



UNIVERSITY OF PISA  
MASTER'S DEGREE IN CYBERSECURITY

## FULL HASH ALGORITHM V1 (AES S-BOX)

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)  
**Nicolò Mariano Fragale**  
**Federico Rossetti**

Academic year 2024/2025

---

# Contents

---

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Project Specifications</b>                            | <b>1</b>  |
| 1.1      | Hardware Design Specifications . . . . .                 | 2         |
| 1.2      | Implementation Plan . . . . .                            | 2         |
| 1.3      | SystemVerilog Architecture Overview . . . . .            | 3         |
| <b>2</b> | <b>High-level Model</b>                                  | <b>4</b>  |
| 2.1      | Message Processing Phase . . . . .                       | 4         |
| 2.2      | Finalization Phase . . . . .                             | 5         |
| 2.3      | Handling Empty Messages . . . . .                        | 5         |
| <b>3</b> | <b>RTL Design</b>  | <b>6</b>  |
| <b>4</b> | <b>Interface Specifications and Expected Behavior</b>    | <b>7</b>  |
| 4.1      | Module <code>control_fsm</code> . . . . .                | 7         |
| 4.1.1    | Functionality . . . . .                                  | 8         |
| 4.2      | Module <code>hash_top</code> . . . . .                   | 12        |
| 4.2.1    | Functionality . . . . .                                  | 12        |
| 4.2.2    | <code>C[i]</code> Construction (Counter Bytes) . . . . . | 13        |
| 4.3      | Module <code>hash_registers</code> . . . . .             | 13        |
| 4.3.1    | Functionality . . . . .                                  | 14        |
| 4.4      | Module <code>shift_xor_unit</code> . . . . .             | 14        |
| 4.4.1    | Functionality . . . . .                                  | 15        |
| 4.5      | Module <code>final_hash_unit</code> . . . . .            | 15        |
| 4.5.1    | Functionality . . . . .                                  | 16        |
| 4.6      | Module <code>round_tracker</code> . . . . .              | 16        |
| 4.6.1    | Functionality . . . . .                                  | 17        |
| 4.7      | Module <code>aes_sbox_lut</code> . . . . .               | 17        |
| 4.7.1    | Functionality . . . . .                                  | 18        |
| <b>5</b> | <b>Functional Verification</b>                           | <b>19</b> |
| 5.1      | Verification of Individual Modules . . . . .             | 20        |

|          |                                    |           |
|----------|------------------------------------|-----------|
| 5.1.1    | hash_registers . . . . .           | 20        |
| 5.1.2    | round_tracker . . . . .            | 20        |
| 5.1.3    | control_fsm . . . . .              | 20        |
| 5.1.4    | hash_top . . . . .                 | 21        |
| 5.1.5    | Waveform Analysis . . . . .        | 22        |
| <b>6</b> | <b>FPGA Implementation Results</b> | <b>25</b> |
| 6.1      | Analysis and Synthesis . . . . .   | 25        |
| 6.2      | Fitter . . . . .                   | 27        |
| 6.3      | Timinig Analysis . . . . .         | 29        |
| 6.4      | Conclusion . . . . .               | 31        |

---

# CHAPTER 1

---

## Project Specifications

---

*An introduction to the project, including the specifications and their analysis.*

The goal of this project is to transform a message (seen as a sequence of bytes) into a unique digest, using two main phases. The first phase, called the **message processing phase**, iterates over each input byte and applies a transformation using the AES S-box. The second phase, the **finalization phase**, is where the internal state is updated one last time using the length of the input message as a counter.

The internal state of the hash is stored in an 8-byte vector  $H[0..7]$ , initialized with the following values:

|        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|
| $H[0]$ | $H[1]$ | $H[2]$ | $H[3]$ | $H[4]$ | $H[5]$ | $H[6]$ | $H[7]$ |
| 0x11   | 0xA3   | 0x1F   | 0x3A   | 0xCC   | 0x84   | 0xCC   | 0xA0   |

For each message byte  $M$ , the following transformation is executed over 36 rounds:

for  $r = 0$  to 35 :    for  $i = 0$  to 7 :     $H[i] = S((H[(i + 1) \bmod 8] \oplus M) \lll i)$

Where:

- $\oplus$  denotes bitwise XOR,
- $\lll i$  is a left circular shift by  $i$  bits,
- $S(\cdot)$  is the AES S-box transformation.

After all the message bytes have been processed, a final update is performed using an 8-byte counter  $C$  (which is simply the message length in bits, encoded in big endian

format):

$$\text{for } i = 0 \text{ to } 7 : \quad H[i] = S((H[(i+1) \bmod 8] \oplus C[i]) \lll i)$$

This step ensures that the final digest depends not only on the content of the message, but also on its exact length. If the input message is empty, the system still runs this final step, applying it directly to the initial  $H[i]$  values with  $C = 0$ .

## 1.1 Hardware Design Specifications

---

The design had to meet the following requirements:

- Support an **asynchronous, active-low reset**.
- Include the flag **valid\_in** that signals when a new byte is valid.
- Count the number of message bytes internally, for use in the finalization step.
- Provide the **digest\_ready** signal to indicate when the output is valid and stable.

These constraints required careful state management and proper coordination between all the internal modules, especially when it came to finalizing the hash computation only when all input data had been correctly processed.

## 1.2 Implementation Plan

---

The development of the project was structured in four key phases:

1. **High-Level Modeling** – A Python prototype (HL\_Model\_Full\_Hash\_S\_BOX.py) simulates the hash function and produce reference digests. This model played a fundamental role in debugging and validating the RTL design.
2. **RTL Design** – The SystemVerilog implementation was developed with modularity in mind, reflecting the conceptual flow of the hash algorithm: message processing, finalization, and output generation.
3. **Simulation and Verification** – Testbenches simulates different message scenarios, including normal messages, edge cases, and the empty string case. Functional correctness was verified against the Python reference.
4. **FPGA Implementation** – The project was synthesized and tested on the Cyclone V FPGA (5CGXFC9D6F27C7) using Intel Quartus. Static timing analysis (STA) was performed to evaluate performance.

### 1.3 SystemVerilog Architecture Overview

---

The SystemVerilog implementation was split into several modular components, each responsible for a specific part of the algorithm:

- **hash\_top.sv**: the top-level that connects all submodules and manages input/output interfaces.
- **control\_fsm.sv**: the heart of the control flow. It manages the state transitions across *IDLE*, *INIT*, *LOAD\_BYTE*, *ROUND\_EXEC*, *FINAL\_HASH*, and *FINALIZE*. It controls when to load data, start rounds, and issue the digest.
- **H[i].sv**: implements the hash register bank with 8 registers representing  $H[0]$  to  $H[7]$ , updated on each round and performs the core operation of each round such as XOR and rotation operations.
- **sbox.sv**: implement 256-entry LUT.
- **round.sv**: manage the number of cycles to perform according to the assignment.

This modular approach helped isolate responsibilities and made the simulation and debugging process much clearer.

---

# CHAPTER 2

---

## High-level Model

---

To guide and verify the RTL implementation, we developed a high-level model of the hash function: `HL_Model_Full_Hash_S_BOX.py` that acts as a golden reference to check the functional correctness of the hardware design.

The model is structured in two distinct phases: message processing and finalization.

### 2.1 Message Processing Phase

---

The internal state  $H[i]$  is initialized with a set of fixed 8-bit values that ensure diffusion and non-linearity from the very beginning. The initial state is defined as:

For each message byte  $M$ , are performed 36 rounds of transformation. Each byte in the state is updated by applying a XOR, left rotation and AES S-box substitution. The logic follows this structure:

**Listing 2.1:** *Main message processing loop*

```
1 for m in msg_bytes:
2     for r in range(36):
3         for i in range(8):
4             index = (1 + i) % 8
5             val = hash_value[index] ^ m
6             shift = ((val << i) | (val >> (8 - i))) & 0xFF
7             hash_value[i] = AES_S_BOX_lookup(shift)
```

This nested loop guarantees that each byte of the message has a global effect across the internal state  $H$ .

## 2.2 Finalization Phase

---

After processing all message bytes, we incorporate the message length which is converted to an 8-byte counter in big-endian format and used for a final round of transformations:

**Listing 2.2:** *Finalization step*

```
1 C = list(len(msg_bytes).to_bytes(8, 'big'))
2 for i in range(8):
3     index = (i + 1) % 8
4     val = hash_value[index] ^ C[i]
5     shift = ((val << i) | (val >> (8 - i))) & 0xFF
6     hash_value[i] = AES_S_BOX_lookup(shift)
```

## 2.3 Handling Empty Messages

---

A key detail is the ability to handle messages with zero length. In such a case, the main message processing loop is skipped entirely applying only the finalization logic using a zero-length counter:

**Listing 2.3:** *Handling empty messages*

```
1     if (len(sys.argv) >= 3):
2         message = sys.argv[2]
3     else:
4         message = ""
```

This behavior is essential to meet the specifications and was carefully validated against the RTL counterpart.



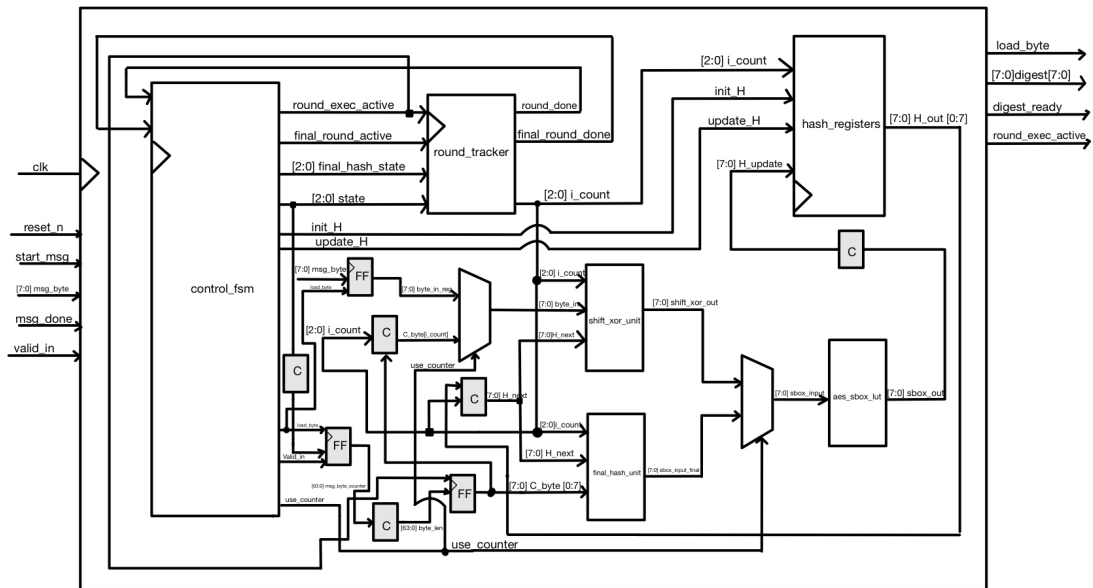
---

# CHAPTER 3

---

## RTL Design

---



**Figure 3.1:** *High-level block diagram.*

The diagram shows the hierarchical structure of the `hash_top` module and its subcomponents, including the `control_fsm`, `round_tracker`, `hash_registers`, and logic blocks such as the `shift_xor_unit` and `aes_sbox_lut`. Combinational logic is represented by plain rectangular blocks, while edge-triggered elements like registers are marked with blank triangles, adhering to standard hardware design conventions.

---

# CHAPTER 4

---

## Interface Specifications and Expected Behavior

---

### 4.1 Module `control_fsm`

---

#### Inputs

- `clk`: Main system clock. Governs all sequential transitions in the FSM.
- `reset_n`: Active-low asynchronous reset. Forces the FSM into the `IDLE` state and resets internal latches.
- `start_msg`: Triggers the beginning of a new hashing session when set high. Causes a transition from `IDLE` to `INIT`.
- `msg_done`: Pulsed high when the external controller has finished sending the message bytes. It is latched internally and used to switch from message processing to finalization.
- `round_done`: Raised by the `round_tracker` to indicate that 8 rounds (i.e., updates of all  $H[i]$ ) have been completed. Used to transition out of the `ROUND_EXEC` or `FINAL_HASH` states.
- `final_round_done`: High when the last update during finalization has completed (i.e., all 8  $C[i]$  bytes have been processed).
- `valid_in`: Indicates that the current message byte at input is valid and can be latched.

#### Outputs

- `init_H`: Signals the `hash_registers` module to initialize the  $H[i]$  registers to their IVs.

- `load_byte`: Enables latching of `msg_byte` into the internal input register in `hash_top`.
- `use_counter`: Selects counter-based byte input ( $C[i]$ ) instead of message byte.
- `round_exec_active`: High during `ROUND_EXEC` to indicate an ongoing round.
- `digest_ready`: Asserted in the `FINALIZE` state to indicate that the 64-bit digest is ready to be read.
- `final_round_active`: Signals that the system is executing the final transformation with  $C[i]$ .
- `update_H`: Tells the `hash_registers` module to latch the new transformed value  $H[i]$ .
- `state`: Exposes the current FSM state as a 3-bit code. Useful for debugging or control monitoring.
- `final_hash_state`: Constant output that holds the encoded ID of the `FINAL_HASH` state. Used by other modules to detect phase transitions.

#### 4.1.1 Functionality

This module defines a 6-state FSM responsible for coordinating the sequence of operations in the hashing process, including initialization, message absorption, iterative rounds, and final digest production. The FSM uses a latch mechanism (`msg_done_latched`) to persist the message completion signal across state transitions.

Each state activates specific control signals that drive external modules such as the hash register file, round logic, and the S-box pipeline. Below, we describe the behavior of each FSM state in detail.

##### IDLE

- **Trigger condition:** Entry on reset or after finalization.
- **Outputs:** All control signals are deasserted.

In this state, the FSM waits for an external request to start processing a new message. Upon receiving `start_msg = 1`, it transitions to the `INIT` state. All outputs are inactive and no module is being driven.

##### INIT

- **Trigger condition:** `start_msg = 1` in `IDLE`.
- **Outputs:**
  - `init_H = 1` → This signal is sent to the `hash_registers` module to initialize the internal hash state  $H[0 \dots 7]$  to their IV (initialization vector) values.

This state resets the internal digest registers to their initial values before beginning message processing. It directly prepares the register bank for a clean hash computation cycle.

#### LOAD\_BYTE

- **Trigger condition:** Completion of `INIT` or return from a completed round.
- **Outputs:**
  - `load_byte = 1` → This signal latches the incoming `msg_byte` into an internal register within `hash_top`, making it available for processing in the datapath.

The FSM waits for a valid input byte. Once `valid_in` is high, it triggers the round computation pipeline and feeds the current byte into the system.

#### ROUND\_EXEC

- **Trigger condition:** Set by `valid_in` from `LOAD_BYTE`.
- **Outputs:**
  - `update_H = 1` → Informs the `hash_registers` to update the  $H[i]$  value using the result coming from the S-box.
  - `round_exec_active = 1` → Drives the `round_tracker` and signals all modules (e.g., shift, `sbox`, `final_hash_unit`) that a transformation round is ongoing.

This state initiates the computation round and activates the core of the hash pipeline. The updated value of  $H[i]$  depends on the result of a sequence: XOR, rotation, and S-box transformation.

#### FINAL\_HASH

- **Trigger condition:** `msg_done_latched = 1` and `round_done = 1`
- **Outputs:**
  - `update_H = 1` → Same as in `ROUND_EXEC`, this updates the hash register using final round logic.
  - `use_counter = 1` → Forces the multiplexer in `hash_top` to use `C_byte[i]` (from the internal length counter) instead of the input message byte.
  - `final_round_active = 1` → Activates the `round_tracker` to prepare the round counter split into bytes and drives `final_hash_unit` for counter-based transformations.

In this state, the system performs finalization rounds using the message length, enhancing collision resistance. Each round uses counter-derived data rather than message input.

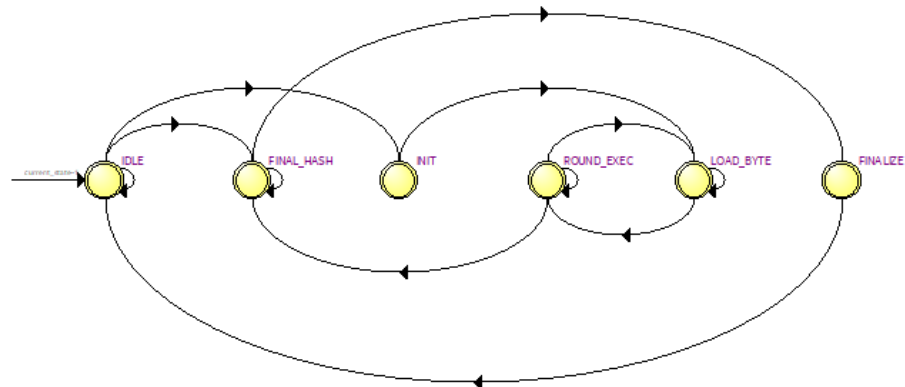
## **FINALIZE**

- **Trigger condition:** `final_round_done = 1`
- **Outputs:**
  - `digest_ready = 1` → Tells `hash_top` to output the current content of  $H[0..7]$  as the final digest.

After all rounds are complete, this signal enables the output logic to propagate the digest through the output port. The FSM then transitions to `IDLE`, resetting for a new hashing session.

**Table 4.1:** State transition table of the *control\_fsm*

| State      | Transition Condition  | Activated Outputs                                     |
|------------|---|---|
| IDLE       | Reset or after finalization. Waits for start_msg = 1.   | None  |
| INIT       | Entered when start_msg = 1 from IDLE.   | init_H = 1  |
| LOAD_BYTE  | Entered after INIT, or after ROUND_EXEC if msg_done_latched = 0 and round_done = 1. Waits for valid_in = 1. | load_byte = 1   |
| ROUND_EXEC | Entered from LOAD_BYTE on valid_in = 1.   | update_H = 1, round_exec_active = 1                   |
| FINAL_HASH | Entered if msg_done_latched = 1 and round_done = 1.   | update_H = 1, use_counter = 1, final_round_active = 1 |
| FINALIZE   | Entered when final_round_done = 1. Automatically transitions to IDLE.                                       | digest_ready = 1                                      |



**Figure 4.1:** Finite State Machine diagram by Quartus

## 4.2 Module `hash_top`

---

### Inputs

- `logic clk`: system clock.
- `logic reset_n`: active-low asynchronous reset.
- `logic start_msg`: input signal that starts a new message.
- `logic [7:0] msg_byte`: byte-wise input of the message to be hashed.
- `logic msg_done`: signal indicating that the message input is complete.
- `logic valid_in`: input byte validity signal.

### Outputs

- `logic round_exec_active`: high when a round is in progress.
- `logic load_byte`: used to latch the message byte into an internal register.
- `logic [7:0] digest [7:0]`: final 64-bit hash digest, output as 8 separate bytes.
- `logic digest_ready`: high when the digest is complete and valid.

### 4.2.1 Functionality

The `hash_top` module serves as the central datapath and top-level coordinator of the hashing process. It integrates and interconnects all submodules, including the control FSM, internal registers, transformation units, and output formatting logic. The goal is to produce a 64-bit hash digest by processing input data byte-by-byte and applying AES-inspired non-linear transformations over multiple rounds.

The module operates across two main phases:

1. **Message absorption phase** — processes one byte at a time from the input stream.
2. **Finalization phase** — injects message length (as an 8-byte counter) to finalize the digest.

The module processes exactly one  $H[i]$  register per cycle, guided by the `i_count` index from the `round_tracker`, and selects either the message byte or counter byte as input depending on the FSM state.

**Dataflow and Control Flow Integration** The following transformations occur per round:

- The control FSM (`control_fsm`) orchestrates the sequence of operations using signals.
- Message bytes arrive externally via `msg_byte`, and are conditionally latched into an internal register `byte_in_reg` when `load_byte` is asserted. Alternatively, is selected `C_byte[i_count]` if `use_counter = 1`.

- The byte is XORed with  $H[(i+1)\%8]$  and rotated left by  $i$  bits in `shift_xor_unit` or `final_hash_unit`, depending on the phase (`use_counter`'s value).
- The rotated value is substituted through the AES S-box (`s_box`), producing a non-linear output used to update  $H[i]$  via `hash_registers`.
- The update occurs when `update_H = 1`, at the index specified by `i_count`.

At the end of the computation, when `digest_ready = 1`, the 8 hash registers are concatenated into the final 64-bit output digest via a packed assignment.

**Output Digest Construction** The final hash is emitted through the output vector `[7:0]digest[0..7]`. Each index corresponds to  $H[0]$  through  $H[7]$ , serialized into a packed structure only when the FSM asserts `digest_ready`. If not ready, the output is forced to zero.

This conditional output design ensures that external modules only read valid digest data.

This ensures seamless switching between message absorption and finalization, without logic duplication.

#### 4.2.2 C[i] Construction (Counter Bytes)

During the finalization phase, the system processes a transformed version of the message length to update each  $H[i]$ . The 64-bit message length, stored in `msg_byte_counter`, is decomposed into 8 individual bytes `C_byte[0..7]` in big-endian order.

This logic is only active when `final_round_active = 1`, as asserted by the FSM during the `FINAL_HASH` state. The counter values serve as non-message-derived inputs to the final transformation, enhancing second-preimage and collision resistance. These bytes are then passed to the `final_hash_unit`, where they are XORed and rotated, similarly to regular message bytes.

**Byte Counter Management** The 64-bit counter `msg_byte_counter` is incremented every time a valid byte is absorbed. It resets to zero on either global reset or at the `INIT` state of the FSM.

This ensures accurate accounting of the total number of bytes processed — a critical input to the final transformation.

### 4.3 Module `hash_registers`

---

#### Inputs

- `logic clk`: clock signal.
- `logic reset_n`: active-low asynchronous reset.
- `logic init_H`: initializes hash register array  $H[0..7]$  with IVs.
- `logic update_H`: allows overwriting one of the  $H$  values.
- `logic [2:0] i_count`: selects which  $H[i]$  to update.
- `logic [7:0] H_update`: value to store in  $H[i\_count]$ .



## Outputs

- `logic [7:0] H_out[0:7]`: full content of the 8 hash registers.

### 4.3.1 Functionality

This module implements a bank of eight 8-bit registers, denoted as  $H[0]$  through  $H[7]$ , which together form the internal 64-bit hash state. These registers are central to the iterative computation of the digest: they are initialized at the beginning of each hashing session and then sequentially updated across rounds during both message absorption and finalization.

**Initialization Phase** The initialization is triggered by the `init_H` signal, which is driven by the `control_fsm` module during the `INIT` state. When asserted, it loads the registers with fixed initial values, referred to as the Initialization Vector (IV). These IVs are hardcoded as a `localparam` array and serve as the starting state of the hashing process.

This operation is synchronous with respect to the rising edge of the `clk` signal and is only allowed when the system is not in reset. If the system is reset asynchronously via `reset_n`, the same IVs are reloaded to ensure deterministic startup behavior.

This design ensures that all rounds start from a known state, a crucial requirement for cryptographic reproducibility and integrity.

**Update Phase** Beyond initialization, the module supports per-cycle selective updates to the internal state registers. This is controlled by two signals:

- `update_H`: asserted by the `control_fsm` during both `ROUND_EXEC` and `FINAL_HASH` states.
- `i_count`: a 3-bit index provided by the `round_tracker`, identifying which of the eight  $H[i]$  registers should be updated.

When `update_H` is high on the rising edge of `clk`, the register at index `i_count` is overwritten with the new value provided on the `H_update` input line. This new value typically comes from the output of the S-box transformation pipeline, which depends on the current byte (either from the message or counter).

This selective write-back mechanism allows the design to process one digest element per cycle, aligning with the round structure of the control FSM. It also enables concurrent streaming of input data and digest accumulation without requiring additional temporary state storage.

At any point, the current contents of all eight registers are available via the output port `H_out[0:7]`, which is used both for feedback during intermediate rounds and for generating the final digest when `digest_ready` is asserted.

## 4.4 Module `shift_xor_unit`

---

### Inputs

- `logic [7:0] H_next`: value of  $H[(i+1) \bmod 8]$ , selected from hash registers.

- `logic [7:0] byte_in`: input byte from message  $M$ .
- `logic [2:0] i_count`: round index  $i$ , used for circular shift.

#### Outputs

- `logic [7:0] shift_xor_out`: result of  $(H_{next} \oplus M[i]) \lll i$ .

#### 4.4.1 Functionality

The `shift_xor_unit` module is a purely combinational block that performs a byte-wise transformation used in each round of the hashing process.

The module performs two key operations:

1. It first computes the bitwise XOR of `H_next` and `byte_in`, which corresponds to the sum in  $GF(2^8)$ . This operation blends the hash state with the current message (or counter) byte:

$$\text{comb\_val} = H_{next} \oplus \text{byte\_in}$$

2. The result, `comb_val`, is then subject to a circular left rotation by  $i$  bits, where  $i = i\_count$ . This rotation is implemented using a bitwise shift-left followed by a shift-right and bitwise OR, effectively rotating the bits:

$$\text{shift\_xor\_out} = ((\text{comb\_val} \ll i) \mid (\text{comb\_val} \gg (8 - i))) \& 8'hFF$$

This ensures a circular shift rather than a logical shift, and the final result is masked with `8'hFF` to maintain 8-bit width. The entire transformation introduces two desirable cryptographic properties:

- **Diffusion:** The XOR spreads bit differences from both the input byte and the hash state.
- **Position-dependence:** The shift by  $i$  introduces index-based variability, ensuring that each  $H[i]$  is affected differently.

### 4.5 Module `final_hash_unit`

---

#### Inputs

- `logic [7:0] H_next`: value of  $H[(i + 1) \bmod 8]$ .
- `wire [7:0] C_byte[0:7]`: the 8-byte counter array, from which  $C[i]$  is extracted.
- `logic [2:0] i_count`: index  $i$  used for addressing and shifting.

#### Outputs

- `logic [7:0] sbx_input_final`: transformed byte to be sent into the AES S-box.

### 4.5.1 Functionality

The `final_hash_unit` is a purely combinational logic module designed for use during the final round of the hash function.

During finalization, each hash register element  $H[i]$  is updated not using a message byte but using the  $i$ -th byte from a counter that encodes the total length of the processed message. This module carries out the following two-step transformation:

1. Compute the XOR between the value of  $H[(i + 1)\%8]$  and the  $i$ -th byte from `C_byte`:

$$\text{xor\_result} = H_{\text{next}} \oplus C_{\text{byte}}[i]$$

This ensures that the final digest depends on the total number of bytes seen, even if the content is otherwise repeated.

2. Apply a circular left rotation of  $i$  bits to `xor_result`, producing:

$$\text{sbox\_input\_final} = ((\text{xor\_result} \ll i) \mid (\text{xor\_result} \gg (8 - i))) \& 8'hFF$$

The rotation adds index-specific diffusion, ensuring that each  $H[i]$  is updated with a unique pattern even if some `C_byte` values are equal.

**Output** The result of this transformation is provided via the output port `sbox_input_final`. This byte is then forwarded to the AES S-box (module `s_box`) to undergo a non-linear substitution, completing the round logic for finalization.

This module plays a crucial role in post-processing the hash state, ensuring that the length of the message contributes to the final digest in a non-linear and position-dependent way — a property important to resist certain classes of extension or collision attacks.

## 4.6 Module `round_tracker`

---

### Inputs

- `logic clk`: system clock used to update both counters synchronously.
- `logic reset_n`: active-low asynchronous reset, used to clear counters to zero.
- `logic round_exec_active`: asserted by the FSM in `ROUND_EXEC` state.
- `logic final_round_active`: asserted by the FSM during the finalization phase (uses counter bytes  $C[i]$  instead of message bytes).
- `logic [2:0] state`: current FSM state, used to validate round completion.
- `logic [2:0] final_hash_state`: expected FSM state for triggering final round termination.

### Outputs

- `logic [2:0] i_count`: index used in each round, cycles from 0 to 7.
- `logic round_done`: raised after each round (8 iterations).
- `logic final_round_done`: high when the final round with the counter is complete.

### 4.6.1 Functionality

The `round_tracker` module is a sequential control unit responsible for managing two key counters during the hash computation: a local per-round index counter  $i$  and a global round counter. These counters enable structured iteration across the eight digest registers ( $H[0]$  through  $H[7]$ ) for each round, and track the overall number of rounds executed across the message and finalization phases.

**1. Round Index Counter `i_count`** This 3-bit counter controls which digest element  $H[i]$  is to be updated in the current cycle. It increments on every clock cycle during `round_exec_active = 1`, cycling from 0 to 7. When `i_count == 7`, the counter wraps to 0, and a full round is considered completed. During `final_round_active`, `i_count` increments in the same way but does not reset any global state.

**2. Round Counter `round_count`** This 6-bit counter tracks how many full rounds have been performed. It increments only when `round_exec_active` is high and `i_count == 7`, indicating the end of a full 8-cycle sequence (i.e., one complete update of  $H[0]$  through  $H[7]$ ). The counter stops incrementing once it reaches the upper bound (36 rounds).

**Completion Signals** Two signals are asserted to synchronize with the FSM:

- `round_done`: high when the current round is the final one of the message phase. Specifically, it checks:

```
round_done = (round_exec_active & round_count == 35 & i_count == 7)
```

This condition ensures that the system completed 36 full rounds (each of 8 updates) — equivalent to processing 288 bytes ( $36 \times 8 = 288$ ).

- `final_round_done`: high during finalization when the FSM is in the `FINAL_HASH` state and  $H[7]$  has just been updated:

```
final_round_done = (state == final_hash_state & i_count == 7)
```

This enables a clean transition to `FINALIZE`, signaling that all final counter-based transformations have been completed.

## 4.7 Module `aes_sbox_lut`

---

### Inputs

- `logic [7:0] sbox_input`: 8-bit input byte used as the index for the AES S-box substitution.

### Outputs

- `logic [7:0] sbox_out`: 8-bit substituted output byte derived from the AES S-box.

#### 4.7.1 Functionality

The `aes_sbox_lut` module implements the AES S-box as a purely combinational logic using a SystemVerilog `function` with an exhaustive `case` statement that maps each of the 256 possible input byte values to their corresponding substituted outputs. This ensures that the module remains synthesizable and memory-free, which is particularly useful for FPGA designs with limited LUT-based memory.

The function `sbox_val` takes the 8-bit input `sbox_input` and deterministically returns the associated S-box value.

This design guarantees that once the input is stable, the output is resolved within the same combinational evaluation cycle, making the substitution delay predictable and consistent.

---

# CHAPTER 5

---

## Functional Verification

---

The verification process was structured to ensure that each module behaves according to its interface and functionality, as defined in Chapter 4. The testing approach followed a bottom-up methodology, where each component was tested individually before being integrated into the full system.

All testbenches were written in SystemVerilog and executed using standard simulation tools. Each testbench was designed to operate independently and focused on the following objectives:

- Apply stimuli that explore normal and boundary-case behaviors.
- Monitor outputs and internal signals through waveform viewers.
- Confirm that timing, control sequences, and state transitions occur as expected.

Most of the validation was based on deterministic test vectors and manual waveform inspection. This choice ensured full transparency and control over each test phase, allowing step-by-step correlation with the expected behavior of the design.

The testbenches provided for individual modules (such as the FSM, round tracker and hash registers) were developed as internal verification tools. These modules were not required to be independently tested as part of the mandatory deliverable, so waveform diagrams are not included. However, we still implemented and executed these testbenches to validate the expected behavior of each unit and to facilitate easier debugging during integration.

## 5.1 Verification of Individual Modules

---

### 5.1.1 hash\_registers

The `tb_H[i].sv` testbench validated two behaviors: (1) the correct initialization of the internal registers to the predefined IV when `init_H` is asserted, and (2) the selective update of a single register at a time, based on the value of `i_count` and when `update_H` is high. Each update was confirmed by observing the change in the corresponding `H_out[i]` output while all others remained constant.

### 5.1.2 round\_tracker

The `tb_round_tracker.sv` testbench simulated a sequence of active hashing rounds. The `round_exec_active` signal was pulsed for 8 consecutive cycles to simulate one full digest update cycle and verifying the correct increment of `i_count` from 0 to 7 and the corresponding update of `round_count`. Additionally, the flags `round_done` and `final_round_done` were monitored to ensure they assert only under the expected conditions, based on state matching and counter completion.

### 5.1.3 control\_fsm

The `tb_fsm.sv` testbench focused on verifying the sequence of state transitions and the correct activation of control signals for each state. The testbench manually stimulated the FSM with transitions such as `start_msg`, `valid_in`, `msg_done`, `msg_done_latched`, `round_done`, and `final_round_done` to emulate realistic operation flow.

The following properties were verified:

- The FSM correctly transitions from IDLE to INIT on `start_msg`.
- From INIT, it moves to LOAD\_BYTE and asserts `init_H` as expected.
- Once received a valid byte and so `valid_in` is correctly asserted, ROUND\_EXEC phase is triggered.
- `update_H` and `round_exec_active` are asserted during ROUND\_EXEC, and the module loops correctly to FINAL\_HASH depending on both `round_done` and `msg_done_latched`.
- FINAL\_HASH, asserts `use_counter`, `update_H` and `final_round_active`.
- Finally, when `final_round_done` is detected, the FSM reaches the FINALIZE state and sets `digest_ready`. Then automatically loops to IDLE.

Each transition and output activation was confirmed via waveform inspection and verified to match the FSM specification. The testbench also allowed edge-case testing, such as toggling reset at different states and testing delayed valid inputs.

During the writing of the testbench, one minor issue encountered was the synchronization between `msg_done` and `round_done`. Initially, `msg_done` was deasserted too quickly, preventing the FSM from entering the FINAL\_HASH state. This was resolved by holding `msg_done` high for a few extra cycles, allowing the FSM to detect and latch it properly.

### 5.1.4 hash\_top

The `tb_hash_top.sv` testbench was the most crucial verification step, as it allowed us to evaluate the full datapath — from message input to final digest output. We started by defining a simple infrastructure: 10ns clock, reset logic, and the DUT wiring.

The stimulus logic is based on feeding each byte of the input message to `msg_byte`, one at a time, waiting for the internal round to complete before sending the next. Here's the snippets of the testbench:

```
1 foreach (message_str[i]) begin
2     wait (round_exec_active==0);
3     @(negedge clk);
4     msg_byte = message_str[i];
5     valid_in = 1;
6     @(posedge load_byte); // core ready
7     @(posedge clk);       // safe latching
8     valid_in = 0;
9 end
```

We honestly struggled with this portion for quite a while. The problem wasn't the logic itself, but understanding how `round_exec_active` and `load_byte` interacted. At first, we were asserting `valid_in` at the wrong time, and sometimes the FSM would skip characters or get stuck waiting.

```
1 @(negedge clk);
2 msg_done = 1;
3 msg_byte = 8'hXX;
4 @(posedge clk);
5 msg_done = 0;
6 valid_in = 0;
7
8 @(posedge digest_ready);
9 @(posedge clk);
```

After the input was sent, we asserted `msg_done`, then waited for `digest_ready` to become high, ensuring the digest was finalized.

To display the result, we used this helper task:

```
1 task print_digest;
2     int i;
3     $write("Digest: ");
4     for (i = 0; i < 8; i++) begin
5         $write("%02h ", digest[i]);
6     end
7     $display();
8 endtask
```

At the beginning, we had a subtle bug where the message got reversed due to a misunderstanding of how `foreach` indexes the string. It took us an entire afternoon to realize it wasn't a bug in the hashing logic but in how we fed the message.

Running the test with a short string allowed us to manually check that the same input consistently produced the same hash. It was quite satisfying to see the correct behavior after debugging all the FSM and signal-timing quirks.

This testbench served not only as a validation tool but also helped us understand how each control signal had to align exactly with the internal state machine of the core. Timing was everything — especially in such a pipelined structure.



### 5.1.5 Waveform Analysis

To further validate our design, we ran the full testbench under two key scenarios.

The message to hash is:  
Hash: 0ab29b3352ee346b

Figure 5.1: Digest for input message length = 0 by Golden Model

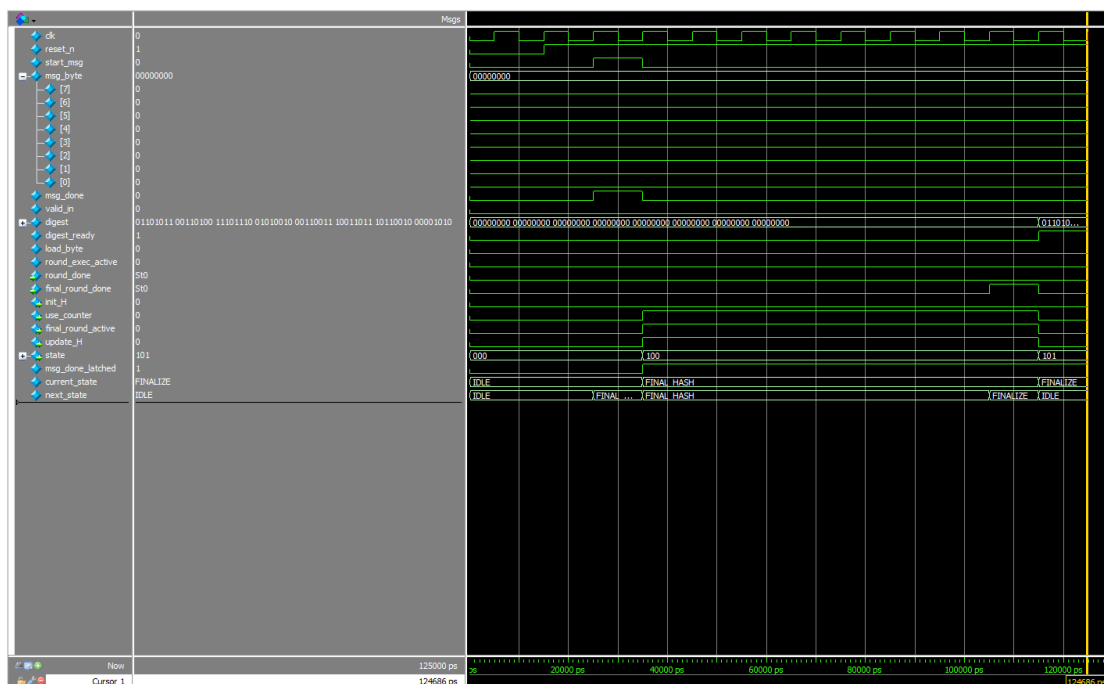


Figure 5.2: Waveform for input message of length 0 by ModelSim

```
VSIM 18> run -all
# 0a b2 9b 33 52 ee 34 6b
# == Fine test hash_top ==
```

Figure 5.3: Digest for input message length = 0 by ModelSim

The message to hash is: Hello World  
Hash: d34c497e35dee350

Figure 5.4: Digest for input message "Hello World" by Golden Model

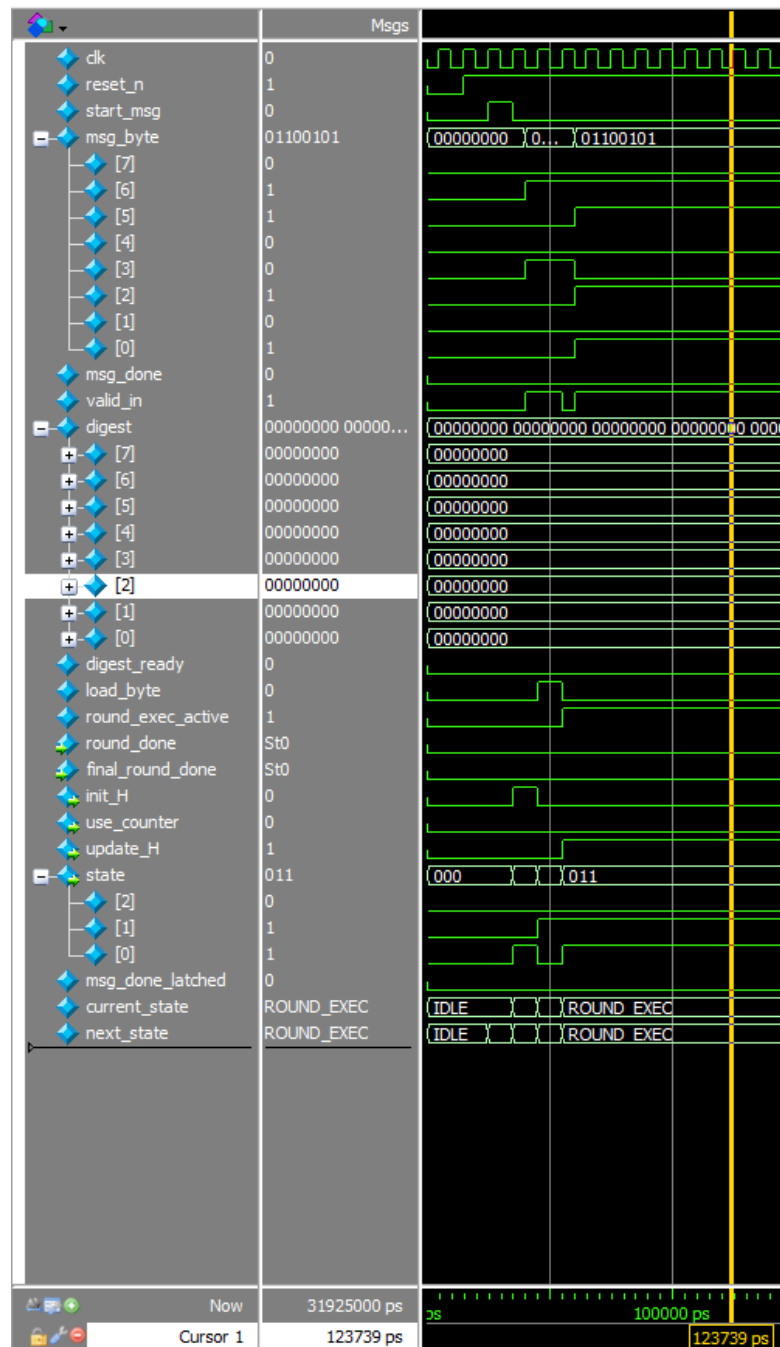
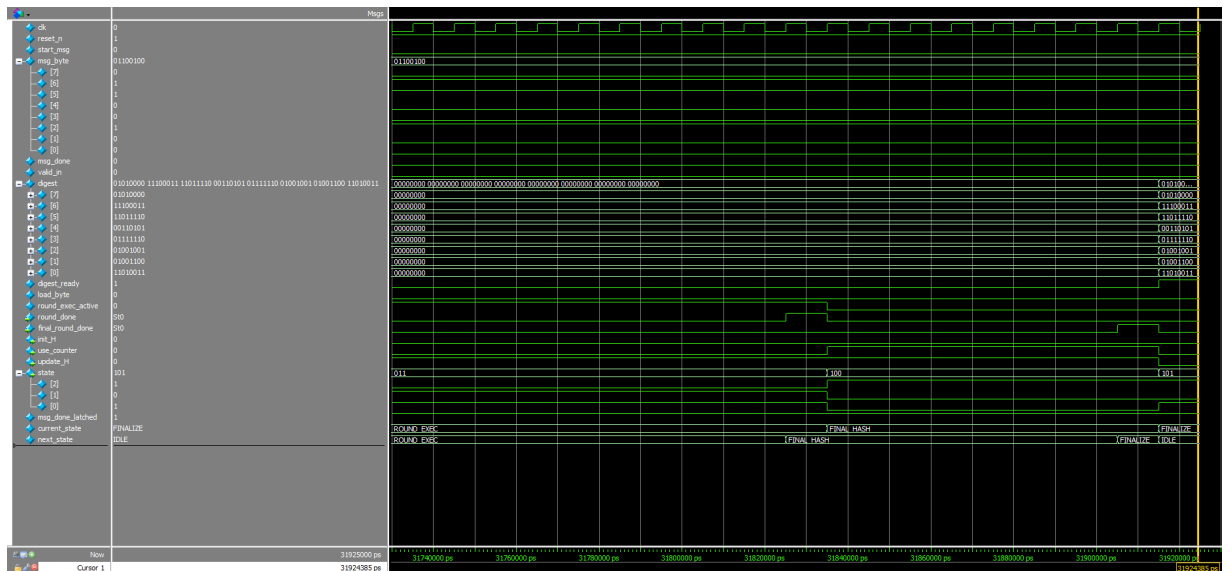


Figure 5.5: Waveform for input message "Hello World"



**Figure 5.6:** Waveform for input message "Hello World"

```
VSIM 15> run -all
# d3 4c 49 7e 35 de e3 50
# === Fine test hash_top ===
```

**Figure 5.7:** Digest for input message "Hello World" by ModelSim

---

# CHAPTER 6

---

## FPGA Implementation Results

---

Following the successful implementation and functional verification of the designed circuit, the complete RTL design flow was carried out using the Intel Quartus Prime software suite. This process began with logic synthesis and proceeded with the fitting of the design onto the physical architecture of the target device. The final stage involved performing a comprehensive Static Timing Analysis (STA) to assess the design's adherence to defined timing constraints and to identify any potential critical paths within the combinational logic. All stages of the design flow were executed with the 5CGXFC9D6F27C7 device from Intel's Cyclone V family as the target, chosen for its alignment with both the structural complexity and performance requirements of the implemented architecture.

### 6.1 Analysis and Synthesis

---


To initiate the Analysis and Synthesis phase—whose objective is to generate the logical representation of the circuit—we carried out a preliminary assignment of virtual pins for all input and output signals (in total 79) mapped in this way:

- 1 pin for the signal `digest_ready`,
- 1 pin for the signal `load_byte`.
- 1 pin for the signal `msg_done`,
- 1 pin for the signal `reset_n`,
- 1 pin for the signal `round_exec_active`,
- 1 pin for the signal `start_msg`,

- 1 pin for the signal valid\_in.
- 64 pins for the signal digest,
- 8 pins for the signal msg\_byte,

|    | Status | From    | To                | Assignment Name | Value   | Enabled | Entity   | Comment | Tag |
|----|--------|---------|-------------------|-----------------|---------|---------|----------|---------|-----|
| 1  | ✓ Ok   |         | out digest_ready  | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 2  | ✓ Ok   |         | out load_byte     | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 3  | ✓ Ok   |         | in msg_done       | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 4  | ✓ Ok   |         | in reset_n        | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 5  | ✓ Ok   |         | out round...ctive | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 6  | ✓ Ok   |         | in start_msg      | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 7  | ✓ Ok   |         | in valid_in       | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 8  | ✓ Ok   |         | out digest        | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 9  | ✓ Ok   |         | in msg_byte       | Virtual Pin     | On      | Yes     | hash_top |         |     |
| 10 | ✓ Ok   |         | in clk            | Location        | PIN_R20 | Yes     |          |         |     |
| 11 |        | <<new>> | <<new>>           | <<new>>         |         |         |          |         |     |

**Figure 6.1:** Assignment Pins

| Flow Summary   |   |
|--|---|
|  <<Filter>> |   |
| Flow Status  | Successful - Sat Jun 21 18:26:26 2025           |
| Quartus Prime Version  | 24.1std.0 Build 1077 03/04/2025 SC Lite Edition |
| Revision Name  | hash_top  |
| Top-level Entity Name  | hash_top  |
| Family   | Cyclone V                                       |
| Device   | 5CGXFC9D6F27C7                                  |
| Timing Models  | Final   |
| Logic utilization (in ALMs)  | 287 / 113,560 ( < 1 % )                         |
| Total registers  | 282   |
| Total pins   | 1 / 378 ( < 1 % )                               |
| Total virtual pins   | 79  |
| Total block memory bits  | 0 / 12,492,800 ( 0 % )                          |
| Total DSP Blocks   | 0 / 342 ( 0 % )                                 |
| Total HSSI RX PCSs   | 0 / 9 ( 0 % )                                   |
| Total HSSI PMA RX Deserializers  | 0 / 9 ( 0 % )                                   |
| Total HSSI TX PCSs   | 0 / 9 ( 0 % )                                   |
| Total HSSI PMA TX Serializers  | 0 / 9 ( 0 % )                                   |
| Total PLLs   | 0 / 17 ( 0 % )                                  |
| Total DLLs   | 0 / 4 ( 0 % )                                   |

**Figure 6.2:** Flow Summary

At the end of the analysis and synthesis phases, we identified a non-compliant implementation of the S-Box module indicated by Quartus warnings as missing connections. These warnings stemmed from our original description, which modeled the LUT as an unpacked array of 256 cells enveloped by a combinational wrapper to encapsulate its logic and streamline the project's top-level hierarchy; although functionally accurate this approach prompted the tool to infer a RAM implementation rather than a read-only memory.

```
1 Warning (10030): Net "sbox.data_a" at sbox.sv(44) has no driver or initial value ,  
    using a default initial value '0'  
2  
3 Warning (10030): Net "sbox.waddr_a" at sbox.sv(44) has no driver or initial value ,  
    using a default initial value '0'  
4  
5 Warning (10030): Net "sbox.we_a" at sbox.sv(44) has no driver or initial value , using  
    a default initial value '0'
```

In order to overcome this challenge we refactored the LUT such that the synthesis now correctly implements the LUT as a ROM, thereby suppressing the warnings and achieving the desired design.


## 6.2 Fitter

---

The Fitter phase follows immediately after the Analysis & Synthesis stage and is responsible for mapping the logical netlist onto the physical architecture of the target device. This involves executing the classic place-and-route procedures, which are essential for determining the circuit's actual timing performance. If the design fails to meet timing requirements at this stage, it will not be able to operate at the intended clock frequency.

In addition, the Fitter also produces a detailed report on resource usage such as the number of Adaptive Logic Modules (ALMs) used, which plays a crucial role in guiding further optimization strategies.

Based on the Fitter summary, the core occupies only a minimal portion of the available logic resources suggesting that the chosen device is significantly over-provisioned for the current design.

| Fitter Summary   |   |
|--|---|
|  <<Filter>> |   |
| Fitter Status  | Successful - Sat Jun 21 22:24:16 2025           |
| Quartus Prime Version  | 24.1std.0 Build 1077 03/04/2025 SC Lite Edition |
| Revision Name  | hash_top  |
| Top-level Entity Name  | hash_top  |
| Family   | Cyclone V                                       |
| Device   | 5CGXFC9D6F27C7                                  |
| Timing Models  | Final   |
| Logic utilization (in ALMs)  | 287 / 113,560 ( < 1 % )                         |
| Total registers  | 282   |
| Total pins   | 1 / 378 ( < 1 % )                               |
| Total virtual pins   | 79  |
| Total block memory bits  | 0 / 12,492,800 ( 0 % )                          |
| Total RAM Blocks   | 0 / 1,220 ( 0 % )                               |
| Total DSP Blocks   | 0 / 342 ( 0 % )                                 |
| Total HSSI RX PCSs   | 0 / 9 ( 0 % )                                   |
| Total HSSI PMA RX Deserializers  | 0 / 9 ( 0 % )                                   |
| Total HSSI TX PCSs   | 0 / 9 ( 0 % )                                   |
| Total HSSI PMA TX Serializers  | 0 / 9 ( 0 % )                                   |
| Total PLLs   | 0 / 17 ( 0 % )                                  |
| Total DLLs   | 0 / 4 ( 0 % )                                   |

**Figure 6.3:** *Fitter Summary*

### 6.3 Timinig Analysis

Upon completion of the synthesis phases, we focused on Static Timing Analysis. We employed an initial Synopsys Design Constraints (.sdc) file, in which we defined a clock with a 7 ns period (  $\approx 142.86$  MHz) and minimum and maximum delay (calculated as 10 % and 20 % of the clock period) constraints on all inputs and outputs. The circuit synthesized with a 7 ns clock period does not satisfy the specified timing constraints because several architectural choices have introduced critical path delays. The main issues are:

- **Inter-path Competition:** this phenomenon occurs when a shared logic node, e.g. `sbox_input`, is traversed by multiple timing paths with heterogeneous propagation-delay requirements. The placer therefore seeks a globally optimal placement that can penalise the most critical paths.
- **Long Combinational Path:** certain paths, such as the one from `round_tracker` to `hash_register`, contain an excessive number of combinational stages between the source and destination registers. Even with near-optimal placement and routing, closing timing for these paths remains challenging.
- **Unbalanced Combinational Logic:** This criticality stems from a structural imbalance between the analysed path and the upstream (source) or downstream (destination) logic blocks, leading to a further degradation of the path's timing performance. In this case as well, the timing path extending from the `round_tracker` to the `hash_register` can be considered.

The cumulative delay introduced by these factors causes the design to violate the default timing constraints in both *Slow* process corners. The worst-case scenario is observed under the *Slow* condition at 1100 mV and 85°C, where a negative setup slack is reported on the critical path.6.1.

**Table 6.1:** Static Timing Analysis measurement with a clock period of 7 ns.

| Condition          | Slack Setup | Slack Hold |
|--------------------|-------------|------------|
| Slow 1100 mV 85 °C | -2.304      | 0.387      |
| Slow 1100 mV 0 °C  | -2.251      | 0.370      |
| Fast 1100 mV 85 °C | 2.081       | 0.181      |
| Fast 1100 mV 0 °C  | 2.273       | 0.172      |

Upon completion of the measurements, it is possible to estimate the minimum clock period required for the circuit to satisfy all timing conditions. Accordingly, we considered the worst-case operating conditions; using the known clock period in conjunction with the setup and hold slack values, the resulting propagation delay is estimated at approximately 9.304 ns.

$$t_{pd} \leq T_{clk} - t_s$$

Consequently, we selected a clock period greater than this value introducing an additional margin—expressed in picoseconds—to accommodate potential routing variations between compilations approximately to 9.7 ns. With this updated clock period,



while retaining the previously defined minimum and maximum delay constraints, the circuit satisfies the timing requirements under all conditions analyzed in table 6.2.

**Table 6.2:** *Static Timing Analysis measurement with a clock period of 9.7ns.*

| <b>Condition</b>   | <b>Slack Setup</b> | <b>Slack Hold</b> |
|--------------------|--------------------|-------------------|
| Slow 1100 mV 85 °C | 0.121              | 0.347             |
| Slow 1100 mV 0 °C  | 0.041              | 0.276             |
| Fast 1100 mV 85 °C | 2.684              | 0.176             |
| Fast 1100 mV 0 °C  | 3.091              | 0.164             |

## 6.4 Conclusion

---

Based on the results obtained, we explored several optimization strategies aimed at increasing the circuit's operating frequency.

Despite numerous implementation attempts, it became clear that achieving meaningful improvements would require a fundamental redesign of certain modules. Given that the existing design was already stable, met all functional and structural requirements, we ultimately decided not to pursue further modifications.

In conclusion, timing analysis confirmed that the circuit operates correctly with a clock period of  $9.7 \text{ ns} \approx 103.09 \text{ MHz}$ .