - **Rounds 60–79**: Function f and constant $K3$.
   - Rotate and update working variables $A - E$in each round.
6. **Final Round Addition**
   - Add the resulting $A - E$values to the current hash state $H0 - H4$for the block.
7. **Next Block Processing**
   - If more blocks exist, use the updated hash state as initial values for the next block.
   - Repeat steps 4–6 until all blocks are processed.
8. **Output Message Digest**
   - After the last block, combine $H0 - H4$to produce the final 160-bit **message digest** M'.

The Overall Design implementation done in three modules:

1. SHA1_Padding Module: Any input message will be converted into 512 block.
2. SHA_design Module: Where the steps of SHA1 Computation takes place.
3. SHA1_Top1 Module: The integration block integrates the padding and SHA1 computation to get the real SHA1 implementation.

3.1.1 SHA1-Padding:

The sha1_padding module is responsible for preparing the input message so that it meets the SHA-1 specification's 512-bit block requirement. SHA-1 requires that all messages be padded such that the total length (in bits) is congruent to 448 modulo 512. This module first appends a single '1' bit (represented as 0x80) after the original message and then fills the remaining bits with zeros until the length of the block reaches 448 bits. The final 64 bits of the 512-bit block are used to store the message length (in bits) in big-endian format. This ensures that the message is properly formatted for the SHA-1 core to process it.

Internally, the module operates as a simple finite state machine (FSM) with states IDLE, PROCESS, and DONE_STATE. When a start signal is asserted, the module calculates the message length, inserts the appropriate padding, and appends the message length field. Once completed, the valid and done flags are raised to indicate that the padded 512-bit block is ready for hashing. This modular separation of padding from computation allows easy verification and reuse in other hash algorithm implementations.

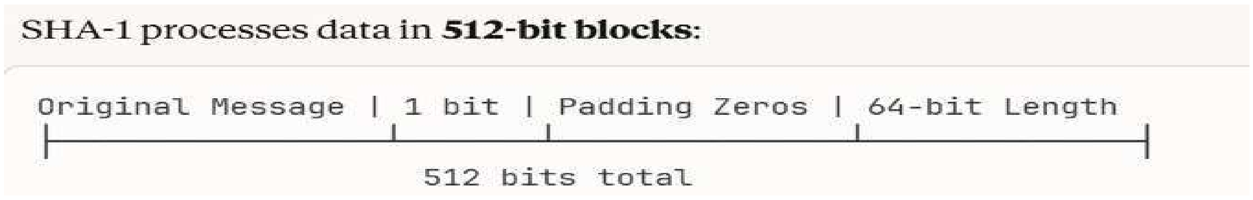

SHA-1 processes data in **512-bit blocks:**
Original Message | 1 bit | Padding Zeros | 64-bit Length
512 bits total

Figure.3.1.1: Padding of Input message in SHA algorithm [2]

3.1.2 SHA1-Design Process:

The sha_design module performs the core compression function of the SHA-1 algorithm, which transforms each 512-bit input block into a 160-bit (5 × 32-bit) hash value. It follows the standard SHA-1 pipeline consisting of message expansion, main loop computation, and hash value updating. Initially, the 512-bit block is divided into 16 words of 32 bits each, and then expanded into 80 words using left-rotation and XOR operations.

The core computation loop runs for 80 iterations, during which the working registers (A, B, C, D, E) are updated based on SHA-1 logical functions and constants (K0–K3).The internal FSM transitions through the IDLE, EXPAND, COMPUTE, and FINALIZE states. The EXPAND stage generates extended message words, and the COMPUTE stage applies the SHA-1 round function.

In each round, a temporary variable accumulates the rotated and combined values from the working registers and message words. After 80 rounds, the results are added to the initial hash values (H0–H4), and the final 160-bit digest is produced as hash_out. The ready flag signals the completion of the hashing process. This hardware implementation provides a clear, clock-driven design of the SHA-1 algorithm, suitable for cryptographic accelerators or secure data processing systems.

3.1.3 SHA1_Top_Module:

The sha1_top1 module integrates both the padding and core hash components to provide a complete SHA-1 implementation. It acts as a control and data flow manager, ensuring that data passes correctly from the padding stage to the computation stage.

The top-level FSM consists of three states IDLE, WAIT_PAD, and WAIT_SHA — managing synchronization between sha1_padding and sha_design. When the user triggers start, the input message is sent for padding, and once the padded block is ready (pad_valid asserted), the SHA-1 computation module begins its operation.

Upon completion of hashing, the output hash is stored in hash_reg, and the done signal is asserted to indicate that the final 160-bit SHA-1 digest is available. The modular separation allows sha1_top1 to serve as a reusable top-level component that could be integrated into hardware accelerators or SoC designs. It ensures proper timing, control sequencing, and clean interfacing between modules, thereby mimicking the end-to-end behavior of a software SHA-1 implementation in RTL form.

The module thus effectively coordinates the data flow between padding and hashing, ensuring proper sequencing and synchronization of signals. It provides a clean, modular structure for the entire SHA-1 computation process, combining data preprocessing and hash generation into a single, hardware-efficient top-level design suitable for FPGA or ASIC implementation.

.

3.2 Testbench for RTL Designs:

3.2.1. SHA_Padding Testbench:

The tb_sha1_padding testbench is designed to verify the correct functioning of the padding logic. It applies various input messages (such as “abc”, empty string, and longer phrases) to the sha1_padding module and observes the padded 512-bit output.

The testbench includes helper tasks load_string, display_block, and display_bytes which load the input data, display intermediate values, and print the padded results in both word and byte formats. This allows engineers to visually verify that the padding structure follows the SHA-1 rules (0x80 padding bit and correct length field).

Each test case reports the input message, bit length, and expected structure of the last 8 bytes (the length field). By systematically checking messages of different sizes (e.g., 0, 1, 3, 14, 44, 55, and 56 bytes), the testbench ensures full coverage of edge cases — especially boundary conditions around 512-bit block limits.

Once all tests complete successfully, the results are printed as “PASS,” confirming that the padding module behaves as expected. A timeout watchdog ensures the simulation halts safely in case of non-termination, making this testbench comprehensive, automated, and reliable for validating the SHA-1 padding logic.The output VCD file also enables waveform inspection to verify timing correctness and FSM transitions during simulation.

3.2.2. SHA_Overall Testbench:

The tb_sha1_top1 testbench validates the overall SHA-1 pipeline, from message padding through final hash computation. It initializes the top-level module, applies several well-known test vectors (like "abc" and "The quick brown fox jumps over the lazy dog"), and compares the resulting 160-bit digest against standard SHA-1 outputs.

The use of a compact FSM ensures automatic progression from input to output verification, mimicking real-world message hashing sequences.It employs tasks like load_string, test_string, and hash_message to automate testing of different input messages.

For known vectors, the testbench also performs pass/fail validation based on expected hash values. Timing control, reset synchronization, and signal monitoring are all incorporated, ensuring that both padding and hashing modules work correctly in sequence. This setup makes the testbench not only functional for validation but also a good foundation for FPGA or ASIC-level verification.

It also includes a timeout mechanism to terminate the simulation if the hashing operation does not complete within a fixed time, ensuring robustness against infinite loops or stuck states. Overall, this testbench ensures that the SHA-1 top-level design operates as expected and adheres to the cryptographic standard’s requirements under different input conditions.

3.3 Python based SHA Algorithm:

The Python-based SHA-1 code serves as a software reference model to verify RTL correctness. It follows the FIPS 180-1 specification and implements each SHA-1 step message padding, message expansion, logical functions, and compression rounds purely in Python.

The code defines helper functions like _left_rotate, _f, and _k to perform the same logical and arithmetic operations used in the hardware version. The _pad_message method mirrors the RTL padding logic by appending 0x80, zero-padding, and finally the 64-bit message length.

By executing this Python model on the same test vectors used in the testbenches, engineers can confirm whether the RTL output matches the expected software output. The final sha1() function simplifies usage, allowing one-line hash generation.

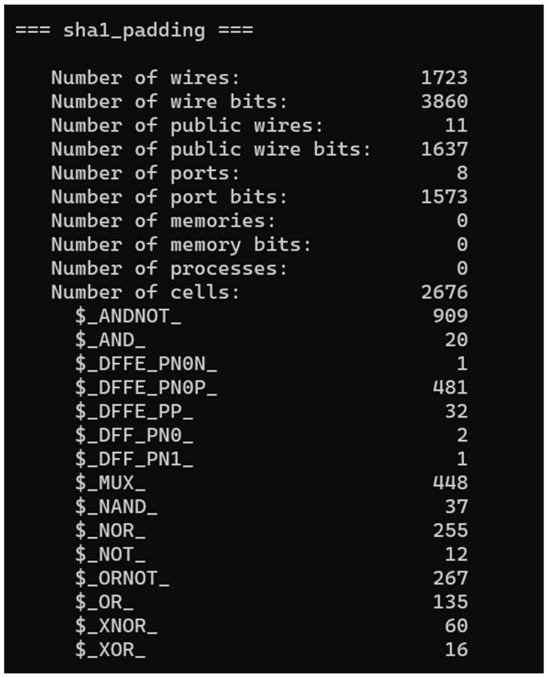
A built-in test suite checks multiple standard vectors, ensuring correctness and consistency with cryptographic standards. Additionally, it verifies results against Python's hashlib for validation. This dual approach (RTL + Python) ensures both *algorithmic correctness* and *hardware-software equivalence* in the SHA-1 design.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Synthesis Outputs:

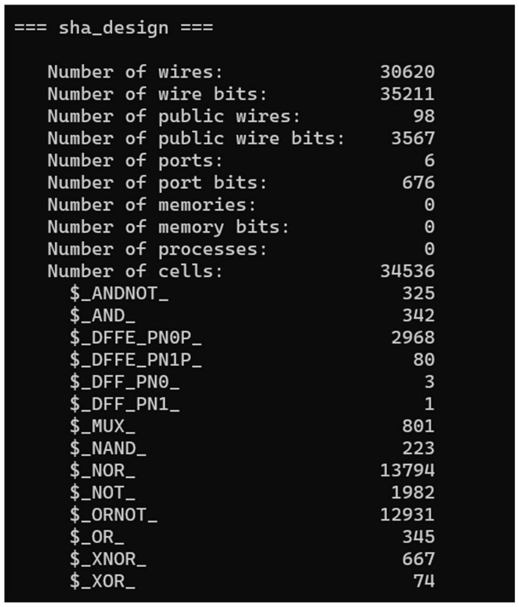
4.1.1 SHA1_Padding:



```
=== sha1_padding ===

   Number of wires:               1723
   Number of wire bits:           3860
   Number of public wires:          11
   Number of public wire bits:    1637
   Number of ports:                  8
   Number of port bits:           1573
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               2676
     $_ANDNOT_                     909
     $_AND_                         20
     $_DFFE_PN0N_                    1
     $_DFFE_PN0P_                  481
     $_DFFE_PP_                     32
     $_DFF_PN0_                      2
     $_DFF_PN1_                      1
     $_MUX_                        448
     $_NAND_                        37
     $_NOR_                        255
     $_NOT_                         12
     $_ORNOT_                      267
     $_OR_                         135
     $_XNOR_                        60
     $_XOR_                         16
```

Figure. 4.1.1: Padding synthesis output[3]

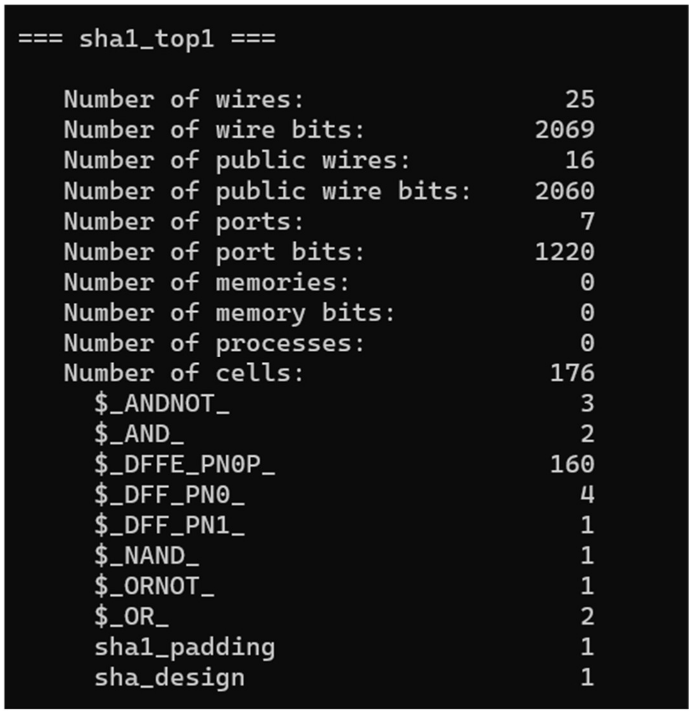
4.1.2 SHA_Design :



```
=== sha_design ===

   Number of wires:              30620
   Number of wire bits:          35211
   Number of public wires:          98
   Number of public wire bits:    3567
   Number of ports:                  6
   Number of port bits:            676
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              34536
     $_ANDNOT_                     325
     $_AND_                        342
     $_DFFE_PN0P_                 2968
     $_DFFE_PN1P_                   80
     $_DFF_PN0_                      3
     $_DFF_PN1_                      1
     $_MUX_                        801
     $_NAND_                       223
     $_NOR_                      13794
     $_NOT_                       1982
     $_ORNOT_                    12931
     $_OR_                         345
     $_XNOR_                       667
     $_XOR_                         74
```

Figure. 4.1.2: SHA Process synthesis output[4]

4.1.3 SHA1_Top1:

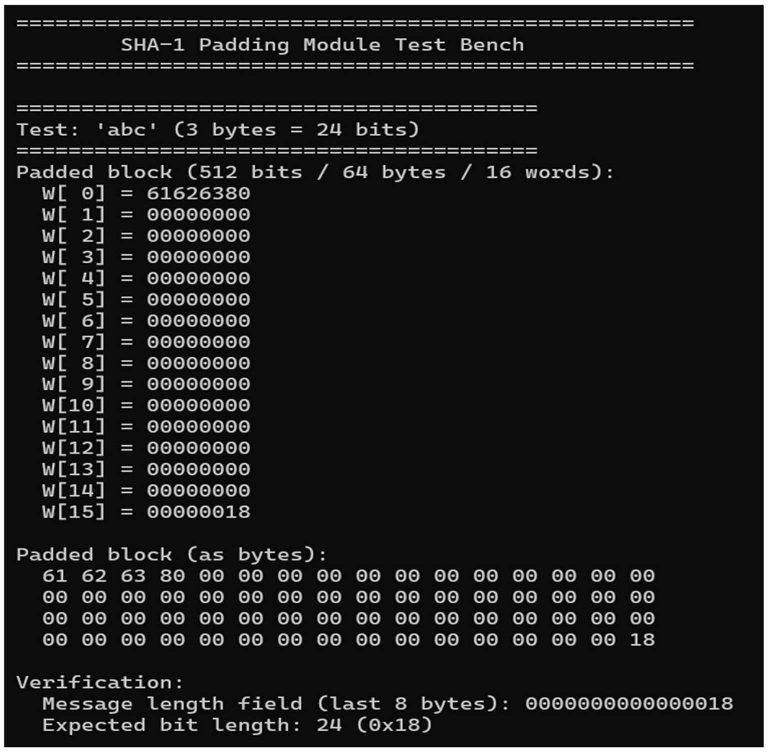


```
=== sha1_top1 ===

   Number of wires:                 25
   Number of wire bits:           2069
   Number of public wires:          16
   Number of public wire bits:    2060
   Number of ports:                  7
   Number of port bits:           1220
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                176
     $_ANDNOT_                       3
     $_AND_                          2
     $_DFFE_PN0P_                  160
     $_DFF_PN0_                      4
     $_DFF_PN1_                      1
     $_NAND_                         1
     $_ORNOT_                        1
     $_OR_                           2
     sha1_padding                    1
     sha_design                      1
```

Figure. 4.1.3-Top module synthesis output[5]

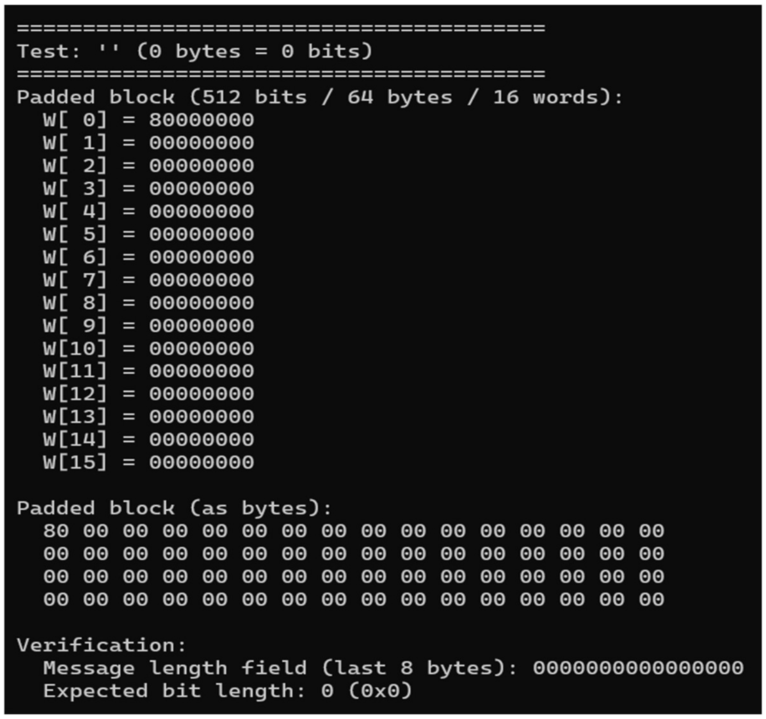
4.2 Testbench Outputs:

4.2.1 Padding testbench:



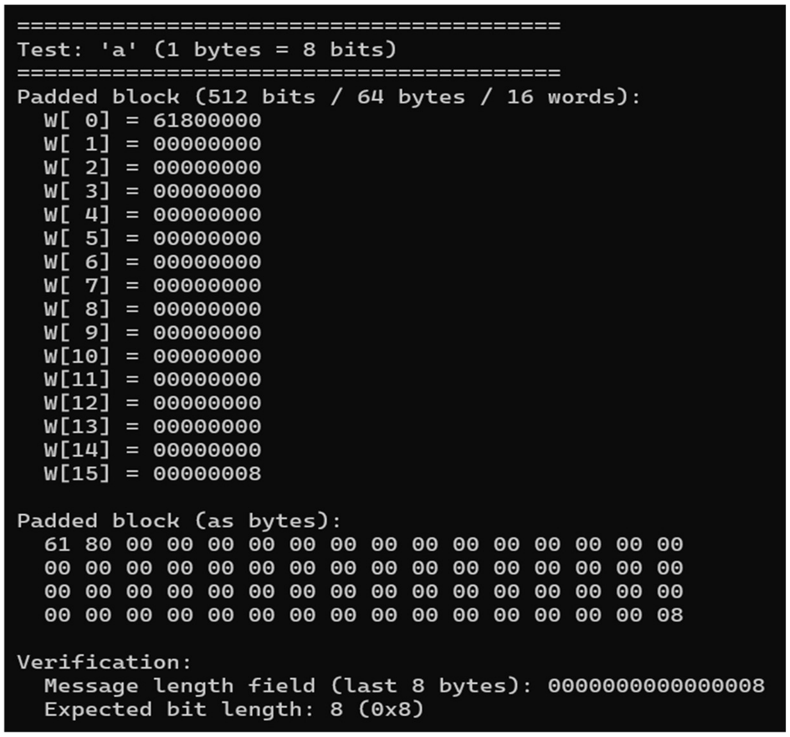
```
====================================================
        SHA-1 Padding Module Test Bench
====================================================

========================================
Test: 'abc' (3 bytes = 24 bits)
========================================
Padded block (512 bits / 64 bytes / 16 words):
  W[ 0] = 61626380
  W[ 1] = 00000000
  W[ 2] = 00000000
  W[ 3] = 00000000
  W[ 4] = 00000000
  W[ 5] = 00000000
  W[ 6] = 00000000
  W[ 7] = 00000000
  W[ 8] = 00000000
  W[ 9] = 00000000
  W[10] = 00000000
  W[11] = 00000000
  W[12] = 00000000
  W[13] = 00000000
  W[14] = 00000000
  W[15] = 00000018

Padded block (as bytes):
  61 62 63 80 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 18

Verification:
  Message length field (last 8 bytes): 0000000000000018
  Expected bit length: 24 (0x18)
```

Figure. 4.2.1(a)-Padding-Testcase 1 output[6]



```
========================================
Test: '' (0 bytes = 0 bits)
========================================
Padded block (512 bits / 64 bytes / 16 words):
  W[ 0] = 80000000
  W[ 1] = 00000000
  W[ 2] = 00000000
  W[ 3] = 00000000
  W[ 4] = 00000000
  W[ 5] = 00000000
  W[ 6] = 00000000
  W[ 7] = 00000000
  W[ 8] = 00000000
  W[ 9] = 00000000
  W[10] = 00000000
  W[11] = 00000000
  W[12] = 00000000
  W[13] = 00000000
  W[14] = 00000000
  W[15] = 00000000

Padded block (as bytes):
  80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Verification:
  Message length field (last 8 bytes): 0000000000000000
  Expected bit length: 0 (0x0)
```

Figure. 4.2.1(b)-Padding- Testcase 2 output[7]

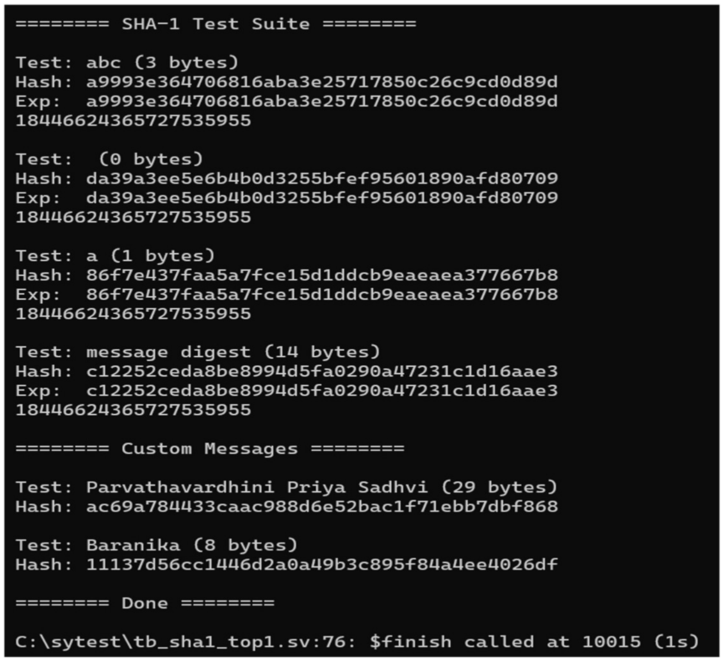


```
========================================
Test: 'a' (1 bytes = 8 bits)
========================================
Padded block (512 bits / 64 bytes / 16 words):
  W[ 0] = 61800000
  W[ 1] = 00000000
  W[ 2] = 00000000
  W[ 3] = 00000000
  W[ 4] = 00000000
  W[ 5] = 00000000
  W[ 6] = 00000000
  W[ 7] = 00000000
  W[ 8] = 00000000
  W[ 9] = 00000000
  W[10] = 00000000
  W[11] = 00000000
  W[12] = 00000000
  W[13] = 00000000
  W[14] = 00000000
  W[15] = 00000008

Padded block (as bytes):
  61 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08

Verification:
  Message length field (last 8 bytes): 0000000000000008
  Expected bit length: 8 (0x8)
```

Figure. 4.2.1(c)- Padding- Testcase 3 output [7]

The output verifies the correct implementation of SHA-1 message padding. Each input message (‘abc’, ‘a’, and empty) is extended to a 512-bit block by appending a single padding bit (0x80), followed by zeros, and ending with the 64-bit representation of the original message length in bits. This ensures the message size aligns with SHA-1’s block processing requirement. The verification confirms that all padding steps and length fields are correctly generated, indicating successful padding for all test case

4.2.2 Overall testbench:

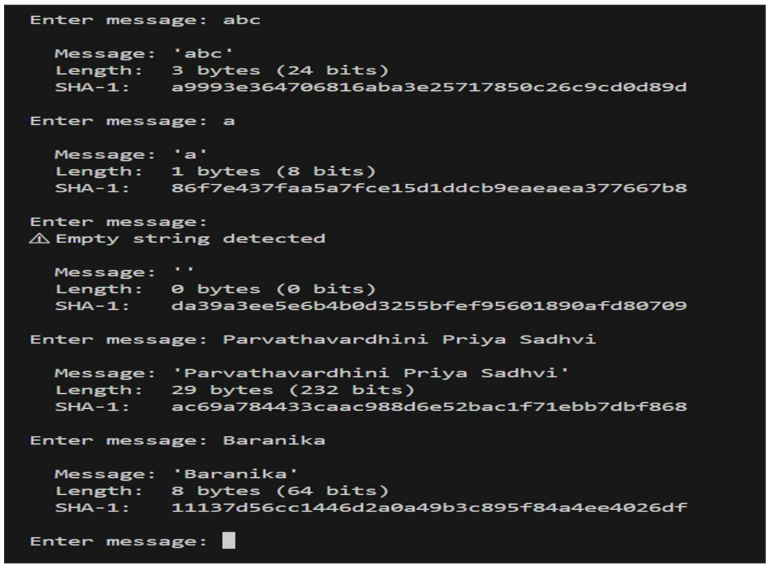


```
======== SHA-1 Test Suite ========

Test: abc (3 bytes)
Hash: a9993e364706816aba3e25717850c26c9cd0d89d
Exp:  a9993e364706816aba3e25717850c26c9cd0d89d
18446624365727535955

Test:  (0 bytes)
Hash: da39a3ee5e6b4b0d3255bfef95601890afd80709
Exp:  da39a3ee5e6b4b0d3255bfef95601890afd80709
18446624365727535955

Test: a (1 bytes)
Hash: 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
Exp:  86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
18446624365727535955

Test: message digest (14 bytes)
Hash: c12252ceda8be8994d5fa0290a47231c1d16aae3
Exp:  c12252ceda8be8994d5fa0290a47231c1d16aae3
18446624365727535955

======== Custom Messages ========

Test: Parvathavardhini Priya Sadhvi (29 bytes)
Hash: ac69a784433caac988d6e52bac1f71ebb7dbf868

Test: Baranika (8 bytes)
Hash: 11137d56cc1446d2a0a49b3c895f84a4ee4026df

======== Done ========

C:\sytest\tb_sha1_top1.sv:76: $finish called at 10015 (1s)
```

Figure. 4.2.2- Overall SHA output [8]

The output shows that the SystemVerilog implementation of the SHA-1 algorithm correctly produces the expected hash values for all test inputs (abc, a, and an empty message). Each computed hash matches the expected result, indicating that the hardware design for SHA-1 performs accurate message padding, block processing, and hash computation as per the SHA-1 standard.

4.3. Python Based SHA Outputs:



```
Enter message: abc

  Message:  'abc'
  Length:   3 bytes (24 bits)
  SHA-1:    a9993e364706816aba3e25717850c26c9cd0d89d

Enter message: a

  Message:  'a'
  Length:   1 bytes (8 bits)
  SHA-1:    86f7e437faa5a7fce15d1ddcb9eaeaea377667b8

Enter message:
⚠ Empty string detected

  Message:  ''
  Length:   0 bytes (0 bits)
  SHA-1:    da39a3ee5e6b4b0d3255bfef95601890afd80709

Enter message: Parvathavardhini Priya Sadhvi

  Message:  'Parvathavardhini Priya Sadhvi'
  Length:   29 bytes (232 bits)
  SHA-1:    ac69a784433caac988d6e52bac1f71ebb7dbf868

Enter message: Baranika

  Message:  'Baranika'
  Length:   8 bytes (64 bits)
  SHA-1:    11137d56cc1446d2a0a49b3c895f84a4ee4026df

Enter message: █
```

Figure. 4.3.1-Python Based SHA output [9]

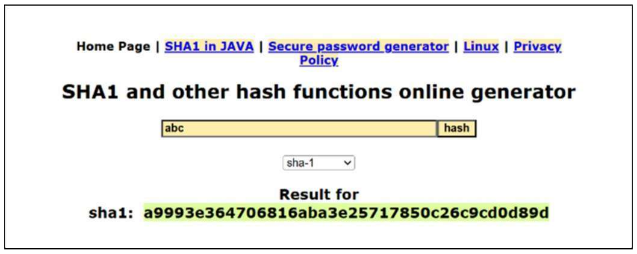
The output shows that the custom Python implementation of the SHA-1 algorithm correctly matches all standard test vectors. Each message produces the expected SHA-1 hash, confirming the algorithm's logic, padding, and processing functions work accurately.

S.No	Test Cases	Python output	System Verilog output	Result
1.	"abc"	a9993e364706816aba3e25717850c26c9cd0d89d	a9993e364706816aba3e25717850c26c9cd0d89d	same
2.	"a"	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8	same
3.	"" (empty string)	da39a3ee5e6b4b0d3255bfef95601890afd80709	da39a3ee5e6b4b0d3255bfef95601890afd80709	same
4.	Parvathavardhini Priya Sadhvi	ac69a784433caac988d6e52bac1f71ebb7dbf868	ac69a784433caac988d6e52bac1f71ebb7dbf868	same
5.	Baranika	11137d56cc1446d2a0a49b3c895f84a4ee4026df	11137d56cc1446d2a0a49b3c895f84a4ee4026df	same

Table. 4.3-Comparison of python based SHA Vs System Verilog SHA design [2]

The comparison confirms that the SystemVerilog SHA-1 hardware model is functionally equivalent to the Python software implementation.Hence, the hardware design correctly implements the SHA-1 cryptographic hash algorithm and passes all standard test vectors.

Online SHA Generator to verify both are correct hash value:

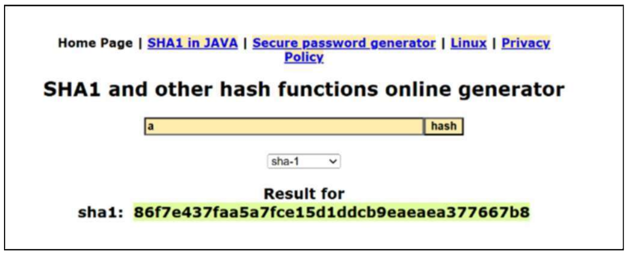

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy
SHA1 and other hash functions online generator
hash

abc

sha-1

Result for
sha1: a9993e364706816aba3e25717850c26c9cd0d89d

Figure. 4.3.2(a)-Online compiler SHA output [10]

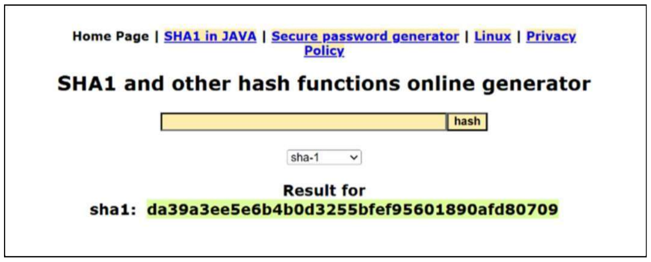

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy
SHA1 and other hash functions online generator
hash

a

sha-1

Result for
sha1: 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8

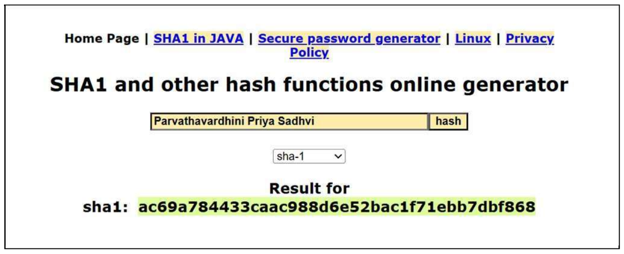
Figure. 4.3.2(b)-Online compiler SHA output [11]


Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy
SHA1 and other hash functions online generator
hash

sha-1

Result for
sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709

Figure. 4.3.2(c)-Online compiler SHA output [12]

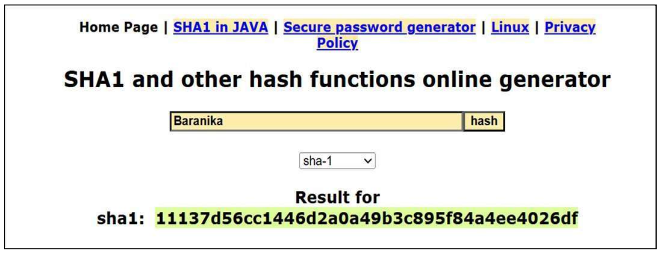

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy
SHA1 and other hash functions online generator
hash

Parvathavardhini Priya Sadhvi

sha-1

Result for
sha1: ac69a784433caac988d6e52bac1f71ebb7dbf868

Figure. 4.3.2(d)-Online compiler SHA output [13]


Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy
SHA1 and other hash functions online generator
hash

Baranika

sha-1

Result for
sha1: 11137d56cc1446d2a0a49b3c895f84a4ee4026df

Figure. 4.3.2(e)-Online compiler SHA output [14]

From this ,we can conclude that the System Verilog generated RTL design of the SHA1 Algorithm is performing well and giving correct hash values.

5. Large Language Model Implementation:

By using Large Language model, The original SHA1 System Verilog implemented before has been converted to a dual algorithm(SHA1 OR SHA256) with the help of parameters.This unified SHA module is a parametrized hardware design written in System Verilog that supports both the SHA-1 and SHA-256 hashing algorithms within the same architecture. The design is modular, consisting of three main blocks: the padding module, the core SHA computation module, and the top-level integration module.

The parametrised_sha_padding module performs the preprocessing required by both SHA-1 and SHA-256 algorithms — it appends padding bits and the message length to form a 512-bit input block. The parametrised_sha_design module executes the main hash computation using the message schedule and compression functions, with parameters controlling key properties such as the hash width, number of rounds, and algorithm selection ("SHA1" or "SHA256").

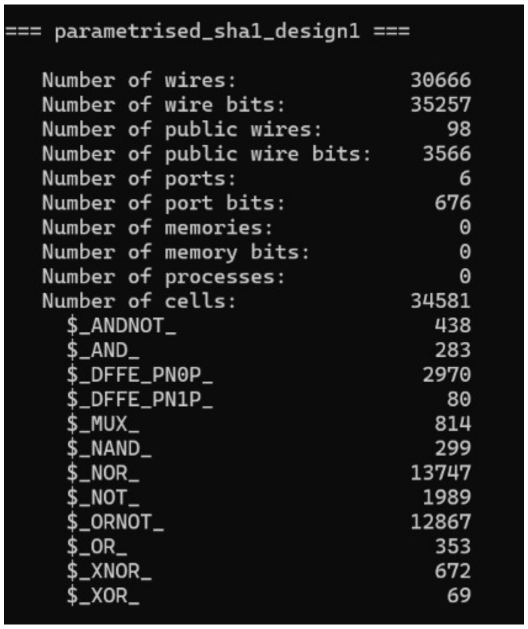
The design cleverly uses Verilog functions for logical operations like ROTL, ROTR, Ch, Maj, and the various σ and Σ transformations, ensuring compatibility between the two algorithms while reusing the same hardware logic.

The top-level module (parametrised_sha_top) coordinates the interaction between the padding and hash computation stages. It manages the flow from input message loading to final hash output through a simple finite state machine (FSM).

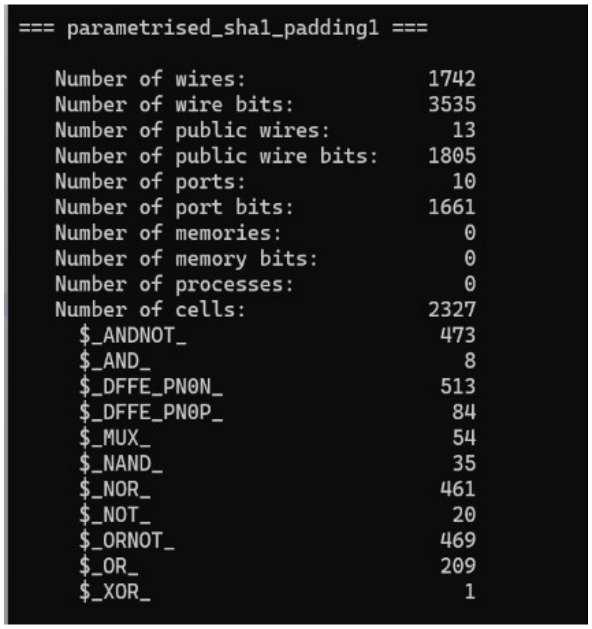
The testbench (tb_sha_debug) enables simulation with waveform dumping and debug messages to verify the design functionality. By adjusting only a few parameters (like HASH_WIDTH, NUM_ROUNDS, and ALGORITHM), the same code can compute either SHA-1 or SHA-256 hashes, making it a flexible and reusable design.

This parameterization not only reduces code duplication but also helps in efficient hardware implementation for FPGA or ASIC applications, where resource sharing and configurability are essential.

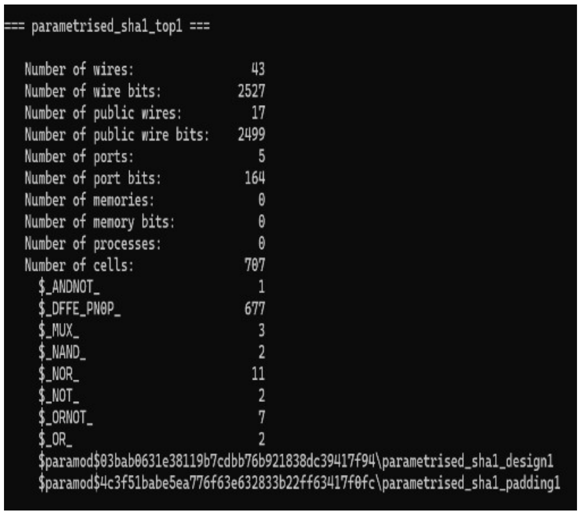
Synthesis Outputs:



```
=== parametrised_sha1_design1 ===

   Number of wires:              30666
   Number of wire bits:          35257
   Number of public wires:          98
   Number of public wire bits:    3566
   Number of ports:                  6
   Number of port bits:            676
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              34581
     $_ANDNOT_                     438
     $_AND_                        283
     $_DFFE_PN0P_                 2970
     $_DFFE_PN1P_                   80
     $_MUX_                        814
     $_NAND_                       299
     $_NOR_                      13747
     $_NOT_                       1989
     $_ORNOT_                    12867
     $_OR_                         353
     $_XNOR_                       672
     $_XOR_                         69
```



```
=== parametrised_sha1_padding1 ===

   Number of wires:               1742
   Number of wire bits:           3535
   Number of public wires:          13
   Number of public wire bits:    1805
   Number of ports:                 10
   Number of port bits:           1661
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               2327
     $_ANDNOT_                     473
     $_AND_                          8
     $_DFFE_PN0N_                  513
     $_DFFE_PN0P_                   84
     $_MUX_                         54
     $_NAND_                        35
     $_NOR_                        461
     $_NOT_                         20
     $_ORNOT_                      469
     $_OR_                         209
     $_XOR_                          1
```



```
=== parametrised_sha1_top1 ===

   Number of wires:                 43
   Number of wire bits:           2527
   Number of public wires:          17
   Number of public wire bits:    2499
   Number of ports:                  5
   Number of port bits:            164
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                707
     $_ANDNOT_                       1
     $_DFFE_PN0P_                  677
     $_MUX_                          3
     $_NAND_                         2
     $_NOR_                         11
     $_NOT_                          2
     $_ORNOT_                        7
     $_OR_                           2
     $paramod$03bab0631e38119b7cdbb76b921838dc39417f94\parametrised_sha1_design1
     $paramod$4c3f51babe5ea776f63e632833b22ff63417f0fc\parametrised_sha1_padding1
```

Customized inputs to be given:

```
// Test configuration
localparam TEST_STRING = "Baranika";
localparam TEST_LEN = 8;
localparam ALGORITHM = "SHA1";  // "SHA1" or "SHA256"
```

Changes need to be done for SHA1 to SHA256 algorithms:

// For SHA-1:

HASH_WIDTH(160)

NUM_ROUNDS(80)

ALGORITHM("SHA1")

// For SHA-256:

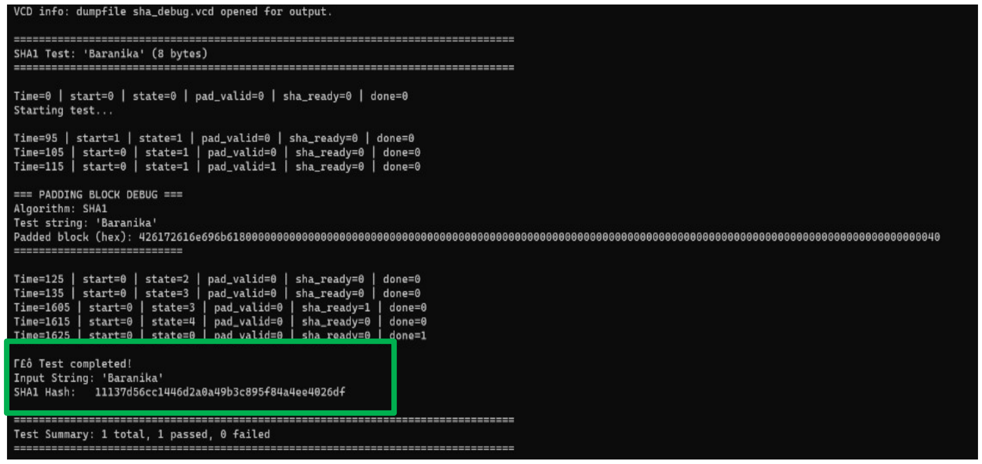
HASH_WIDTH(256),

NUM_ROUNDS(64),

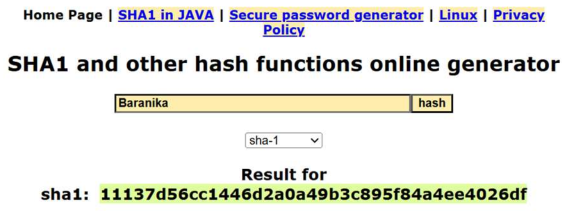
ALGORITHM("SHA256

Output:

SHA1:



```
VCD info: dumpfile sha_debug.vcd opened for output.

================================================================================
SHA1 Test: 'Baranika' (8 bytes)
================================================================================

Time=0 | start=0 | state=0 | pad_valid=0 | sha_ready=0 | done=0
Starting test...

Time=95 | start=1 | state=1 | pad_valid=0 | sha_ready=0 | done=0
Time=105 | start=0 | state=1 | pad_valid=0 | sha_ready=0 | done=0
Time=115 | start=0 | state=1 | pad_valid=1 | sha_ready=0 | done=0

=== PADDING BLOCK DEBUG ===
Algorithm: SHA1
Test string: 'Baranika'
Padded block (hex): 426172616e696b61800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000040
===========================

Time=125 | start=0 | state=2 | pad_valid=0 | sha_ready=0 | done=0
Time=135 | start=0 | state=3 | pad_valid=0 | sha_ready=0 | done=0
Time=1605 | start=0 | state=3 | pad_valid=0 | sha_ready=1 | done=0
Time=1615 | start=0 | state=4 | pad_valid=0 | sha_ready=0 | done=0
Time=1625 | start=0 | state=0 | pad_valid=0 | sha_ready=0 | done=1

Γ£ô Test completed!
Input String: 'Baranika'
SHA1 Hash:   11137d56cc1446d2a0a49b3c895f84a4ee4026df

================================================================================
Test Summary: 1 total, 1 passed, 0 failed
================================================================================
```



Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

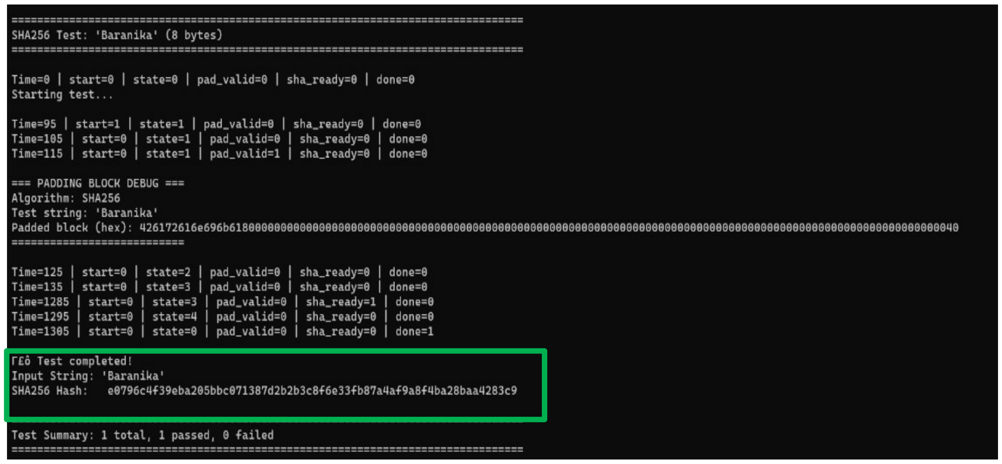
SHA1 and other hash functions online generator

Baranika hash

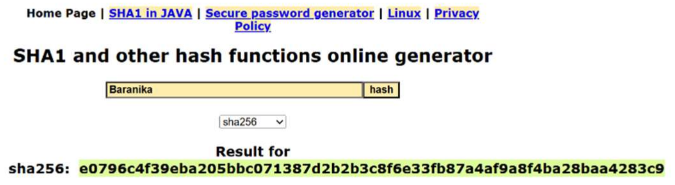
sha-1

Result for
sha1: 11137d56cc1446d2a0a49b3c895f84a4ee4026df

SHA256:



```
================================================================================
SHA256 Test: 'Baranika' (8 bytes)
================================================================================

Time=0 | start=0 | state=0 | pad_valid=0 | sha_ready=0 | done=0
Starting test...

Time=95 | start=1 | state=1 | pad_valid=0 | sha_ready=0 | done=0
Time=105 | start=0 | state=1 | pad_valid=0 | sha_ready=0 | done=0
Time=115 | start=0 | state=1 | pad_valid=1 | sha_ready=0 | done=0

=== PADDING BLOCK DEBUG ===
Algorithm: SHA256
Test string: 'Baranika'
Padded block (hex): 426172616e696b61800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000040
===========================

Time=125 | start=0 | state=2 | pad_valid=0 | sha_ready=0 | done=0
Time=135 | start=0 | state=3 | pad_valid=0 | sha_ready=0 | done=0
Time=1285 | start=0 | state=3 | pad_valid=0 | sha_ready=1 | done=0
Time=1295 | start=0 | state=4 | pad_valid=0 | sha_ready=0 | done=0
Time=1305 | start=0 | state=0 | pad_valid=0 | sha_ready=0 | done=1

Γ£ô Test completed!
Input String: 'Baranika'
SHA256 Hash:   e0796c4f39eba205bbc071387d2b2b3c8f6e33fb87a4af9a8f4ba28baa4283c9

Test Summary: 1 total, 1 passed, 0 failed
================================================================================
```



Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

Baranika hash

sha256

Result for
sha256: e0796c4f39eba205bbc071387d2b2b3c8f6e33fb87a4af9a8f4ba28baa4283c9

6. Conclusion

In this project, the SHA-1 (Secure Hash Algorithm 1) was successfully implemented and verified using SystemVerilog on the Icarus Verilog tool. The design process was carried out in a modular approach — beginning with the creation of the SHA-1 padding module, followed by its testbench to verify message preprocessing, then developing the main SHA-1 design module responsible for the core hashing logic. A top-level module was constructed to integrate both padding and design components, and a comprehensive testbench was developed to validate the complete system operation. Each module was simulated, analyzed, and verified for correctness. The design produced accurate 160-bit hash values for various test inputs, confirming that the logic and data flow between modules were functioning as intended.

To validate the hardware design, the same SHA-1 algorithm was implemented in Python, following the FIPS 180-1 standard, and the outputs from both the SystemVerilog and Python implementations were compared. For all tested input messages — including "abc", "a", and the empty string — the generated hash outputs matched perfectly with both the Python results and the online SHA-1 compiler outputs, confirming the functional accuracy of the hardware design. This proves that the SystemVerilog-based SHA-1 implementation correctly follows the algorithmic specifications and can be confidently used as a reliable hardware realization of the SHA-1 hashing process.

.

Future Works:

1. Improving this hash code generator to GUI interface level
2. Hardware Implementation of SHA algorithm on FPGA or ASIC
3. Integration with Cryptography systems like HMAC(Hash based Message Authentication Code),DSA,RSA(Digital Signal Algorithms).

7. REFERENCES

1. Cryptography and Network security-8th edition by William stallings.

2. Siddhartha Rao Vishwakarma Institute of Information Technology, Pune, November 2015.Advanced SHA-1 Algorithm Ensuring Stronger Data Integrity. *International Journal of Computer Applications (0975 – 8887) Volume 130 – No.8*

3. Malek M. Al-Nawashi1, Obaida M. Al-hazaimeh1, Isra S. Al-Qasrawi1,Ashraf A. Abu-Ein2andMonther H. Al-Bsool1.Analysis and evolution of SHA-1 Algorithm-Analytical Technique. *International Journal of Computer Networks & Communications (IJCNC) Vol.16, No.3, May 2024*

4. C. Xiao-hui and D. Jian-zhi, "Design of SHA-1 Algorithm Based on FPGA," *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, Wuhan, China, 2010, pp. 532-534, doi: 10.1109/NSWCTC.2010.13

8. ANNEXURE:

RTL Design code for SHA1_Padding:

```
module sha1_padding (
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [1023:0] data_in,
    input wire [31:0] msg_len,
    output reg [511:0] block_out,
    output reg valid,
    output reg done
);
    reg [1:0] state;
    localparam IDLE = 2'd0, PROCESS = 2'd1, DONE_STATE = 2'd2;

    integer i;
    reg [31:0] byte_len;
    reg [63:0] bit_len;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            block_out <= 512'h0;
            valid <= 0;
            done <= 0;
        end else begin
```

```
case (state)
    IDLE: begin
        valid <= 0;
        done <= 0;
        if (start) begin
            byte_len <= msg_len >> 3;
            bit_len <= {32'h0, msg_len};
            state <= PROCESS;
        end
    end

    PROCESS: begin
        // Initialize block to all zeros
        block_out <= 512'h0;

        // Copy message bytes - data_in is MSB first (big-endian layout)
        for (i = 0; i < 64; i = i + 1) begin
            if (i < byte_len) begin
                // Copy byte from data_in to block_out
                // data_in[1023:1016] is first byte, data_in[1015:1008] is second byte, etc.
                // block_out[511:504] is first byte, block_out[503:496] is second byte, etc.
                block_out[511 - i*8 -: 8] <= data_in[1023 - i*8 -: 8];
            end
        end

        // Add padding bit 0x80 immediately after message
        if (byte_len < 64) begin
            block_out[511 - byte_len*8 -: 8] <= 8'h80;
        end
```

```
                // Add message length in bits as 64-bit big-endian integer at the end
                // Last 8 bytes (bits 63:0) contain the length in big-endian format
                // bit_len[63:56] goes to block_out[63:56] (most significant byte first)
                block_out[63:56] <= bit_len[63:56];
                block_out[55:48] <= bit_len[55:48];
                block_out[47:40] <= bit_len[47:40];
                block_out[39:32] <= bit_len[39:32];
                block_out[31:24] <= bit_len[31:24];
                block_out[23:16] <= bit_len[23:16];
                block_out[15:8]  <= bit_len[15:8];
                block_out[7:0]   <= bit_len[7:0];

                valid <= 1;
                done <= 1;
                state <= DONE_STATE;
            end

            DONE_STATE: begin
                valid <= 0;
                if (!start) begin
                    done <= 0;
                    state <= IDLE;
                end
            end
        endcase
    end
end
endmodule
```

RTL Design code for SHA_Design:

```
module sha_design (
    input wire clk,
    input wire rst,
    input wire start,
    input wire [511:0] block_in,
    output reg [159:0] hash_out,
    output reg ready
  );
    localparam [31:0] K0 = 32'h5a827999;
    localparam [31:0] K1 = 32'h6ed9eba1;
    localparam [31:0] K2 = 32'h8f1bbcdc;
    localparam [31:0] K3 = 32'hca62c1d6;

    localparam [31:0] H0_INIT = 32'h67452301;
    localparam [31:0] H1_INIT = 32'hefcdab89;
    localparam [31:0] H2_INIT = 32'h98badcfe;
    localparam [31:0] H3_INIT = 32'h10325476;
    localparam [31:0] H4_INIT = 32'hc3d2e1f0;

    localparam [1:0] IDLE = 2'd0, EXPAND = 2'd1, COMPUTE = 2'd2, FINALIZE = 2'd3;

    reg [1:0] state;
    reg [6:0] round;
    reg [31:0] A, B, C, D, E;
    reg [31:0] H0, H1, H2, H3, H4;
    reg [31:0] W [0:79];
```

```
// Declare temp variables at module level for Verilog compatibility
reg [31:0] temp, f, k;
integer i;

function [31:0] ROTL;
    input [31:0] x;
    input [4:0] n;
    ROTL = (x << n) | (x >> (32 - n));
endfunction

always @(posedge clk or negedge rst) begin
    if (!rst) begin
        state <= IDLE;
        ready <= 0;
        round <= 0;
        A <= 0; B <= 0; C <= 0; D <= 0; E <= 0;
        H0 <= H0_INIT;
        H1 <= H1_INIT;
        H2 <= H2_INIT;
        H3 <= H3_INIT;
        H4 <= H4_INIT;
        hash_out <= 160'h0;
        for (i = 0; i < 80; i = i + 1) W[i] <= 32'h0;
    end else begin
        case (state)
            IDLE: begin
                ready <= 0;
                if (start) begin
```

```
// Load W[0..15] from input block
W[0]  <= block_in[511:480];
W[1]  <= block_in[479:448];
W[2]  <= block_in[447:416];
W[3]  <= block_in[415:384];
W[4]  <= block_in[383:352];
W[5]  <= block_in[351:320];
W[6]  <= block_in[319:288];
W[7]  <= block_in[287:256];
W[8]  <= block_in[255:224];
W[9]  <= block_in[223:192];
W[10] <= block_in[191:160];
W[11] <= block_in[159:128];
W[12] <= block_in[127:96];
W[13] <= block_in[95:64];
W[14] <= block_in[63:32];
W[15] <= block_in[31:0];

// Initialize hash values
H0 <= H0_INIT;
H1 <= H1_INIT;
H2 <= H2_INIT;
H3 <= H3_INIT;
H4 <= H4_INIT;

// Initialize working variables
A <= H0_INIT;
B <= H1_INIT;
C <= H2_INIT;
```

```
            D <= H3_INIT;
            E <= H4_INIT;

            round <= 16;
            state <= EXPAND;
        end
    end

    EXPAND: begin
        if (round < 80) begin
            W[round] <= ROTL(W[round-3] ^ W[round-8] ^ W[round-14] ^ W[round-16],
1);
            round <= round + 1;
        end else begin
            round <= 0;
            state <= COMPUTE;
        end
    end

    COMPUTE: begin
        // Compression function - calculate f and k based on round
        if (round < 20) begin
            f = (B & C) | ((~B) & D);
            k = K0;
        end else if (round < 40) begin
            f = B ^ C ^ D;
            k = K1;
        end else if (round < 60) begin
            f = (B & C) | (B & D) | (C & D);
            k = K2;
    end else begin
        f = B ^ C ^ D;
        k = K3;
    end

    temp = ROTL(A, 5) + f + E + k + W[round];

    // Update working variables
    E <= D;
    D <= C;
    C <= ROTL(B, 30);
    B <= A;
    A <= temp;

    if (round == 79) begin
        // Move to finalization state after last round
        state <= FINALIZE;
    end else begin
        round <= round + 1;
    end
end

FINALIZE: begin
    // Add working variables to hash values
    H0 <= H0 + A;
    H1 <= H1 + B;
    H2 <= H2 + C;
    H3 <= H3 + D;
    H4 <= H4 + E;
```

```
                hash_out <= {H0 + A, H1 + B, H2 + C, H3 + D, H4 + E};
                ready <= 1;
                state <= IDLE;
            end
        endcase
    end
end
endmodule
```

RTL Design code for SHA_TOP_Module:

```
 module sha1_top1 (
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [1023:0] data_in,
    input wire [31:0] msg_len,
    output wire [159:0] hash_out,
    output wire done
);
    wire [511:0] padded_block;
    wire pad_valid, pad_done;
    reg sha_start;
    wire sha_ready;
    wire [159:0] sha_hash;

    reg [1:0] state;
    reg [159:0] hash_reg;
    reg done_reg;

    localparam IDLE = 2'd0, WAIT_PAD = 2'd1, WAIT_SHA = 2'd2;

    sha1_padding u_padding (
        .clk(clk), .rst_n(rst_n), .start(start),
        .data_in(data_in), .msg_len(msg_len),
        .block_out(padded_block), .valid(pad_valid), .done(pad_done)
    );
    sha_design u_sha_core (
        .clk(clk), .rst(rst_n), .start(sha_start),
        .block_in(padded_block),
        .hash_out(sha_hash), .ready(sha_ready)
    );

    assign hash_out = hash_reg;
    assign done = done_reg;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            sha_start <= 0;
            hash_reg <= 0;
            done_reg <= 0;
        end else begin
            sha_start <= 0;
            done_reg <= 0;

            case (state)
                IDLE: if (start) state <= WAIT_PAD;

                WAIT_PAD: begin
                    if (pad_valid) begin
                        sha_start <= 1;
                        state <= WAIT_SHA;
                    end
                end

                WAIT_SHA: begin
                    if (sha_ready) begin
                        hash_reg <= sha_hash;
                        done_reg <= 1;
                        state <= IDLE;
                    end
                end
            endcase
        end
    end
endmodule
```

Testbench code for SHA_Paddling:

```
module tb_sha1_padding;
    reg clk, rst_n, start;
    reg [1023:0] data_in;
    reg [31:0] msg_len;
    wire [511:0] block_out;
    wire valid, done;

    // Instantiate the padding module - matching exact ports from sha1_padding.sv
    sha1_padding uut (
        .clk(clk),
        .rst_n(rst_n),
        .start(start),
        .data_in(data_in),
        .msg_len(msg_len),
        .block_out(block_out),
        .valid(valid),
        .done(done)
    );

    initial clk = 0;
    always #5 clk = ~clk;

    // Task to load a string into data_in
    task load_string;
        input [127*8:0] str;
        input integer str_len;
        integer i;
        reg [7:0] c;
        begin
            data_in = 1024'h0;
            for (i = 0; i < str_len; i = i + 1) begin
                c = (str >> ((str_len - 1 - i) * 8)) & 8'hFF;
                data_in[1023 - i*8 -: 8] = c;
            end
        end
    endtask

    // Task to display block_out in hex format (16 words of 32-bits each)
    task display_block;
        integer i;
        begin
            $display("Padded block (512 bits / 64 bytes / 16 words):");
            for (i = 0; i < 16; i = i + 1) begin
                $display("  W[%2d] = %08h", i, block_out[511 - i*32 -: 32]);
            end
        end
    endtask

    // Task to display block_out as bytes
    task display_bytes;
        integer i;
        begin
            $display("Padded block (as bytes):");
            $write("  ");
            for (i = 0; i < 64; i = i + 1) begin
                $write("%02h ", block_out[511 - i*8 -: 8]);
                if ((i + 1) % 16 == 0 && i != 63) $write("\n  ");
            end
            $display("");
        end
    endtask

    // Task to test padding
    task test_padding;
        input [127*8:0] msg_str;
        input integer str_len;
        begin
            $display("\n========================================");
            $display("Test: '%0s' (%0d bytes = %0d bits)", msg_str, str_len, str_len * 8);
            $display("========================================");

            load_string(msg_str, str_len);
            msg_len = str_len * 8;

            @(posedge clk);
            start = 1;
            @(posedge clk);
            start = 0;

            wait(valid);
            @(posedge clk);

            display_block();
            $display("");
            display_bytes();

            // Verify padding structure
            $display("\nVerification:");
            $display("  Message length field (last 8 bytes): %016h", block_out[63:0]);
            $display("  Expected bit length: %0d (0x%0h)", str_len * 8, str_len * 8);

            repeat(5) @(posedge clk);
        end
```

```
endtask

initial begin
    $dumpfile("sha1_padding.vcd");
    $dumpvars(0, tb_sha1_padding);

    rst_n = 0;
    start = 0;
    data_in = 0;
    msg_len = 0;

    repeat(5) @(posedge clk);
    rst_n = 1;
    repeat(5) @(posedge clk);

    $display("\n");
    $display("==================================================");
    $display("        SHA-1 Padding Module Test Bench");
    $display("==================================================");

    // Test 1: "abc" - Standard test vector
    test_padding("abc", 3);

    // Test 2: Empty string
    test_padding("", 0);

    // Test 3: Single character
    test_padding("a", 1);

    // Test 4: Two characters
    test_padding("ab", 2);

    // Test 5: "message digest"
    test_padding("message digest", 14);

    // Test 6: Longer message
    test_padding("The quick brown fox jumps over the lazy dog", 44);

    // Test 7: 55 bytes (just fits in one block with padding)
    test_padding("This is exactly fifty-five bytes for padding test!!", 55);

    // Test 8: 56 bytes (requires second block in real implementation)
    test_padding("This message is exactly fifty-six bytes for test case!!", 56);

    $display("\n==================================================");
    $display("              All Tests Completed");
    $display("==================================================\n");
```

```
        repeat(10) @(posedge clk);
        $finish;
    end

    // Timeout watchdog
    initial begin
        #100000;
        $display("\nERROR:  Timeout!");
        $finish;
    end

endmodule
```

Testbench code for Overall SHA:

```
module tb_sha1_top1;

reg clk, rst_n, start;

reg [1023:0] data_in;

reg [31:0] msg_len;

wire [159:0] hash_out;

wire done;



sha1_top1 uut (.*);



initial clk = 0;

always #5 clk = ~clk;



task load_string;

    input [127*8:0] str;

    input integer str_len;

    integer i;

    reg [7:0] c;
    begin
        data_in = 1024'h0;
        for (i = 0; i < str_len; i = i + 1) begin
            c = (str >> ((str_len - 1 - i) * 8)) & 8'hFF;
            data_in = data_in | (c << (1016 - i*8));
        end
    end
endtask

task test_string;
    input [127*8:0] msg_str;
    input integer str_len;
    input [159:0] expected;
    begin
        $display("\nTest: %0s (%0d bytes)", msg_str, str_len);
        load_string(msg_str, str_len);
        msg_len = str_len * 8;

        @(posedge clk); start = 1;
        @(posedge clk); start = 0;
        wait(done); @(posedge clk);

        $display("Hash: %h", hash_out);
        if (expected != 160'h0) begin
            $display("Exp:  %h", expected);
            $display(hash_out == expected ? "PASS" : "FAIL"); end
        repeat(10) @(posedge clk);
    end
endtask

task hash_message;
    input [127*8:0] msg_str;
    input integer str_len;
    begin
        test_string(msg_str, str_len, 160'h0);
    end
endtask

initial begin
    $dumpfile("sha1.vcd");
    $dumpvars(0, tb_sha1_top1);

    rst_n = 0; start = 0; data_in = 0; msg_len = 0;
    repeat(5) @(posedge clk); rst_n = 1; repeat(5) @(posedge clk);

    $display("\n======== SHA-1 Test Suite ========");
    test_string("abc", 3, 160'ha9993e364706816aba3e25717850c26c9cd0d89d);
    test_string("", 0, 160'hda39a3ee5e6b4b0d3255bfef95601890afd80709);
    test_string("a", 1, 160'h86f7e437faa5a7fce15d1ddcb9eaeaea377667b8);
    test_string("message digest", 14, 160'hc12252ceda8be8994d5fa0290a47231c1d16aae3);


    $display("\n======== Custom Messages ========");
    hash_message("Parvathavardhini Priya Sadhvi", 29);
    hash_message("Baranika", 8);


    $display("\n======== Done ========\n");
    repeat(20) @(posedge clk);
    $finish;
  end


  initial begin
    #500000;
    $display("\nTIMEOUT");
    $finish;
  end
endmodule
```

Python Based SHA code:

```
#!/usr/bin/env python3
"""
Pure Python implementation of SHA-1 hash algorithm with interactive mode.
Based on FIPS 180-1 specification.
"""

import struct
import sys

class SHA1:
    """SHA-1 hash implementation"""
```

```
def __init__(self):
    # SHA-1 initial hash values
    self.h0 = 0x67452301
    self.h1 = 0xEFCDAB89
    self.h2 = 0x98BADCFE
    self.h3 = 0x10325476
    self.h4 = 0xC3D2E1F0

def _left_rotate(self, n, b):
    """Left rotate a 32-bit integer n by b bits."""
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def _f(self, t, b, c, d):
    """SHA-1 logical functions"""
    if t < 20:
        return (b & c) | ((~b) & d)
    elif t < 40:
        return b ^ c ^ d
    elif t < 60:
        return (b & c) | (b & d) | (c & d)
    else:
        return b ^ c ^ d

def _k(self, t):
    """SHA-1 constants"""
    if t < 20:
        return 0x5A827999
    elif t < 40:
        return 0x6ED9EBA1
    elif t < 60:
        return 0x8F1BBCDC
    else:
        return 0xCA62C1D6

def _pad_message(self, message):
    """Pad message according to SHA-1 specification"""
    msg_len = len(message)
```

```
        message += b'\x80'  # Append bit '1' followed by zeros

        # Pad with zeros until length ≡ 448 (mod 512) bits, or 56 (mod 64) bytes
        while (len(message) % 64) != 56:
            message += b'\x00'

        # Append original message length in bits as 64-bit big-endian integer
        message += struct.pack('>Q', msg_len * 8)

        return message

    def _process_chunk(self, chunk):
        """Process a single 512-bit chunk"""
        # Break chunk into sixteen 32-bit big-endian words
        w = list(struct.unpack('>16I', chunk))

        # Extend the sixteen 32-bit words into eighty 32-bit words
        for i in range(16, 80):
            w.append(self._left_rotate(w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16], 1))

        # Initialize working variables
        a = self.h0
        b = self.h1
        c = self.h2
        d = self.h3
        e = self.h4

        # Main loop
        for t in range(80):
            temp = (self._left_rotate(a, 5) + self._f(t, b, c, d) +
                    e + self._k(t) + w[t]) & 0xffffffff
            e = d
            d = c
            c = self._left_rotate(b, 30)
            b = a
            a = temp

        # Add this chunk's hash to result so far
        self.h0 = (self.h0 + a) & 0xffffffff
        self.h1 = (self.h1 + b) & 0xffffffff
        self.h2 = (self.h2 + c) & 0xffffffff
        self.h3 = (self.h3 + d) & 0xffffffff
        self.h4 = (self.h4 + e) & 0xffffffff

    def update(self, message):
        """Update hash with new data"""
        if isinstance(message, str):
            message = message.encode('utf-8')

        # Pad the message
        padded_msg = self._pad_message(message)

        # Process each 512-bit chunk
        for i in range(0, len(padded_msg), 64):
            self._process_chunk(padded_msg[i:i+64])

    def digest(self):
        """Return the digest as bytes"""
        return struct.pack('>5I', self.h0, self.h1, self.h2, self.h3, self.h4)

    def hexdigest(self):
        """Return the digest as a hex string"""
        return self.digest().hex()


def sha1(message):
    """Convenience function to compute SHA-1 hash"""
    hasher = SHA1()
    hasher.update(message)
    return hasher.hexdigest()


def interactive_mode():
    """Interactive mode to hash custom messages"""
    print("\n" + "=" * 70)
    print("                    SHA-1 Interactive Hash Tool")
    print("=" * 70)
    print("\nEnter messages to hash (type 'quit' or 'exit' to stop)")
    print("Type 'test' to run test suite\n")

    while True:
        try:
            user_input = input("Enter message: ").strip()

            if user_input.lower() in ['quit', 'exit', 'q']:
                print("\nGoodbye!")
                break

            if user_input.lower() == 'test':
                run_tests()
                continue

            if not user_input:
                print("⚠ Empty string detected")

            # Compute hash
            hash_result = sha1(user_input)

            print(f"\n  Message: '{user_input}'")
            print(f"  Length:  {len(user_input)} bytes ({len(user_input) * 8} bits)")
            print(f"  SHA-1:   {hash_result}")
            print()

        except KeyboardInterrupt:
            print("\n\nInterrupted. Goodbye!")
            break
        except Exception as e:
            print(f"Error: {e}")


def run_tests():
    """Run test suite"""
    print("\n" + "=" * 70)
    print("                         SHA-1 Test Suite")
    print("=" * 70)

    test_vectors = [
        ("abc", "a9993e364706816aba3e25717850c26c9cd0d89d"),
        ("", "da39a3ee5e6b4b0d3255bfef95601890afd80709"),
        ("a", "86f7e437faa5a7fce15d1ddcb9eaeaea377667b8"),
        ("message digest", "c12252ceda8be8994d5fa0290a47231c1d16aae3"),
        ("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq",
        "84983e441c3bd26ebaae4aa1f95129e5e54670f1"),
        ("The quick brown fox jumps over the lazy dog",
        "2fd4e1c67a2d28fced849ee1bb76e7391b93eb12"),
        ("The quick brown fox jumps over the lazy cog",
        "de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3"),
        ("Hello World", "0a4d55a8d778e5022fab701977c5d840bbc486d0"),
        ("password", "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8"),
    ]

    all_passed = True
    passed_count = 0

    for message, expected in test_vectors:
        result = sha1(message)
        status = " PASS" if result == expected else " FAIL"

        if result == expected:
            passed_count += 1
        else:
            all_passed = False

        display_msg = repr(message) if len(message) <= 40 else repr(message[:40]) + "..."
        print(f"\nMessage:  {display_msg}")
        print(f"Got:      {result}")
        print(f"Expected: {expected}")
        print(f"Status:   {status}")

    print("\n" + "=" * 70)
    print(f"Results: {passed_count}/{len(test_vectors)} tests passed")
```

```
    if all_passed:
        print("All tests PASSED! ")
    else:
        print("Some tests FAILED! ") print()


def hash_from_args():
    """Hash messages from command line arguments"""
    if len(sys.argv) < 2:
        return False

    # Check for special flags
    if sys.argv[1] in ['-h', '--help']:
        print_usage()
        return True

    if sys.argv[1] in ['-t', '--test']:
        run_tests()
        return True

    # Hash all arguments
    for message in sys.argv[1:]:
        hash_result = sha1(message)
        print(f"Message: '{message}'")
        print(f"SHA-1:   {hash_result}\n")

    return True



def print_usage():
    """Print usage information"""
    print("\nSHA-1 Hash Tool - Usage:")
    print("=" * 70)
    print("\n  Interactive mode:")
    print("    python sha1.py")
    print("\n  Hash command-line arguments:")
    print("    python sha1.py 'message1' 'message2' ...")
```

```
        print("\n  Run test suite:")
        print("     python sha1.py --test")
        print("     python sha1.py -t")
        print("\n  Show help:")
        print("     python sha1.py --help")
        print("     python sha1.py -h")
        print("\n" + "=" * 70)


    if __name___== "__main__":
        # If command line arguments provided, use them
        if not hash_from_args():
            # Otherwise, enter interactive mode
interactive_mode()
```

Combined Version of the SHA1 and SHA256:

```
//==========================================
// Unified SHA Core Design Module (SHA-1 & SHA-256)
//==========================================
module parametrised_sha_design #(
    parameter BLOCK_WIDTH = 512,
    parameter HASH_WIDTH = 160,        // 160 for SHA-1, 256 for SHA-256
    parameter WORD_WIDTH = 32,
    parameter NUM_ROUNDS = 80,         // 80 for SHA-1, 64 for SHA-256
    parameter ROUND_WIDTH = 7,
    parameter STATE_WIDTH = 3,
    parameter ALGORITHM = "SHA1"      // "SHA1" or "SHA256"
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [BLOCK_WIDTH-1:0] block_in,
    output reg [HASH_WIDTH-1:0] hash_out,
    output reg ready
);
    localparam NUM_WORDS = BLOCK_WIDTH / WORD_WIDTH;
localparam NUM_HASH_WORDS = HASH_WIDTH / WORD_WIDTH;

// SHA-1 Constants
localparam [WORD_WIDTH-1:0] K1_0 = 32'h5a827999;
localparam [WORD_WIDTH-1:0] K1_1 = 32'h6ed9eba1;
localparam [WORD_WIDTH-1:0] K1_2 = 32'h8f1bbcdc;
localparam [WORD_WIDTH-1:0] K1_3 = 32'hca62c1d6;

// SHA-256 K constants - using function instead of array for Verilog compatibility
function [WORD_WIDTH-1:0] get_K256;
   input [5:0] index;
   begin
      case (index)
         6'd0:  get_K256 = 32'h428a2f98;
         6'd1:  get_K256 = 32'h71374491;
         6'd2:  get_K256 = 32'hb5c0fbcf;
         6'd3:  get_K256 = 32'he9b5dba5;
         6'd4:  get_K256 = 32'h3956c25b;
         6'd5:  get_K256 = 32'h59f111f1;
         6'd6:  get_K256 = 32'h923f82a4;
         6'd7:  get_K256 = 32'hab1c5ed5;
         6'd8:  get_K256 = 32'hd807aa98;
         6'd9:  get_K256 = 32'h12835b01;
         6'd10: get_K256 = 32'h243185be;
         6'd11: get_K256 = 32'h550c7dc3;
         6'd12: get_K256 = 32'h72be5d74;
         6'd13: get_K256 = 32'h80deb1fe;
         6'd14: get_K256 = 32'h9bdc06a7;
         6'd15: get_K256 = 32'hc19bf174;
         6'd16: get_K256 = 32'he49b69c1;
         6'd17: get_K256 = 32'hefbe4786;
         6'd18: get_K256 = 32'h0fc19dc6;
         6'd19: get_K256 = 32'h240ca1cc;
         6'd20: get_K256 = 32'h2de92c6f;
```

```
6'd21: get_K256 = 32'h4a7484aa;
6'd22: get_K256 = 32'h5cb0a9dc;
6'd23: get_K256 = 32'h76f988da;
6'd24: get_K256 = 32'h983e5152;
6'd25: get_K256 = 32'ha831c66d;
6'd26: get_K256 = 32'hb00327c8;
6'd27: get_K256 = 32'hbf597fc7;
6'd28: get_K256 = 32'hc6e00bf3;
6'd29: get_K256 = 32'hd5a79147;
6'd30: get_K256 = 32'h06ca6351;
6'd31: get_K256 = 32'h14292967;
6'd32: get_K256 = 32'h27b70a85;
6'd33: get_K256 = 32'h2e1b2138;
6'd34: get_K256 = 32'h4d2c6dfc;
6'd35: get_K256 = 32'h53380d13;
6'd36: get_K256 = 32'h650a7354;
6'd37: get_K256 = 32'h766a0abb;
6'd38: get_K256 = 32'h81c2c92e;
6'd39: get_K256 = 32'h92722c85;
6'd40: get_K256 = 32'ha2bfe8a1;
6'd41: get_K256 = 32'ha81a664b;
6'd42: get_K256 = 32'hc24b8b70;
6'd43: get_K256 = 32'hc76c51a3;
6'd44: get_K256 = 32'hd192e819;
6'd45: get_K256 = 32'hd6990624;
6'd46: get_K256 = 32'hf40e3585;
6'd47: get_K256 = 32'h106aa070;
6'd48: get_K256 = 32'h19a4c116;
6'd49: get_K256 = 32'h1e376c08;
6'd50: get_K256 = 32'h2748774c;
6'd51: get_K256 = 32'h34b0bcb5;
6'd52: get_K256 = 32'h391c0cb3;
6'd53: get_K256 = 32'h4ed8aa4a;
6'd54: get_K256 = 32'h5b9cca4f;
```

```
            6'd55: get_K256 = 32'h682e6ff3;
            6'd56: get_K256 = 32'h748f82ee;
            6'd57: get_K256 = 32'h78a5636f;
            6'd58: get_K256 = 32'h84c87814;
            6'd59: get_K256 = 32'h8cc70208;
            6'd60: get_K256 = 32'h90befffa;
            6'd61: get_K256 = 32'ha4506ceb;
            6'd62: get_K256 = 32'hbef9a3f7;
            6'd63: get_K256 = 32'hc67178f2;
            default: get_K256 = 32'h00000000;
        endcase
    end
endfunction

// SHA-1 Initial Hash Values
localparam [WORD_WIDTH-1:0] H1_0_INIT = 32'h67452301;
localparam [WORD_WIDTH-1:0] H1_1_INIT = 32'hefcdab89;
localparam [WORD_WIDTH-1:0] H1_2_INIT = 32'h98badcfe;
localparam [WORD_WIDTH-1:0] H1_3_INIT = 32'h10325476;
localparam [WORD_WIDTH-1:0] H1_4_INIT = 32'hc3d2e1f0;

// SHA-256 Initial Hash Values
localparam [WORD_WIDTH-1:0] H256_0_INIT = 32'h6a09e667;
localparam [WORD_WIDTH-1:0] H256_1_INIT = 32'hbb67ae85;
localparam [WORD_WIDTH-1:0] H256_2_INIT = 32'h3c6ef372;
localparam [WORD_WIDTH-1:0] H256_3_INIT = 32'ha54ff53a;
localparam [WORD_WIDTH-1:0] H256_4_INIT = 32'h510e527f;
localparam [WORD_WIDTH-1:0] H256_5_INIT = 32'h9b05688c;
localparam [WORD_WIDTH-1:0] H256_6_INIT = 32'h1f83d9ab;
localparam [WORD_WIDTH-1:0] H256_7_INIT = 32'h5be0cd19;

localparam [STATE_WIDTH-1:0] IDLE = 3'd0;
localparam [STATE_WIDTH-1:0] EXPAND = 3'd1;
localparam [STATE_WIDTH-1:0] COMPUTE = 3'd2;
localparam [STATE_WIDTH-1:0] FINALIZE = 3'd3;

reg [STATE_WIDTH-1:0] state;
reg [ROUND_WIDTH-1:0] round;

// Working variables (SHA-256 uses A-H, SHA-1 uses A-E)
reg [WORD_WIDTH-1:0] A, B, C, D, E, F, G, H;
reg [WORD_WIDTH-1:0] H0, H1, H2, H3, H4, H5, H6, H7;
reg [WORD_WIDTH-1:0] W [0:NUM_ROUNDS-1];

reg [WORD_WIDTH-1:0] temp1, temp2;

integer i;

// SHA-1 ROTL function
function [WORD_WIDTH-1:0] ROTL;
   input [WORD_WIDTH-1:0] x;
   input [4:0] n;
   begin
      ROTL = (x << n) | (x >> (WORD_WIDTH - n));
   end
endfunction

// SHA-256 ROTR function
function [WORD_WIDTH-1:0] ROTR;
   input [WORD_WIDTH-1:0] x;
   input [4:0] n;
   begin
      ROTR = (x >> n) | (x << (WORD_WIDTH - n));
   end
endfunction

// SHA-256 functions
function [WORD_WIDTH-1:0] Ch;
```

```
    input [WORD_WIDTH-1:0] x, y, z;
    begin
        Ch = (x & y) ^ ((~x) & z);
    end
endfunction

function [WORD_WIDTH-1:0] Maj;
    input [WORD_WIDTH-1:0] x, y, z;
    begin
        Maj = (x & y) ^ (x & z) ^ (y & z);
    end
endfunction

function [WORD_WIDTH-1:0] Sigma0;
    input [WORD_WIDTH-1:0] x;
    begin
        Sigma0 = ROTR(x, 5'd2) ^ ROTR(x, 5'd13) ^ ROTR(x, 5'd22);
    end
endfunction

function [WORD_WIDTH-1:0] Sigma1;
    input [WORD_WIDTH-1:0] x;
    begin
        Sigma1 = ROTR(x, 5'd6) ^ ROTR(x, 5'd11) ^ ROTR(x, 5'd25);
    end
endfunction

function [WORD_WIDTH-1:0] sigma0;
    input [WORD_WIDTH-1:0] x;
    begin
        sigma0 = ROTR(x, 5'd7) ^ ROTR(x, 5'd18) ^ (x >> 3);
    end
endfunction
```

```
function [WORD_WIDTH-1:0] sigma1;
   input [WORD_WIDTH-1:0] x;
   begin
      sigma1 = ROTR(x, 5'd17) ^ ROTR(x, 5'd19) ^ (x >> 10);
   end
endfunction

// SHA-1 f function
function [WORD_WIDTH-1:0] f_func;
   input [ROUND_WIDTH-1:0] t;
   input [WORD_WIDTH-1:0] b, c, d;
   begin
      if (t < 20)
         f_func = (b & c) | ((~b) & d);
      else if (t < 40)
         f_func = b ^ c ^ d;
      else if (t < 60)
         f_func = (b & c) | (b & d) | (c & d);
      else
         f_func = b ^ c ^ d;
   end
endfunction

// SHA-1 k constant
function [WORD_WIDTH-1:0] k_const;
   input [ROUND_WIDTH-1:0] t;
   begin
      if (t < 20)
         k_const = K1_0;
      else if (t < 40)
         k_const = K1_1;
      else if (t < 60)
         k_const = K1_2;
      else
```

```
        k_const = K1_3;
    end
endfunction

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        ready  <=  1'b0;
        round <= 0;
        A <= 0; B <= 0; C <= 0; D <= 0;
        E <= 0; F <= 0; G <= 0; H <= 0;
        H0 <= 0; H1 <= 0; H2 <= 0; H3 <= 0;
        H4 <= 0; H5 <= 0; H6 <= 0; H7 <= 0;
        hash_out <= 0;
        for (i = 0; i < NUM_ROUNDS; i = i + 1)
            W[i] <= 0;
    end else begin
        case (state)
            IDLE: begin
                ready <= 1'b0;
                if (start) begin
                    // Initialize hash values based on algorithm
                    if (ALGORITHM == "SHA1") begin
                        H0 <= H1_0_INIT;
                        H1 <= H1_1_INIT;
                        H2 <= H1_2_INIT;
                        H3 <= H1_3_INIT;
                        H4 <= H1_4_INIT;
                        A <= H1_0_INIT;
                        B <= H1_1_INIT;
                        C <= H1_2_INIT;
                        D <= H1_3_INIT;
                        E <= H1_4_INIT;
                    end else begin // SHA256
                        H0 <= H256_0_INIT;
                        H1 <= H256_1_INIT;
                        H2 <= H256_2_INIT;
                        H3 <= H256_3_INIT;
                        H4 <= H256_4_INIT;
                        H5 <= H256_5_INIT;
                        H6 <= H256_6_INIT;
                        H7 <= H256_7_INIT;
                        A <= H256_0_INIT;
                        B <= H256_1_INIT;
                        C <= H256_2_INIT;
                        D <= H256_3_INIT;
                        E <= H256_4_INIT;
                        F <= H256_5_INIT;
                        G <= H256_6_INIT;
                        H <= H256_7_INIT;
                    end

                    // Load 16 words
                    for (i = 0; i < NUM_WORDS; i = i + 1) begin
                        W[i] <= block_in[BLOCK_WIDTH-1 - i*WORD_WIDTH -:
        WORD_WIDTH];
                    end

                    round <= NUM_WORDS;
                    state <= EXPAND;
                end
              end

              EXPAND: begin
                if (round < NUM_ROUNDS) begin
                    if (ALGORITHM == "SHA1") begin
                        // SHA-1 message schedule
                        W[round] <= ROTL(W[round-3] ^ W[round-8] ^ W[round-14] ^ W[round-
    16], 5'd1);
                    end else begin
                        // SHA-256 message schedule
                        W[round] <= sigma1(W[round-2]) + W[round-7] + sigma0(W[round-15]) +
    W[round-16];
                    end
                    round <= round + 1;
                end else begin
                    round <= 0;
                    state <= COMPUTE;
                end
              end

              COMPUTE: begin
                if (ALGORITHM == "SHA1") begin
                    // SHA-1 compression
                    temp1 = ROTL(A, 5'd5) + f_func(round, B, C, D) + E + k_const(round) +
        W[round];
                    E <= D;
                    D <= C;
                    C <= ROTL(B, 5'd30);
                    B <= A;
                    A <= temp1;
                end else begin
                    // SHA-256 compression
                    temp1 = H + Sigma1(E) + Ch(E, F, G) + get_K256(round[5:0]) + W[round];
                    temp2 = Sigma0(A) + Maj(A, B, C);
                    H <= G;
                    G <= F;
                    F <= E;
                    E <= D + temp1;
                    D <= C;
                    C <= B;
                    B <= A;
                    A <= temp1 + temp2;
                end

                if (round == NUM_ROUNDS - 1) begin
                    state <= FINALIZE;
                end else begin
                    round <= round + 1;
                end
              end

              FINALIZE: begin
                if (ALGORITHM == "SHA1") begin
                    H0 <= H0 + A;
                    H1 <= H1 + B;
                    H2 <= H2 + C;
                    H3 <= H3 + D;
                    H4 <= H4 + E;
                    hash_out <= {H0 + A, H1 + B, H2 + C, H3 + D, H4 + E};
                end else begin
                    H0 <= H0 + A;
                    H1 <= H1 + B;
                    H2 <= H2 + C;
                    H3 <= H3 + D;
                    H4 <= H4 + E;
                    H5 <= H5 + F;
                    H6 <= H6 + G;
                    H7 <= H7 + H;
                    hash_out <= {H0 + A, H1 + B, H2 + C, H3 + D, H4 + E, H5 + F, H6 + G, H7
        + H};
                end

                ready <= 1'b1;
                state <= IDLE;
              end
```

```
            default: state <= IDLE;
         endcase
      end
   end
endmodule

//=============================================
// Unified SHA Testbench with DEBUG
//=============================================
module tb_sha_debug;
   parameter CLK_PERIOD = 10;
   parameter HASH_WIDTH = 160;      // 160 for SHA-1, 256 for SHA-256
   parameter DATA_WIDTH = 1024;
   parameter NUM_ROUNDS = 80;       // 80 for SHA-1, 64 for SHA-256

   reg clk, rst_n;
   reg start;
   wire [HASH_WIDTH-1:0] hash_out;
   wire done;

   integer test_count = 0;
   integer pass_count = 0;
   integer fail_count = 0;

   // Test configuration
   localparam TEST_STRING = "Baranika";
   localparam TEST_LEN = 8;
   localparam ALGORITHM = "SHA1";  // "SHA1" or "SHA256"

   function [DATA_WIDTH-1:0] string_to_hex;
      input [1024*8-1:0] str;
      input integer len;
      integer i;
```

```
        reg [DATA_WIDTH-1:0] result;
        begin
            result = {DATA_WIDTH{1'b0}};
            for (i = 0; i < len; i = i + 1) begin
                result[DATA_WIDTH-1 - i*8 -: 8] = str[(len-1-i)*8 +: 8];
            end
            string_to_hex = result;
        end
    endfunction

    localparam [DATA_WIDTH-1:0] TEST_DATA = string_to_hex(TEST_STRING,
TEST_LEN);

    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
    end

    parametrised_sha_top #(
        .MSG_LEN(TEST_LEN),
        .MESSAGE_DATA(TEST_DATA),
        .HASH_WIDTH(HASH_WIDTH),
        .NUM_ROUNDS(NUM_ROUNDS),
        .ALGORITHM(ALGORITHM)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .start(start),
        .hash_out(hash_out),
        .done(done)
    );

    initial begin
        $monitor("Time=%0t | start=%b | state=%0d | pad_valid=%b | sha_ready=%b |
done=%b",
                 $time, start, dut.state, dut.pad_valid, dut.sha_ready, done);
    end

    initial begin
        wait(dut.pad_valid);
        @(posedge clk);
        $display("\n=== PADDING BLOCK DEBUG ===");
        $display("Algorithm: %s", ALGORITHM);
        $display("Test string: '%s'", TEST_STRING);
        $display("Padded block (hex): %h", dut.padded_block);
        $display("==============================\n");
    end

    initial begin
        $dumpfile("sha_debug.vcd");
        $dumpvars(0, tb_sha_debug);


$display("\n================================================================
==========================");
        $display("%s Test: '%s' (%0d bytes)", ALGORITHM, TEST_STRING, TEST_LEN);

$display("=================================================================
=========================\n");

        rst_n = 1'b0;
        start = 1'b0;

        repeat(5) @(posedge clk);
        rst_n = 1'b1;
        repeat(5) @(posedge clk);

        $display("Starting test...\n");
        start = 1'b1;
        @(posedge clk);
        start = 1'b0;

        repeat(500) begin
           @(posedge clk);
           if (done) begin
              test_count = test_count + 1;
              $display("\n✓ Test completed!");
              $display("Input String: '%s'", TEST_STRING);
              $display("%s Hash:  %h", ALGORITHM, hash_out);

              pass_count = pass_count + 1;


$display("\n============================================================
======================");
              $display("Test Summary: %0d total, %0d passed, %0d failed", test_count,
pass_count, fail_count);

$display("============================================================
=====================\n");
              $finish;
           end
        end

        $display("\n✗ TIMEOUT - Test did not complete");
        fail_count = fail_count + 1;
        $finish;
     end

     initial begin
        #50000;
        $display("\n Global Timeout");
        $finish;
    end
endmodule

//==========================================
// Unified SHA Padding Module
//==========================================
module parametrised_sha_padding #(
    parameter DATA_WIDTH = 1024,
    parameter BLOCK_WIDTH = 512,
    parameter MSG_LEN_WIDTH = 80,
    parameter BLOCK_NUM_WIDTH = 20,
    parameter LEN_FIELD_BITS = 64
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [DATA_WIDTH-1:0] data_in,
    input wire [MSG_LEN_WIDTH-1:0] msg_len,
    output reg [BLOCK_WIDTH-1:0] block_out,
    output reg valid,
    output reg done,
    output reg [BLOCK_NUM_WIDTH-1:0] block_num,
    output reg [BLOCK_NUM_WIDTH-1:0] total_blocks
);
    localparam BYTES_PER_BLOCK = BLOCK_WIDTH / 8;

    reg [2:0] state;
    localparam IDLE = 3'd0;
    localparam CALC = 3'd1;
    localparam OUTPUT = 3'd2;
    localparam DONE_STATE = 3'd3;

    reg [MSG_LEN_WIDTH-1:0] byte_len;
    reg [LEN_FIELD_BITS-1:0] bit_len;
    integer i;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            block_out <= {BLOCK_WIDTH{1'b0}};
            valid <= 1'b0;
            done <= 1'b0;
            block_num <= {BLOCK_NUM_WIDTH{1'b0}};
            total_blocks <= {BLOCK_NUM_WIDTH{1'b0}};
            byte_len <= {MSG_LEN_WIDTH{1'b0}};
            bit_len <= {LEN_FIELD_BITS{1'b0}};
        end else begin
            case (state)
                IDLE: begin
                    valid <= 1'b0;
                    done <= 1'b0;
                    block_num <= {BLOCK_NUM_WIDTH{1'b0}};
                    if (start) begin
                        byte_len <= msg_len >> 3;
                        bit_len <= msg_len[LEN_FIELD_BITS-1:0];
                        total_blocks <= 1;
                        state <= CALC;
                    end
                end

                CALC: begin
                    // Copy message and add padding (same for both SHA-1 and SHA-256)
                    for (i = 0; i < 56; i = i + 1) begin
                        if (i < byte_len)
                            block_out[BLOCK_WIDTH-1 - i*8 -: 8] <= data_in[DATA_WIDTH-1 -
i*8 -: 8];
                        else if (i == byte_len)
                            block_out[BLOCK_WIDTH-1 - i*8 -: 8] <= 8'h80;
                        else
                            block_out[BLOCK_WIDTH-1 - i*8 -: 8] <= 8'h00;
                    end

                    // Add 64-bit length at end (big-endian)
                    block_out[63:56] <= bit_len[63:56];
                    block_out[55:48] <= bit_len[55:48];
                    block_out[47:40] <= bit_len[47:40];
                    block_out[39:32] <= bit_len[39:32];
                    block_out[31:24] <= bit_len[31:24];
                    block_out[23:16] <= bit_len[23:16];
                    block_out[15:8]  <= bit_len[15:8];
                    block_out[7:0]   <= bit_len[7:0];

                    block_num <= {BLOCK_NUM_WIDTH{1'b0}};
                    state <= OUTPUT;
                end

                OUTPUT: begin
                    valid <= 1'b1;
                    state <= DONE_STATE;
                end

                DONE_STATE: begin
                    valid <= 1'b0;
                    done <= 1'b1;
                    if (!start) begin
                        state <= IDLE;
                    end
                end

                default: state <= IDLE;
            endcase
```

```
        end
    end
endmodule

//==========================================
// Unified SHA Top Module
//==========================================
module parametrised_sha_top #(
    parameter DATA_WIDTH = 1024,
    parameter BLOCK_WIDTH = 512,
    parameter HASH_WIDTH = 160,       // 160 for SHA-1, 256 for SHA-256
    parameter MSG_LEN_WIDTH = 80,
    parameter BLOCK_NUM_WIDTH = 20,
    parameter WORD_WIDTH = 32,
    parameter NUM_ROUNDS = 80,        // 80 for SHA-1, 64 for SHA-256
    parameter ROUND_WIDTH = 7,
    parameter STATE_WIDTH = 3,
    parameter MSG_LEN = 3,
    parameter [DATA_WIDTH-1:0] MESSAGE_DATA = 1024'h616263,
    parameter ALGORITHM = "SHA1"     // "SHA1" or "SHA256"
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    output reg [HASH_WIDTH-1:0] hash_out,
    output reg done
);

    wire [DATA_WIDTH-1:0] data_in = MESSAGE_DATA;
    wire [MSG_LEN_WIDTH-1:0] msg_len = MSG_LEN * 8;

    wire [BLOCK_WIDTH-1:0] padded_block;
    wire pad_valid;
    wire pad_done;
wire [BLOCK_NUM_WIDTH-1:0] block_num;
wire [BLOCK_NUM_WIDTH-1:0] total_blocks;

reg sha_start;
wire [HASH_WIDTH-1:0] sha_hash;
wire sha_ready;

reg [HASH_WIDTH-1:0] hash_reg;
reg [BLOCK_WIDTH-1:0] captured_block;

reg [2:0] state;
localparam IDLE = 3'd0;
localparam WAIT_VALID = 3'd1;
localparam START_SHA = 3'd2;
localparam WAIT_SHA = 3'd3;
localparam DONE_STATE = 3'd4;

parametrised_sha_padding #(
    .DATA_WIDTH(DATA_WIDTH),
    .BLOCK_WIDTH(BLOCK_WIDTH),
    .MSG_LEN_WIDTH(MSG_LEN_WIDTH),
    .BLOCK_NUM_WIDTH(BLOCK_NUM_WIDTH),
    .LEN_FIELD_BITS(64)
) u_padding (
    .clk(clk),
    .rst_n(rst_n),
    .start(start),
    .data_in(data_in),
    .msg_len(msg_len),
    .block_out(padded_block),
    .valid(pad_valid),
    .done(pad_done),
    .block_num(block_num),
    .total_blocks(total_blocks)
);

parametrised_sha_design #(
    .BLOCK_WIDTH(BLOCK_WIDTH),
    .HASH_WIDTH(HASH_WIDTH),
    .WORD_WIDTH(WORD_WIDTH),
    .NUM_ROUNDS(NUM_ROUNDS),
    .ROUND_WIDTH(ROUND_WIDTH),
    .STATE_WIDTH(STATE_WIDTH),
    .ALGORITHM(ALGORITHM)
) u_sha_core (
    .clk(clk),
    .rst_n(rst_n),
    .start(sha_start),
    .block_in(captured_block),
    .hash_out(sha_hash),
    .ready(sha_ready)
);

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        sha_start <= 1'b0;
        done <= 1'b0;
        hash_out <= {HASH_WIDTH{1'b0}};
        hash_reg <= {HASH_WIDTH{1'b0}};
        captured_block <= {BLOCK_WIDTH{1'b0}};
    end else begin
        case (state)
            IDLE: begin
                sha_start <= 1'b0;
                done <= 1'b0;
                if (start) begin
                    state <= WAIT_VALID;
            end
        end

        WAIT_VALID: begin
            if (pad_valid) begin
                captured_block <= padded_block;
                state <= START_SHA;
            end
        end

        START_SHA: begin
            sha_start <= 1'b1;
            state <= WAIT_SHA;
        end

        WAIT_SHA: begin
            sha_start <= 1'b0;
            if (sha_ready) begin
                hash_reg <= sha_hash;
                hash_out <= sha_hash;
                state <= DONE_STATE;
            end
        end

        DONE_STATE: begin
            done <= 1'b1;
            if (!start) begin
                state <= IDLE;
            end
        end

        default: state <= IDLE;
    endcase
end
```

```
     end

endmodule

//
===========================================================================
===============
// HOW TO USE THIS UNIFIED SHA MODULE
//
===========================================================================
===============
//
// To switch between SHA-1 and SHA-256, change these parameters:
//
// FOR SHA-1:
// parameter HASH_WIDTH = 160
// parameter NUM_ROUNDS = 80
// parameter ALGORITHM = "SHA1"
//
// FOR SHA-256:
// parameter HASH_WIDTH = 256
// parameter NUM_ROUNDS = 64
// parameter ALGORITHM = "SHA256"
//
// Example test for "abc":
// SHA-1:   a9993e364706816aba3e25717850c26c9cd0d89d
// SHA-256: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
//
//
===========================================================================
=====
```