



UNIVERSITY OF PISA
MASTER'S DEGREE IN CYBERSECURITY

FULL HASH ALGORITHM V3 (DES S-BOX)

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Bernardini Diego
Cisternini Giovanni

Academic year 2024/2025

Contents

1	Project Specifications	1
1.1	Introduction	1
1.2	Hardware Design Specifications:	3
1.3	Project phases	3
1.4	System Verilog	4
2	High-level Model	5
2.1	High-Level C Model for RTL Development and Verification	5
3	RTL Design	8
3.1	Description of the RTL design architecture	8
4	Interface Specifications and Expected Behavior	12
4.1	Module <code>full_hash</code>	12
4.2	Module <code>control_part</code>	15
4.3	Module <code>operative_part</code>	20
4.4	Future upgrades	23
5	Functional Verification	24
5.1	<code>testbench.sv</code>	24
6	FPGA Implementation Results	34
6.1	Synthesis and Timing Analysis Phase	34

CHAPTER 1

Project Specifications

1.1 Introduction

This document describes the design and implementation of the "**Full Hash Algorithm v3**", a hardware hashing module based on the utilization of a S-box derived from the DES algorithm. The project's objective is the generation of a 32-bit digest, composed of eight nibbles (4-bit vectors) $H[i]$ for $i \in [0, 7]$, where $H[0]$ represents the most significant nibble.

The module's operational specifications are defined by two distinct sequences of operations, applied iteratively based on the processing phase.

Initialization:

The digest nibbles $H[i]$ are initialized with the following hexadecimal values:

H[0]	H[1]	H[2]	H[3]	H[4]	H[5]	H[6]	H[7]
4'h3	4'hF	4'hA	4'h1	4'hE	4'hF	4'h2	4'h3

Table 1.1: Initialization Values for Hash Nibbles

First Phase:

For each byte M of the input message (8-bit ASCII code of a character), the module performs the following operations, iterated 12 times ($r \in [0, 11]$) for each nibble $H[i]$ ($i \in [0, 7]$):

$$H[i] = (H[(i + 1) \pmod{8}] \oplus S(M_6)) \ll \lfloor i/2 \rfloor$$

Where:

- $(\text{mod } n)$: Modulo operator by n .
- \oplus : XOR operator.
- $\ll n$: Left circular shift (rotation) by n bits.
- $\lfloor x \rfloor$: Floor function (applied to argument x).
- M_6 : A 6-bit vector derived from M using the following compression function:

$$M_6 = \{M[3] \oplus M[2], M[1], M[0], M[7], M[6], M[5] \oplus M[4]\}$$

(Here, $\{ \dots \}$ denotes the concatenation operator, and $M[n]$ is the n -th bit of $M[7 : 0]$).

- $S(\cdot)$: The DES algorithm's S-box transformation, implemented as a Look-Up Table (LUT) for performance optimization. The S-box accepts 6 input bits (outer bits define the row, central bits define the column) and produces 4 output bits.

Second Phase:

Once the last message byte has been processed, the module performs the following operations for each nibble $H[i]$ ($i \in [0, 7]$):

$$H[i] = (H[(i + 1) \text{ (mod } 8)] \oplus S(C_6[i])) \ll \lfloor i/2 \rfloor$$

Where:

- C : A 64-bit counter representing the actual message length in bytes (e.g., $C = 1$ for a 1-byte message, not 0).
- $C_6[i]$: A 6-bit vector obtained from the i -th byte $C[i]$ of the counter C ($C[i] = C[i][7 : 0]$) using the following compression function:

$$C_6[i] = \{C[i][7] \oplus C[i][1], C[i][3], C[i][2], C[i][5] \oplus C[i][0], C[i][4], C[i][6]\}$$

Digest computation is also possible for empty messages; in this case, only Phase 2 operations are performed, with $H[i]$ nibbles maintained at their initialization values and $C = 0$.

1.2 Hardware Design Specifications:

The module includes the following hardware interface specifications:

- An asynchronous, active-low reset flag.
- Two flags for data synchronization:
 - F_rtr: to signal waiting for a new byte.
 - F_dr: to indicate when the new data is ready.
- One End-of-File flag, to determine when the message has terminated.
- A message byte counter, useful for the second phase of operations.
- One flag indicating the end of operations and that the digest is ready.

This architecture aims to provide a robust and efficient hash algorithm, with a focus on hardware security and data integrity.

1.3 Project phases

The project development followed a structured methodological approach comprising four main phases:

- **High-level Modeling:** Creation of a high-level model in C++ (`high_level_model.cpp`). This model simulated the intended functionalities of the hash module, serving as a fundamental reference for subsequent RTL design and verification phases.
- **RTL Design:** Definition of the module's logical architecture and decomposition into sub-components. This phase was crucial for identifying the necessary modules prior to writing the SystemVerilog code.
- **Simulation and Test:** Implementation of the design in SystemVerilog, segmenting the code into specific modules. The module was then subjected to a rigorous testing campaign using a dedicated testbench, covering multiple operational scenarios.
- **FPGA Implementation:** Synthesis and verification of the hardware module on an Intel Cyclone V FPGA (5CGXFC9D6F27C7) using the Intel Quartus Prime development environment. This phase was complemented by a Static Timing Analysis (STA) to validate timing performance.

1.4 System Verilog

The SystemVerilog code is structured into the following modules:

- **full_hash_v3.sv:** Represents the top-level module of the design, responsible for interfacing with external modules. It integrates two main sub-modules and one auxiliary support module.
- **control_part.sv:** One of the two main sub-modules, tasked with generating and managing all control signals. It synchronizes operations with external modules and manages the advancement of the hashing process's execution phases.
- **operative_part.sv:** The other main sub-module, composed of several combinational units that perform the logical operations defined by the algorithm. This module also implements the message byte counting system.
- **Combinational Modules:** These auxiliary modules perform specific functions within `operative_part.sv`:
 - `c6.sv`: for generating the C6 vector.
 - `m6.sv`: for generating the M6 vector.
 - `sbox.sv`: implements the DES S-box as a Look-Up Table (LUT).
 - `xor_shift.sv`: for executing XOR and circular shift operations.
- **inverter.sv:** A combinational module added during the testing phase, whose function is to invert the byte order in the digest output.

CHAPTER 2

High-level Model

2.1 High-Level C Model for RTL Development and Verification

To support the development of the Register-Transfer Level (RTL) phase schematic and to provide a mechanism for functional verification, a high-level model in C, named `high_level_model.c`, has been implemented. This model is structured into distinct functions, each reflecting the specific requirements of the design specification.

Model Components

`main`

This function serves as the primary entry point of the program. Initially, it acquires a message from standard input via `my_getline` and determines its length (`C`). A check on the message length is performed: if `C` is greater than zero, the function iterates through each byte of the message (`messaggio[M_byte]`), invoking the `prima_operazione` function for each byte.

Subsequently, the `seconda_operazione` and `compatta_H` functions are called sequentially. Upon completion of the processing, the final digest (`Hf`) is printed in both decimal and hexadecimal formats. Finally, the dynamically allocated memory for `messaggio` is freed using `free`, and the function returns 0 to indicate successful termination. The specific case of an empty message (`C == 0`) results in bypassing the execution of `prima_operazione`.

```

1 int main(int argc, char* argv[]){
2
3     size_t len_max;
4     printf("Inserisci_un_messaggio:_");
5     my_getline(&messaggio, &C, stdin);
6     len_max = C;
7
8     if(C!=0){
9         for(int M_byte = 0; M_byte<len_max;M_byte++){
10             prima_operazione(messaggio[M_byte]);
11         }
12     }
13
14     seconda_operazione();
15     compatta_H();
16
17     printf("Digest_Intero:_%u,_Digest_Hex:_%#X", Hf, Hf);
18     free(messaggio);
19     return 0;
20 }

```

prima_operazione

This function processes a single input byte (M). The operation is structured with two nested for loops. The outer loop (r) iterates for 12 rounds. The inner loop (i) executes 8 iterations for each round. Within each iteration, a temporary value a is calculated by applying a rotation (rotate_lower4) to the result of an XOR operation between an element of the H array (specifically H[(i+1)%8]) and the result of the funzione_S function, which operates on a value generated by crea_M6(M). The calculated value a is then assigned to H[i], updating the algorithm's state.

```

1 void prima_operazione(uint8_t M){
2
3     for(int r=0; r<12;r++){
4         for(int i=0;i<8;i++){
5             uint8_t a = rotate_lower4(
6                 (H[(i+1)%8]^funzione_S(crea_M6(M))),
7                 i
8             );
9             H[i] = a;
10         }
11     }
12 }

```


seconda_operazione

This function represents the final phase of the algorithm. It consists of a `for` loop that iterates 8 times (from `i = 0` to 7). In each iteration, the element `H[i]` of the `H` array is updated. The new value is the result of a rotation (`rotate_lower4`) applied to the output of an XOR operation. The XOR operation combines the `H[(i+1)%8]` element with the result of the `funzione_S` function, which in turn takes as an argument a value derived from `crea_C6(restituisce_C_i(i))`.

```
1 void seconda_operazione() {  
2  
3     for(int i = 0; i<8;i++){  
4         H[i] = rotate_lower4((H[(i+1)%8]^funzione_S(crea_C6(restituisce_C_i(i  
5             )))),i);  
6     }  
7 }
```

CHAPTER 3

RTL Design

3.1 Description of the RTL design architecture

Block diagrams

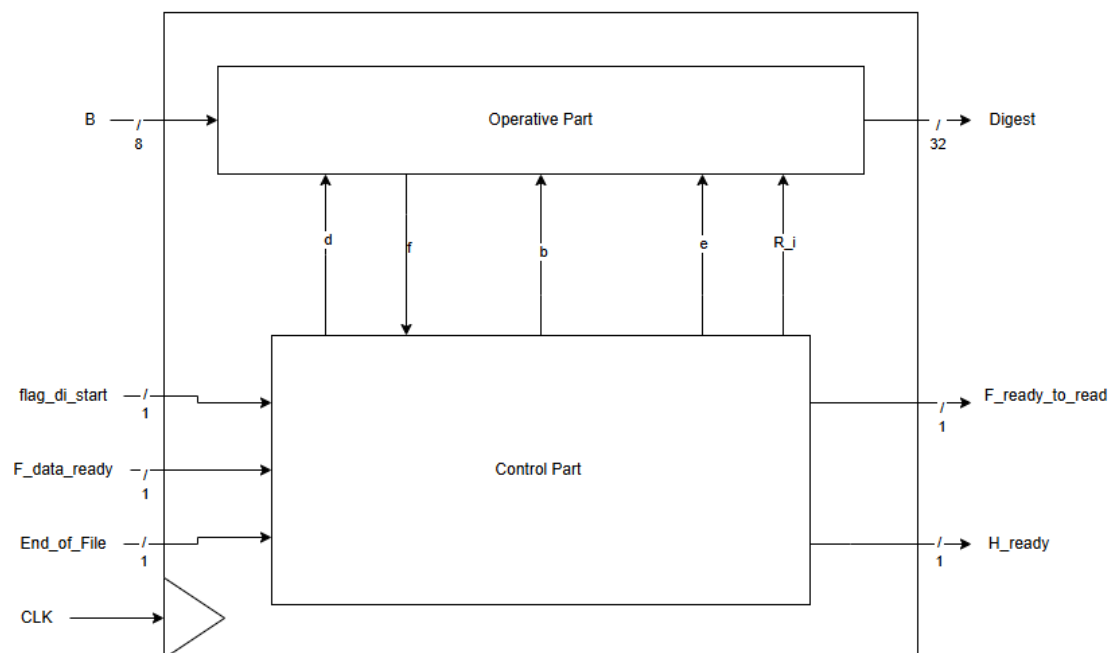


Figure 3.1: High-level block diagram.

Table 3.1: Legend

Letter	Logic name
b	validate_input
d	validate_R_H
e	switch_operation
f	case_R_C_0

The roles played by the two modules will be described in the following sections, while the characteristics and functional details of the various elements will be described in the next chapters.

Operative Part

The Operative Part represents the datapath of the system and it is responsible for the core processing of the input data.

- **Inputs:**
 - B [7:0] – 8-bit input data
- **Outputs:**
 - Digest [31:0] – 32-bit output result
- **Connected to Control Part via:**
 - Control signals: b, e, R_i
 - Feedback signals: d, f

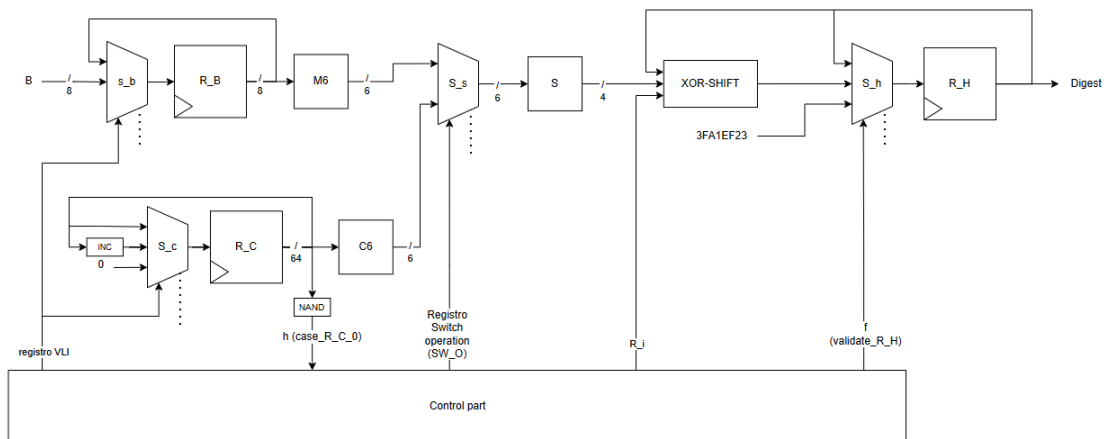


Figure 3.2: *Operative part circuit*

Control Part

The Control Part manages the system states, coordinates the data flow, and drives the Operative Part via a Finite State Machine (FSM).

• **Inputs:**

- flag_di_start – Start signal
- F_data_ready – Data ready flag
- End_of_File – End of input data
- CLK – System clock

• **Outputs:**

- F_ready_to_read – Ready to read new data
- H_ready – Final result ready

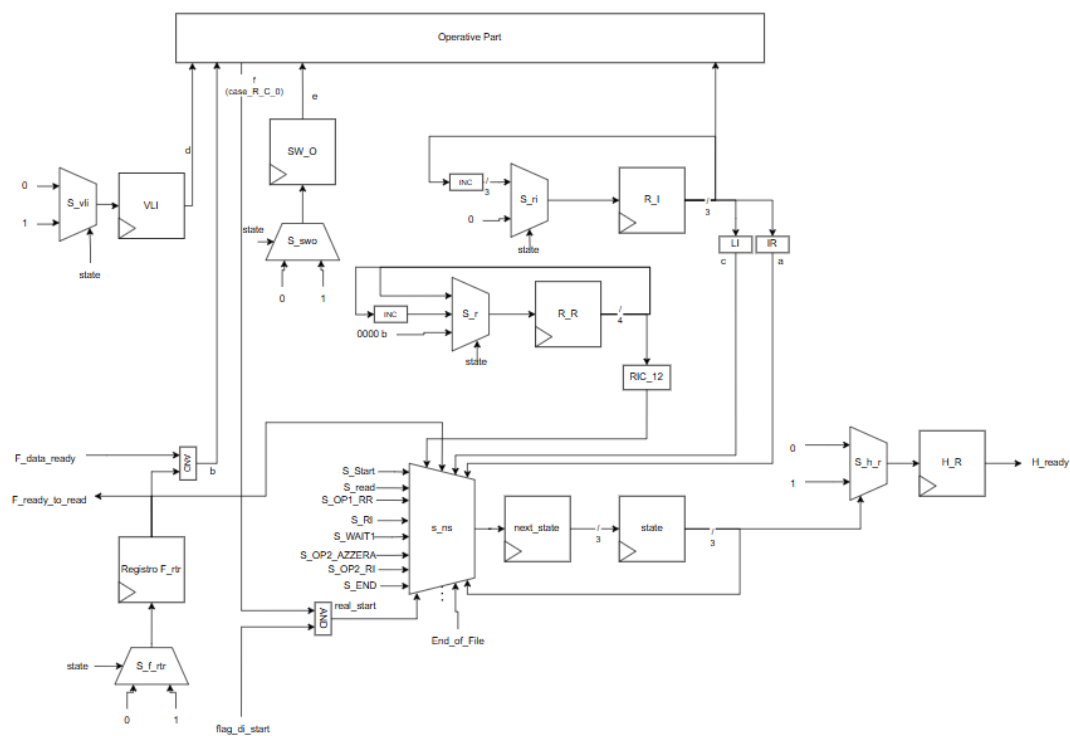
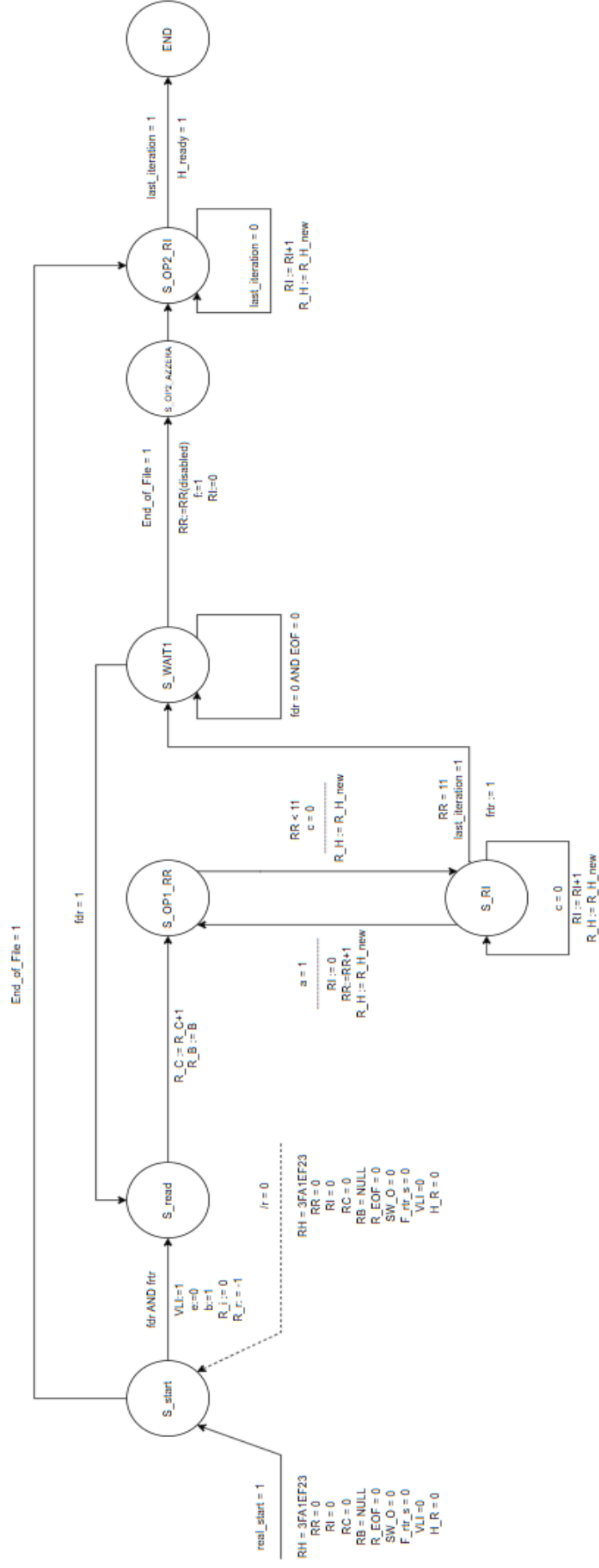


Figure 3.3: Control part circuit

Table 3.2: Legend

Letter	Logic name
a	incrementation _round
b	validate _input
c	last _iteration
d	validate _R _H
e	switch _operation
f	case _R _C _0

FSM diagrams



CHAPTER 4

Interface Specifications and Expected Behavior

4.1 Module `full_hash`

The `full_hash` module represents the top-level of the system and is responsible for orchestrating the complete hash calculation process. It integrates two main sub-modules: `control_part` for flow management and control signals, and `operative_part` for executing the hashing operations on the data. An additional module, `inverter`, handles a post-processing step of the final digest.

Inputs

The `full_hash` module receives the following input signals:

- **clk**: Main clock signal. It is used to synchronize all operations and state advancements within all registers of the system.
- **rst_n**: Asynchronous reset signal, active low. When this signal is 0, it forces a complete reset of the entire system, bringing the Finite State Machine (FSM) to its initial state and resetting all counters.
- **start**: General start signal. This signal is used to initiate the entire hashing process.
- **Byte**: 8-bit data input. It represents the current data byte that needs to be processed by the system.
- **End_of_File**: "End of File" Flag. This signal is indicated by an external module to signal that there is no more data to process, meaning that the last byte of the file has been processed or is currently being processed.

- **F_dr**: "Data Ready" Flag. This signal comes from an external module and indicates that a new data byte is available on the `Byte` input and can be read by the system.

Outputs

The `full_hash` module generates the following output signals:

- **R_h**: 32-bit output register. It contains the final hash digest calculated at the end of the hashing process. This value is the result of processing all bytes and the final inversion.
- **F_rtr**: "Ready To Read" Flag. This signal indicates to the external module that the system is ready to receive a new data byte.
- **H_ready**: "Hash Ready" Flag. This signal indicates that the entire hashing process has been successfully completed and that the final value present in `R_h` is valid and can be used.

Internal Circuitry and Interconnections between Modules

The `full_hash` module uses a series of internal wires to connect and allow its sub-modules to communicate:

- **case_rc0**: Connection wire. This signal originates from the `operative_part` module and is sent to the `control_part` module. It is used for a conditioned start logic, specifically to prevent unwanted multiple activations of the hashing process via the `start` signal.
- **sw_o**: "Switch Operation" connection wire. Generated by the `control_part` module and sent to the `operative_part` module. It instructs the operative module to switch from "Operation 1" logic to "Operation 2", managing the different phases of the hashing algorithm.
- **vi**: "Validate Input" connection wire. Generated by the `control_part` module and sent to the `operative_part` module. It signals that the current input byte is valid and can be processed by the `operative_part` module.
- **v_R_H**: "Read/Handshake Validation" connection wire. An internal validation signal generated by the `control_part` module and sent to the `operative_part` module. It is used in specific phases of the operations to validate the read or handshake state.
- **i**: Connection wire for the iteration counter (3 bits). It comes from the `control_part` module and is sent to the `operative_part` module. It corresponds to the `R_i` counter and manages the internal iterations within a "round" of the hashing operations.
- **digest**: Connection wire (32 bits). Represents the output of the `operative_part` module. It contains the hash digest calculated by the operations, before the final inversion is applied.

- **real_digest**: Connection wire (32 bits). Represents the output of the `inverter` module. It contains the final hash digest after its bits have been inverted. This is the value that will then be assigned to the `R_h` output of the `full_hash` module.

Internal Modules and their Function

The `full_hash` module integrates the following sub-modules:

- **control_part** (`control`):
 - **Purpose**: This module implements the Finite State Machine (FSM) that manages the overall control flow of the hashing process. It is the "brain" that determines the sequence of operations.
 - **Main Functions**: It generates the necessary control signals for the operative module (`switch_operation`, `validate_input`, `validate_R_H`), manages the round and iteration counters (`R_i`), and orchestrates the transitions between the different phases of the process (data reading, Operation 1, Operation 2, wait, end of process). It receives feedback from the operative module (`case_rc0`) and external inputs (`F_dr`, `End_of_File`, `start`).
- **operative_part** (`operative`):
 - **Purpose**: This module is responsible for executing the actual hashing operations (OP1 and OP2) on the input data. It is the "engine" that processes the data.
 - **Main Functions**: It works on the received bytes (`Byte`) and the values of the iteration counters (`R_i`) to calculate the partial or final hash digest. Its operations are guided by the control signals received from the `control_part` module (`validate_input`, `switch_operation`, `validate_R_h`). It produces an intermediate digest (`digest`) and provides a feedback signal (`case_R_c_zero`) to the control module.
- **inverter** (`invertitore`):
 - **Purpose**: This is a simpler module that performs a post-processing operation on the hash digest.
 - **Main Functions**: It receives the hash digest produced by the `operative_part` module (`H_in`) and bitwise inverts its value, producing the `real_digest`. This inverter was introduced due to an implementation error, as the digest in the high-level test module was written with a little-endian architecture [LSB:MSB], while in the module it was implemented with a big-endian architecture [MSB:LSB].

In summary, the `full_hash` module coordinates a sequential data processing pipeline, delegating flow control to `control_part`, operation execution to `operative_part`, and a final transformation phase to `inverter`, to produce a final cryptographic hash.

4.2 Module `control_part`

This `control_part` module implements a Finite State Machine (FSM) responsible for generating control signals and managing counters (`R_i` and `R_r`) for a process that includes data reading phases, two types of operations (`OP1`, `OP2`), and start/end-of-file logic.

Inputs

- **`clk`**: Main clock signal, used to synchronize all registers and state advancement.
- **`rst_n`**: Asynchronous reset signal, active low. When at 0, it forces the FSM to the initial state `S_start` and resets the counters.
- **`F_dr`**: "Data Ready" Flag. Indicated by an external module to signal that new data is available for reading.
- **`End_of_File`**: "End of File" Flag. Indicated by an external module to signal that there is no more data to process. Used to switch to Operation 2, it functions similarly to `F_dr`, waiting for the process to finish (i.e., for `F_rtr` to return to 1) and then signaling that the previous byte was the last one.
- **`start`**: General start signal. Used in conjunction with `case_rc0` for a conditioned start, to prevent further unwanted starts during the process.
- **`case_rc0`**: Condition useful for preventing multiple uses of the start signal.

Outputs

- **`F_rtr`**: "Ready To Read" Flag. Indicates to the external module that the system is ready to receive new data.
- **`switch_operation`**: "Switch Operation" Flag. Communicates to the operational module that the system is switching from "Operation 1" logic to "Operation 2".
- **`H_ready`**: "H_Ready" Flag. Indicates that the entire process has been completed and the module is in the final state `S_END`.
- **`validate_input`**: "Input Validated" Flag. Signals that the input data is valid, based on the condition that `F_dr` is 1 and `F_rtr` is 1.
- **`validate_R_H`**: "Read/Handshake Validation" Flag. An internal validation signal used in specific phases of the operations.
- **`R_i`**: 3-bit counter (from 0 to 7). Manages internal iterations within a "round", and is used by the operational module to perform operations.

Internal Circuitry

- **R_r**: 4-bit counter (from 0 to 15, but the maximum relevant value is 11). Manages the total "rounds" of the process. It is initialized to 15 to synchronize it with the R_i register and to have a more readable count of the rounds; since its increment occurs in the S_OP1_RR state, the first iteration, if initialized to 0, would directly lead to count 1, an event that would not allow correct operation.
- **last_iteration**: Active signal when R_i = 6, useful for signaling the start of the last cycle of a round, and to synchronize the VLI signal.
- **validate_input**: Active signal when F_dr and F_rtr are 1, useful for signaling to registers R_b and R_c that they can update their value.
- **incrementation_round**: Active signal when R_i = 7, useful for signaling the end of the round.

State Advancement Logic and Description of Each State

The module moves between eight distinct states, managed by a sequential block (`always_ff`) and a combinatorial block (`always_comb`). Asynchronous reset (`rst_n`) or a synchronous start (`real_start`) force the FSM into the S_start state.

S_start (3'd0) - Initial State

- **Activated by**: Reset (`rst_n=0`), start (`real_start=1`).
- **Transition conditions**:
 - If F_dr is 1 (data ready) **AND** F_rtr_s is 1 (ready to read), transition to S_read.
 - If F_rtr_s is 1 (ready to read) **AND** End_of_File is 1 (end of file) (empty message case), transition to S_OP2_AZZERA thus starting with Operation 2.
 - Otherwise, remain in S_start.
- **Actions in this state**:
 - R_i and R_r are reset (`R_i <= 0`, `R_r <= 15` (in this case, it starts from 15 for synchronization purposes, as the increment occurs during the S_OP1_RR state, and consequently would happen one more time)).
 - SW_O (switch_operation) is 0, signaling to the operational part which combinatorial module to use.
 - F_rtr_s (F_rtr) is 1, signaling to the external module that a byte can be transmitted.
 - VLI (validate_R_H) is 0, signaling that the produced values should not be stored.
 - H_R (H_ready) is 0.

S_read (3'd1) - Data Read State

- **Activated by:** S_start when F_dr and F_rtr_s are 1, or from S_WAIT1 when F_dr is 1, as a new value has been sent.
- **Transition conditions:**
 - Always transitions to S_OP1_RR. It is a transitional state useful for module synchronization.
- **Actions in this state:**
 - F_rtr_s is 0, to signal that the new byte has been read and therefore the process is running.

S_OP1_RR (3'd2) - Operation 1 State: Round Increment

- **Activated by:** S_read if the byte is new, or from S_RI for the ongoing process, when incrementation_round is 1 and ric_12 is 0; in the first case it is a forced transition, in the second case it means that the iterations for that round are finished, but the first operation is not yet completed.
- **Transition conditions:**
 - If ric_12 is 0 (R_r has not reached 11) **AND** incrementation_round is 0 (R_i has not reached 7), transition to S_RI.
 - Otherwise, remain in S_OP1_RR.
- **Actions in this state:**
 - R_i is reset ($R_i \leq 0$).
 - R_r is incremented ($R_r \leq R_r + 1$).
 - VLI (validate_R_H) is 1, which indicates to the operational module which values are valid.

S_RI (3'd3) - State: Iteration Increment

- **Activated by:** S_OP1_RR.
- **Transition conditions:**
 - If incrementation_round is 1 (R_i has reached 7) **AND** ric_12 is 0 (R_r has not reached 11), transition to S_OP1_RR.
 - If ric_12 is 1 (R_r has reached 11) **AND** last_iteration is 1 (R_i has reached 6) indicates that byte processing is complete, and transitions to S_WAIT1.
 - Otherwise, remain in S_RI.
- **Actions in this state:**
 - R_i is incremented ($R_i \leq R_i + 1$).

S_WAIT1 (3'd4) - Wait State

- **Activated by:** S_RI when all Operation 1 rounds are completed, it waits for the external module to be ready to communicate. **Transition conditions:**
 - If F_dr is 1 (data ready), transition to S_read.
 - If End_of_File is 1 (end of file), transition to S_OP2_AZZERA.
 - Otherwise, remain in S_WAIT1.
- **Actions in this state:**
 - R_i is reset ($R_i \leq 0$).
 - R_r is initialized to 15 ($R_r \leq 15$).
 - F_rtr_s (F_rtr) is 1.
 - VLI (validate_R_H) is 0.

S_OP2_AZZERA (3'd5) - Operation 2 State: Reset

- **Activated by:** S_start or S_WAIT1 when End_of_File is 1. This is a transitional state useful for synchronizing registers and allowing correct advancement to Operation 2.
- **Transition conditions:**
 - Always transitions to S_OP2_RI.
- **Actions in this state:**
 - R_i is reset ($R_i \leq 0$).
 - SW_O (switch_operation) is 1, signaling to the operational module to switch to the second operation.
 - F_rtr_s (F_rtr) is 0.
 - VLI (validate_R_H) is 1.

S_OP2_RI (3'd6) - Operation 2 State: Iteration Increment

- **Activated by:** Direct transition from S_OP2_AZZERA. This is the main state for the second operation.
- **Transition conditions:**
 - If last_iteration is 1 (R_i has reached 6), transition to S_END.
 - Otherwise, remain in S_OP2_RI.
- **Actions in this state:**
 - R_i is incremented ($R_i \leq R_i + 1$).

S_END (3'd7) - Final Completion State

- **Activated by:** S_OP2_RI when Operation 2 is completed.
- **Transition conditions:**
 - Remains in S_END.
- **Actions in this state:**
 - VLI (validate_R_H) is 0.
 - H_R (H_ready) is 1.

4.3 Module `operative_part`

The `operative_part` module is responsible for executing the main hashing operations. Based on input data and control signals from the `control_part` module, it implements the combinatorial and sequential logic necessary for processing input bytes, managing internal counters, and calculating intermediate hash digests through several specialized submodules. It serves as the computational "engine" of the hashing system.

Inputs

The `operative_part` module receives the following input signals:

- **B**: Represents the current data byte that needs to be processed by the module.
- **start**: A general start signal, used in combination with internal logic to ensure controlled system initialization.
- **clock**: The main clock signal, used to synchronize all sequential logic (registers) within this module.
- **rstn**: Asynchronous reset signal, active low. When 0, it forces the internal registers of the module to their initial reset values.
- **validate_input**: This is a flag from the `control_part` module. When 1, it indicates that a new valid byte is present on input B and must be loaded for processing.
- **switch_operation**: A flag from the `control_part` module. It signals whether the system is currently performing "Operation 1" or "Operation 2", influencing the S-box input selection.
- **validate_R_h**: A flag from the `control_part` module. When 1, it signals that the calculated intermediate hash value (`xor_shift_out`) is ready and must be loaded into the main hash register `R_h`.
- **R_i**: Represents the current iteration counter from the `control_part` module. It is used to control specific operations within submodules (e.g., C6 and `xor_shift`).

Outputs

The `operative_part` module generates the following output signals:

- **R_h**: A 32-bit output register. This register contains the current hash digest, whether intermediate or final, which is calculated. Its value is updated sequentially.
- **case_R_c_zero**: Output wire. This signal is 1 when the internal counter `R_c` is 0, indicating an initial state or a reset condition for the round counter.

Internal Signals and Registers

The module uses the following internal registers and wires:

- **R_b**: 8-bit register. Stores the last valid input byte B that has been loaded for processing.
- **R_c**: 64-bit register. A counter, typically used to track the number of processed bytes. It is initialized to 0. It is used during the final phase of the process to perform necessary operations.
- **m6_out**: Output of the M6 submodule.
- **m6_in**: Input of the M6 submodule, directly connected to the R_b register.
- **c6_out**: Output of the C6 submodule.
- **c6_in**: Input of the C6 submodule, directly connected to the R_c register.
- **sbox_in**: Input of the S-box (sbox submodule). Its value depends on the state of switch_operation.
- **sbox_out**: Output of the S-box (sbox submodule).
- **xor_shift_in**: Input of the xor_shift submodule, directly connected to the R_h register.
- **xor_shift_out**: Output of the xor_shift submodule, representing the modified hash digest after applying XOR and shift operations.
- **real_start**: A derived start signal, active when both case_R_c_zero is 1 and the external start signal is 1. It ensures a synchronized and conditioned start.

Combinational Logic (assign Statements)

The following assign statements describe the combinational logic of the module:

- **assign case_R_c_zero = (R_c == 64'd0)?1:0;** This statement continuously checks if the R_c counter is zero and sets the case_R_c_zero flag accordingly. This flag is crucial for the initial setup and conditioned start logic.
- **assign m6_in = R_b;** The current processed byte, stored in R_b, is passed as input to the M6 submodule.
- **assign c6_in = R_c;** The current round/byte counter, R_c, is passed as input to the C6 submodule.
- **assign real_start = case_R_c_zero & start;** This ensures that the internal reset/initialization, triggered by the start signal, occurs only when R_c is zero, preventing unwanted re-initialization mid-process.
- **assign sbox_in = switch_operation == 1? c6_out : m6_out;** This statement implements a multiplexer for the S-box input. If switch_operation is 1 (indicating Operation 2), the output of C6 (c6_out) is used. Otherwise (Operation 1), the output of M6 (m6_out) is used.

- **assign xor_shift_in = R_h;** The current value of the hash register R_h is passed as input to the xor_shift submodule to be modified.

Instantiation of Submodules

The operative_part module instantiates the following functional blocks:

- **M6** (modulo_m6):
 - **Purpose:** This module performs an initial transformation and permutation on the input byte R_b, following the specifications provided in the trace.
 - **Connections:** It receives an 8-bit input (m6_in) and produces a 6-bit output (m6_out).
- **C6** (modulo_c6):
 - **Purpose:** This module performs a transformation and permutation based on the i-th byte (selected by R_i) of the round counter R_c.
 - **Connections:** It receives a 64-bit input C6_in (R_c) and a 3-bit iteration index i (R_i), producing a 6-bit output C6_out.
- **sbox** (modulo_sbox):
 - **Purpose:** A Substitution Box (S-box) is a fundamental component in many cryptographic algorithms. It performs a non-linear substitution, providing the property of "confusion" in the cipher.
 - **Connections:** It receives a 6-bit input in (sbox_in) and produces a 4-bit output out (sbox_out).
- **xor_shift** (modulo_xor_shift):
 - **Purpose:** This module performs a combination of XOR (exclusive OR) and bit shift operations on the current hash digest, incorporating the S-box output and the iteration counter used to modify the nibbles of R_h. This provides the property of "diffusion" in the cipher.
 - **Connections:** It receives the current hash H (xor_shift_in), the S-box output S (sbox_out), and the iteration index I (R_i), producing a modified hash H_modified (xor_shift_out).

Sequential Logic (always_ff Block)

This block describes how the internal registers R_c, R_h, and R_b are updated on the rising edge of the clock or the falling edge of rstn.

- **Reset Condition (!rstn):**
 - If rstn is 0 (active low reset), all registers are asynchronously reset to their initial values: R_c to 64'd0, R_h to 32'h32FE1AF3 (the latter was initialized considering the definition structure used in the high-level model), and R_b to 8'd0 (NULL value).
- **Initial Start Condition (real_start == 1'b1):**

- If `real_start` is 1 (indicating a synchronized start when `R_c` is zero), the registers are explicitly initialized to the same values as the reset.

- **Normal Operation:**

- If `validate_input` is 1, it means a new valid byte has arrived. In this case, `R_c` is incremented by 1, and the input byte `B` is loaded into the `R_b` register.
- If `validate_R_h` is 1, it means that the newly inserted byte and subsequent processed values must be able to modify the main hash register `R_h`, otherwise the calculated values must be ignored.

In summary, the `operative_part` module acts as the computational heart of the hashing system, processing input bytes through a series of transformations and accumulating the results in the `R_h` register, all under the precise control of the `control_part` module.

4.4 Future upgrades

In future developments, the current design could be enhanced by integrating the state information into the operational part of the system. This modification will require corresponding changes to the Finite State Machine (FSM). Furthermore, one of the primary objectives of this upgrade is to optimize performance, with the intention of achieving a potential increase in system speed.

CHAPTER 5

Functional Verification

5.1 testbench.sv

The testbench is designed to simulate **five primary operational scenarios**, which include normal data flow, reset handling, unexpected control signal management, simulation of transmission delays, and processing of a null input. **Furthermore, the testbench evaluates the module's robustness and functional consistency through tests on pairs of strings, distinguishing between:**

- pairs with identical strings;
- pairs with similar strings but differing in length (by a single character);
- pairs with strings of the same length but with different content.

Simulation Parameters

- `\timescale 1ns/1ps`: Defines the simulation time unit as 1 nanosecond (ns) and the simulation precision as 1 picosecond (ps).
- `parameter CLK_PERIOD = 20;`: Sets the clock signal period to 20 ns. This corresponds to a clock frequency of 50 MHz (1 / 20 ns).

General Structure

Internal and Support Signals This section declares the signals required for interaction between the testbench and the DUT.

- **Input Signals to the DUT (declared as `reg`):**

- `reg clk = 1'b0;`: The clock signal, initialized to low. It is fundamental for synchronizing sequential logic within the DUT.
- `reg rst_n;`: Asynchronous reset signal, active low. Used to return the DUT to an initial state.
- `reg start;`: Start signal to initiate the hashing operation.
- `reg [7:0] Byte;`: 8-bit bus carrying the current data byte to be processed.
- `reg End_of_File;`: Signal indicating the end of the data stream to be hashed.
- `reg F_dr;`: "Data Ready" flag from the testbench to the DUT, indicating that a new byte is available on `Byte`.

- **Output Signals from the DUT (declared as `wire`):**

- `wire [0:31] R_h;`: 32-bit bus representing the final or intermediate hash digest calculated by the DUT.
- `wire F_rtr;`: "Ready To Receive" flag from the DUT to the testbench, indicating that the DUT is ready to accept a new input byte.
- `wire H_ready;`: "Hash Ready" flag from the DUT to the testbench, indicating that the final hash calculation is complete and the value on `R_h` is valid.

- **Internal Testbench Registers and Variables:**

- `reg [31:0] test_output;`: A working register for potential output checks, although not explicitly used to store the final hash in this testbench.
- `integer i;`: Integer variable used as a counter in `for` loops for sending bytes.
- `integer reset_done = 0;`: Flag used to control the application of reset in specific tests, to prevent multiple process resets.
- `string test = "CiaoMondo";`: The predefined input string that will be hashed in the tests.
- `string test2 = "CiaoMondo";`: The second input string that will be hashed in the tests.
- `string test3 = "CiaoMondo!";`: The input string that will be hashed in the tests and compared with the first.
- `string test3 = "CiaoCiao!";`: The last input string that will be hashed in the tests and compared with the first.
- `integer test_reset = $urandom_range(0,test.len());`: Generates a randomly chosen position within the `test` string where a reset will be applied.
- `integer test_start = $urandom_range(0,test.len());`: Generates a random position where a `start` signal will be attempted during the process.

- `integer ritardo = $urandom_range(0,test.len());` :: Generates a random position where a delay will be introduced in data transmission.

DUT Instantiation The `full_hash` dut block creates an instance of the `full_hash` module (the Design Under Test) and connects its ports to the corresponding signals declared in the testbench. This establishes the link between the stimuli generated by the testbench and the DUT's reactions.

Clock Generator The `always #((CLK_PERIOD)/2) clk = ~clk;` block generates a continuous clock signal with a period defined by `CLK_PERIOD`. Every half period (`CLK_PERIOD/2`), the value of `clk` is inverted, creating a square wave.

Main Stimulus Block (`initial`)

This `initial` block defines the sequence of stimuli applied to the DUT and controls the different test scenarios.

Global Initialization

At the start of the simulation, all control and data signals are initialized to a known state:

- `clk = 1'b0;`
- `rst_n = 1'b1;` (reset released)
- `start = 1'b0;`
- `F_dr = 1'b0;`
- `Byte = 8'h00;`
- `End_of_File = 1'b0;`

Test 1: Regular Execution

Purpose: Verify the basic operation of the hashing module in an ideal environment, without interruptions or errors.

Description:

1. The testbench asserts the `start` signal for one clock cycle to initiate the hashing operation.
2. Subsequently, a `for` loop iterates over each character of the `test` string ("CiaoMondo").
3. For each byte:
 - `F_dr` is set to 1 to signal that a new byte is ready.
 - The testbench waits for `F_rtr` from the DUT to be 1, indicating that the DUT is ready to receive the byte.
 - The current byte `test[i]` is placed on the `Byte` bus.

- The testbench waits for the rising edge of the clock (`@ (posedge clk)`) to allow synchronization.
 - `F_dr` is brought back to 0.
 - The testbench waits for `F_rtr` from the DUT to be 0, confirming that the DUT has acquired the byte and is processing it.
4. After all bytes have been sent, the testbench waits for `F_rtr` to be 1 (the DUT has finished processing the last byte and is ready for any further input), then sets `End_of_File` to 1 for one clock cycle, signaling the end of the data stream.
 5. The testbench waits for the `H_ready` signal from the DUT, which indicates that the final hash has been calculated.
 6. The final hash value (`R_h`) is displayed.
 7. A small delay (`# (CLK_PERIOD * 2)`) is introduced to stabilize signals before the next test.

Test 2: Execution with Random Reset

Purpose: Verify the module's ability to handle an asynchronous reset during operation and correctly resume hashing from the beginning.

Description:

1. The DUT is initially reset (`rst_n = 1'b0`) for 5 clock cycles and then released (`rst_n = 1'b1`) (optional, but necessary in this case and in subsequent tests, as the start signal would be blocked from the previous test).
2. The `start` signal is asserted. (real start of the second simulation)
3. A `for` loop similar to Test 1 begins sending bytes.
4. When the counter `i` reaches the `test_reset` position and the reset has not yet been applied (`reset_done == 0`):
 - `rst_n` is driven to 0 (active reset).
 - `F_dr` and `Byte` are reset. (necessary to also reset the DUT module)
 - The counter `i` is set to -1 (upon the next increment, it will restart from 0 to resend the test string).
 - `reset_done` is set to 1 to prevent further resets.
 - A delay of 5 clock cycles is applied with the reset active, to ensure signal stabilization.
 - `rst_n` is released.
 - The testbench waits for a rising clock edge to synchronize (otherwise `F_dr` would toggle too quickly and there would be synchronization issues).
5. The data flow resumes from the beginning (thanks to `i = -1` and the subsequent `i = i + 1`).
6. At the end of the data flow, `End_of_File` is signaled, and `H_ready` is awaited.
7. The final hash is displayed.

Test 3: Blocking Start Signal during Execution

Purpose: Verify that a `start` signal asserted while the module is already processing is ignored or handled without compromising the current operation.

Description:

1. The DUT is reset and then released, and the `start` signal is asserted to begin the operation.
2. A `for` loop sends the bytes of the `test` string.
3. When the counter `i` reaches the `test_start` position:
 - The `start` signal is momentarily asserted for one clock cycle. This simulates an external attempt to restart hashing while it is already in progress.
4. The data flow continues normally.
5. At the end of the flow, `End_of_File` is signaled, `H_ready` is awaited, and the final hash is displayed.

Test 4: Execution with Data Transmission Delay

Purpose: Verify the module's robustness and its ability to wait for a new input byte for a prolonged period, without stalling or producing incorrect results.

Description:

1. The DUT is reset and then released, and the `start` signal is asserted.
2. A `for` loop sends the bytes of the `test` string.
3. When the counter `i` reaches the `ritardo` position:
 - A significant delay ($\#(\text{CLK_PERIOD} * 500)$) is introduced in the transmission of the next byte. This simulates a pause in the data stream input.
 - The testbench waits for a rising clock edge to synchronize after the delay. (necessary to wait for the next rising edge in case of delay)
4. The data flow resumes normally with the byte transmission.
5. At the end of the flow, `End_of_File` is signaled, `H_ready` is awaited, and the final hash is displayed.

Test 5: Empty Value Input

Purpose: Verify the module's behavior when provided with an empty input string. It should calculate a hash for an empty string (often a predefined or algorithm-specified value).

Description:

1. The `test` variable is explicitly set to an empty string (`test = ""`);).
2. The DUT is reset and then released.
3. The `start` signal is asserted.

4. The `for` loop for sending bytes will not execute, as `test.len()` will be 0.
5. The testbench immediately waits for `F_rtr` to be 1 (if the DUT is configured to signal `Ready To Receive` even with empty input), then sets `End_of_File` to 1 for one clock cycle.
6. `H_ready` is awaited, and the final hash is displayed. This value should be the hash of the empty string.

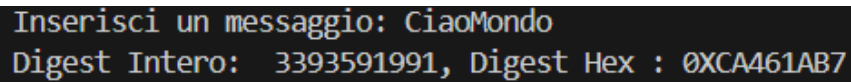
Additional Tests: Regular Execution with Different Strings

Purpose: To verify the consistent operation of the module using 3 pairs of strings, including 2 identical, 2 similar but differing in length, and 2 with the same length but different content. **Description:**

1. For each string, a reset of the values is performed first.
2. Subsequently, the DUT is invoked twice for each test.
3. A normal execution then follows, as previously described.
4. The final hash value (`R_h`) is displayed for each string. It is noted that for the first pair, the result is the same for each string, while for the other 2 pairs, the expected differences are evident.

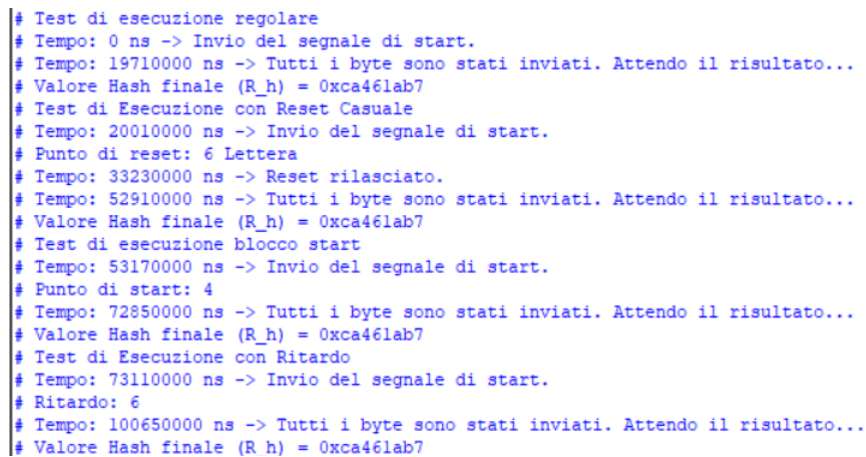
Results and Waveforms

Results



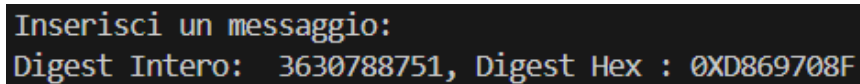
```
Inserisci un messaggio: CiaoMondo
Digest Intero: 3393591991, Digest Hex : 0XCA461AB7
```

Figure 5.1: *High-level model test result*



```
# Test di esecuzione regolare
# Tempo: 0 ns -> Invio del segnale di start.
# Tempo: 19710000 ns -> Tutti i byte sono stati inviati. Attendo il risultato...
# Valore Hash finale (R_h) = 0xca461ab7
# Test di Esecuzione con Reset Casuale
# Tempo: 20010000 ns -> Invio del segnale di start.
# Punto di reset: 6 Lettera
# Tempo: 33230000 ns -> Reset rilasciato.
# Tempo: 52910000 ns -> Tutti i byte sono stati inviati. Attendo il risultato...
# Valore Hash finale (R_h) = 0xca461ab7
# Test di esecuzione blocco start
# Tempo: 53170000 ns -> Invio del segnale di start.
# Punto di start: 4
# Tempo: 72850000 ns -> Tutti i byte sono stati inviati. Attendo il risultato...
# Valore Hash finale (R_h) = 0xca461ab7
# Test di Esecuzione con Ritardo
# Tempo: 73110000 ns -> Invio del segnale di start.
# Ritardo: 6
# Tempo: 100650000 ns -> Tutti i byte sono stati inviati. Attendo il risultato...
# Valore Hash finale (R_h) = 0xca461ab7
```

Figure 5.2: *DUT test results*



```
Inserisci un messaggio:
Digest Intero: 3630788751, Digest Hex : 0XD869708F
```

Figure 5.3: *High-level model test result*


```
# Test valore vuoto
# Tempo: 100910000 ns -> Invio del segnale di start.
# Tempo: 100950000 ns -> Tutti i byte sono stati inviati. Attendo il risultato...
# Valore Hash finale (R_h) = 0xd869708f
```

Figure 5.4: *DUT test result*

```
# Test stringhe uguali
# Valore prima stringa: CiaoMondo
# Valore Hash finale (R_h) = 0xca461ab7
# Valore seconda stringa: CiaoMondo
# Valore Hash finale (R_h) = 0xca461ab7
# Test stringhe simili
# Valore prima stringa: CiaoMondo
# Valore Hash finale (R_h) = 0xca461ab7
# Valore seconda stringa: CiaoMondo!
# Valore Hash finale (R_h) = 0x5c7cd9eb
# Test stringhe simili
# Test di Esecuzione con Reset Casuale
# Valore prima stringa: CiaoMondo
# Valore Hash finale (R_h) = 0xca461ab7
# Test di Esecuzione con Reset Casuale
# Valore seconda stringa: CiaoCiao!
# Valore Hash finale (R_h) = 0x973b920d
```

Figure 5.5: *DUT test result*

Waveforms

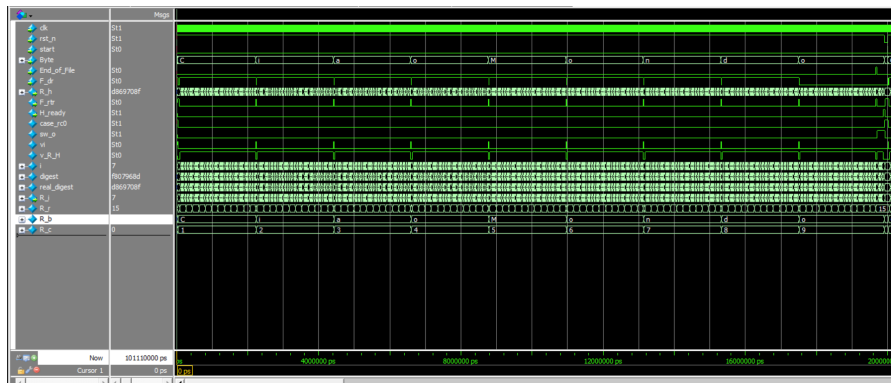


Figure 5.6: Test 1

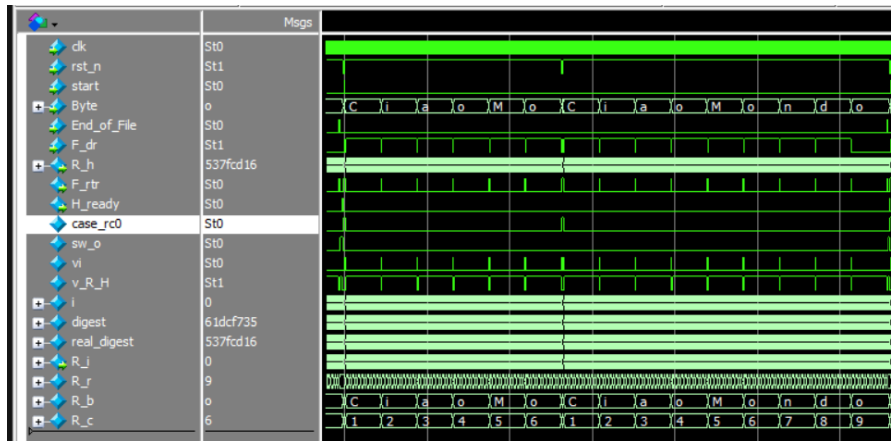


Figure 5.7: Test 2

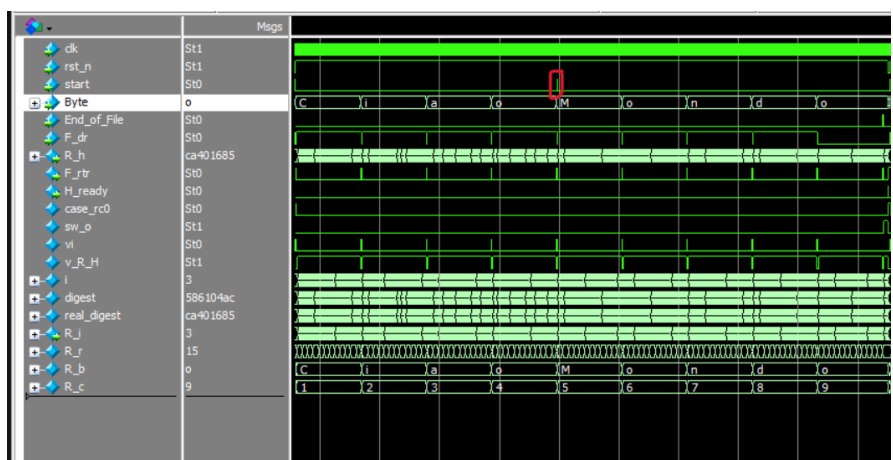
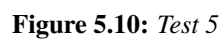
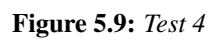


Figure 5.8: Test 3



CHAPTER 6

FPGA Implementation Results

6.1 Synthesis and Timing Analysis Phase

After verifying the module's correct functionality through testbenches, the synthesis phase commenced using the Intel Quartus Prime software. The selected target device for implementation was the 5CGXFC9D6F27C7, a member of the Cyclone V family. The synthesis process was structured into three main phases: Analysis & Synthesis, Fitter, and Static Timing Analysis (STA), aimed at determining the maximum operating frequency of the design.

Analysis & Synthesis

This initial phase began with the assignment of Virtual Pins, a practice useful for simulating the module as a component of a larger system rather than as a top-level entity, thereby facilitating integration and verification. Subsequently, the logical representation of the circuit was generated from the HDL code. The analysis did not report any significant warnings, indicating a correct translation of the HDL code into a logical netlist.

Compilation Report - full_hash_v3 X Assignment Editor X									
<<new>>		Filter on node names: *						Category: All	
tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag	
1	✓	in F_dr	Virtual Pin	On	Yes	full_hash			
2	✓	out F_rtr	Virtual Pin	On	Yes	full_hash			
3	✓	out H_ready	Virtual Pin	On	Yes	full_hash			
4	✓	case_rc0	Virtual Pin	On	Yes	full_hash			
5	✓	real_digest	Virtual Pin	On	Yes	full_hash			
6	✓	in rst_n	Virtual Pin	On	Yes	full_hash			
7	✓	start	Virtual Pin	On	Yes	full_hash			
8	✓	sw_o	Virtual Pin	On	Yes	full_hash			
9	✓	v_R_H	Virtual Pin	On	Yes	full_hash			
10	✓	vi	Virtual Pin	On	Yes	full_hash			
11	✓	Byte	Virtual Pin	On	Yes	full_hash			
12	✓	out R_h	Virtual Pin	On	Yes	full_hash			
13	✓	digest	Virtual Pin	On	Yes	full_hash			
14	✓	i	Virtual Pin	On	Yes	full_hash			
15	✓	contr...ntrol	Virtual Pin	On	Yes	full_hash			
16	✓	invert...tore	Virtual Pin	On	Yes	full_hash			
17	✓	operat...rative	Virtual Pin	On	Yes	full_hash			
18	✓	End_of_File	Virtual Pin	On	Yes	full_hash			
19	<<new>>	<<new>>	<<new>>						

Figure 6.1: *Assignements Virtual PIN*

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed Sep 10 19:38:49 2025
Quartus Prime Version	23.1std.1 Build 993 05/14/2024 SC Lite Edition
Revision Name	full_hash_v3
Top-level Entity Name	full_hash
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	123
Total pins	1
Total virtual pins	46
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Figure 6.2: *Analysis report*

Fitter

The Fitter phase was dedicated to mapping the logical netlist onto the physical resources of the FPGA device. This includes the allocation of logic blocks, registers, and routing paths. The report generated from this phase provided the count of utilized ALMs (Adaptive Logic Modules), a crucial metric for evaluating hardware implementation efficiency and for future resource optimizations.

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Wed Sep 10 21:46:51 2025
Quartus Prime Version	23.1std.1 Build 993 05/14/2024 SC Lite Edition
Revision Name	full_hash_v3
Top-level Entity Name	full_hash
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	145 / 113,560 (< 1 %)
Total registers	167
Total pins	1 / 378 (< 1 %)
Total virtual pins	46
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 Report

Static Timing Analysis (STA)

Upon completion of the physical implementation phases (synthesis and fitting), Static Timing Analysis was performed to determine the maximum sustainable clock frequency for the device. This analysis was conducted by defining appropriate timing constraints using an SDC (Synopsys Design Constraints) file. Specifically, the setup and hold times were set to 10The analysis process started with an initial clock period of 5ns (corresponding to 200MHz) and was progressively increased (or frequency decreased) until the analyzer no longer reported timing violations. Considering the worst-case scenario (Slow 1100mv 0C Model), a maximum operating frequency of 136.54 MHz was determined, corresponding to a clock period of approximately 7.35ns.

Condition	Slack Setup	Slack Hold
Slow 1100mV 85C Model	-2.718	0.346
Slow 1100mV 0C Model	-2.676	0.120
Fast 1100mV 85C Model	0.900	0.175
Fast 1100mV 0C Model	1.260	0.159

Table 6.1: 5ns results

Condition	Slack Setup	Slack Hold
Slow 1100mV 85C Model	0.054	0.317
Slow 1100mV 0C Model	0.006	0.300
Fast 1100mV 85C Model	2.819	0.166
Fast 1100mV 0C Model	3.105	3.058

Table 6.2: 7.35ns results