UNIVERSITY OF PISA

MASTER'S DEGREE IN CYBERSECURITY

LIGHT HASH ALGORITHM V4

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Giacomo Maldarella
Andrea Fabbri

Academic year 2023/2024

Contents

CHAPTER 1

Project Specifications

1.1 Introduction

The goal of this project is to design and implement a lightweight hash module, referred to as the Light Hash algorithm v4, which generates a 32-bit digest for each 32-bit input data block. This chapter provides an overview of the project specifications and an analysis of the hash function's operations.

1.2 Specifications

The core functionality of the hash module revolves around the following operations:

```
for (r = 0; r < 24; r++) {
    if (r == 0)
        H = SA(m);
    else
        H = H ⊕ IV;

    H = Θ(H);
    H = ρ(H);
}
d = FPX(H);
```

Where:

- r is the number of rounds (24 rounds in total).
- m is the 32-bit input message, divided into four 8-bit blocks.

- H is the state array.
- IV is the Initialization Vector, with a fixed value of $[0x34, 0x55, 0x0F, 0x14]$.
- d is the final 32-bit digest of the message.

The hash function consists of the following components:

1.2.1 State Array (SA) Function

The SA function takes the input message m and XORs it with the Initialization Vector IV to initialize the state array H. The operation is as follows:

```
for (i = 0; i < 4; i++)
    H[i] = m[i] xor IV[i];
```

1.2.2 Permutation Function (Theta, Θ)

The Theta function permutes the state array by reversing its elements:

```
for (i = 0; i < 4; i++)
    H[i] = H[3 - i];
```

1.2.3 Modification Function (ρ)

The ρ function modifies each element of the state array by adding a constant value (0x85) and then performing modulo 0xFD operations:

```
for (i = 0; i < 4; i++)
    H[i] = (H[i] + 0x85) mod 0xFD;
```

1.2.4 Final Permutation and Xoring Function (FPX)

The FPX function finalizes the digest by permuting the state array once again and XORing it with the Initialization Vector:

```
for (i = 0; i < 4; i++)
    d[i] = H[3 - i] xor IV[i];
```

1.3 Conclusion

The Light Hash algorithm v4 implements a series of operations to process a 32-bit input block into a 32-bit digest using 24 rounds of transformations. The main steps include initializing the state array, applying permutation and modification functions, and finally generating the digest through the final permutation and XOR operations.

It is important to note that while the algorithm consists of 24 rounds, the first round differs slightly from the subsequent rounds. In the first round, the State Array (SA) function is applied. From the second round onward, the state array is simply XORed with the IV, bypassing the SA function. The rest of the operations—Theta permutation and rho modification—are applied consistently across all 24 rounds, ensuring that the input is thoroughly transformed before producing the final 32-bit digest.

CHAPTER 2

High-level Model

2.1 Introduction

This chapter describes the high-level model of the Light Hash algorithm v4, which was implemented in Python. The model implements the algorithm as described in the previous chapter, ensuring that the hash module operates correctly before transitioning to a lower-level hardware description language (HDL) for further implementation.

2.2 Python Implementation

The code is divided into modular functions, each corresponding to a specific operation in the hashing process, such as the state array initialization, permutation, modification, and final XORing.

2.2.1 Main Function

The ‘hash_function()‘ function coordinates the entire process by calling the other functions sequentially to generate the 32-bit digest. The number of rounds is set to 24 by default.

```
def hash_function(m, IV, rounds=24):
    for r in range(rounds):
        if r == 0:
            H = SA(m, IV)
        else:
            H = [H[i] ^ IV[i] for i in range(4)]
        H = Theta(H)
        H = rho(H)
```

```
    d = FPX(H, IV)
    return d
```

2.2.2 State Array Initialization (SA)

The state array initialization function, 'SA(m, IV)', takes the 32-bit input message m and XORs it with the 32-bit Initialization Vector IV, creating an initial state array H.

```
def SA(m, IV):
    H = [0] * 4
    for i in range(4):
        H[i] = m[i] ^ IV[i]
    return H
```

2.2.3 Permutation Function (Theta)

The permutation function, 'Theta(H)', reverses the elements of the state array, effectively scrambling the internal state of the hash function.

```
def Theta(H):
    H_theta = [0] * 4
    for i in range(4):
        H_theta[i] = H[3 - i]
    return H_theta
```

2.2.4 Modification Function (rho)

The 'rho(H)' function adds a constant to each element of the state array and then performs a modulo operation to further modify the state.

```
def rho(H):
    for i in range(4):
        temp = (H[i] + 0x85) & 0xFF
        H[i] = temp % 0xFD
    return H
```

As can be seen, an additional bitwise `& 0xFF` operation has been added in the Python implementation. This was necessary because of an issue encountered during the Verilog implementation.

System Verilog Behaves

Verilog, as a hardware description language, is designed to describe circuits and hardware that inherently operate on fixed-width registers (such as 8-bit, 16-bit, or 32-bit registers). In these systems, overflow is a common behavior, as registers cannot store more than their fixed number of bits. For an 8-bit register, the range is 0-255, and any addition that exceeds this range wraps around. This is analogous to how digital counters work, where counting beyond the maximum value causes the counter to reset.

Python Behaves

Python, on the other hand, is a high-level language designed with arbitrary-precision integers. This means that integers in Python are not constrained by a fixed bit-width. You can keep adding to a number, and Python will dynamically allocate more memory to handle the larger values. Thus, without intervention, Python will not overflow at 8 bits, 16 bits, or any other boundary—unless explicitly programmed to do so.

The Solution

To replicate the Verilog behavior in Python, we applied the `& 0xFF` operation. This operation ensures that only the lower 8 bits of the result are preserved, thus truncating the result to fit within the 8-bit boundary, just as Verilog would. This modification guarantees that the Python model behaves consistently with the Verilog implementation, making the high-level model a faithful representation of the hardware behavior.

2.2.5 Final Permutation and XORing Function (FPX)

The FPX function permutes the state array one final time and XORs it with the Initialization Vector IV to generate the final digest.

```
def FPX(H, IV):
    d = [0] * 4
    for i in range(4):
        d[i] = H[3 - i] ^ IV[i]
    return d
```

2.2.6 Input Handling

The ‘read_message()‘ function is responsible for handling user input. It ensures that the user provides a valid 4-byte message in hexadecimal format, which is then used as the input to the hash function.

```
def read_message():
    while True:
        try:
            message = input("Insert a 4 byte message: ")
            m = [int(x, 16) for x in message.split()]
            if len(m) != 4: raise ValueError
            return m
        except ValueError:
            print("Input not valid.")
```

This input handling was designed to mimic how input would be provided in an HDL environment using test vectors. By ensuring that the input is formatted as 4 bytes in hexadecimal, we can easily replicate the same behavior when testing the HDL implementation of the hash function.

2.3 Conclusion

The high-level Python implementation of the Light Hash algorithm v4 accurately reflects the intended design and behavior of the hash function. It serves as a useful prototype for verifying the correctness of the algorithm before transitioning to hardware design in SystemVerilog.

CHAPTER 3

RTL Design

In this chapter, we describe the RTL design architecture for the Light Hash algorithm v4. The design has been divided into multiple modules to allow for clarity, reusability, and ease of testing. We have structured the design with a top-level module that coordinates the operation of the hash algorithm and various submodules that implement the core logic such as rounds and specific transformations.

3.1 Design Overview

The top-level module is responsible for arranging the overall hash computation by managing the input and output, controlling the rounds of transformation, and ensuring proper synchronization between the submodules. The modules are:

- **Hash_light_top.sv:** The top-level module, which manages the flow of data and controls the rounds.
- **Hash_round.sv:** This module handles the transformations for each round of the hash algorithm.
- **SA.sv:** This module implements the State Array (SA) function, which processes the input message.
- **Theta.sv:** This module applies the permutation function on the state array.
- **Rho.sv:** This module applies a specific transformation to the state elements.
- **FPX.sv:** This module implements the final permutation and XORing function.
- **Xor_module.sv:** This module handles the XOR operations used across the design.

3.2 Top-Level Module: Hash_light_top.sv

The top-level module, *Hash_light_top.sv*, controls the flow of data through the rounds, manages input/output signals, and interfaces with the submodules. The main logic is sequential, driven by a clock signal, and uses control logic to manage round operations.

Key Responsibilities:

- Managing input and output interfaces.
- Arranging multiple rounds by controlling the *Hash_round* module.
- Resetting and synchronizing the internal state.

The top-level module implementation is shown below:

```
module Hash_light_top (
    input  logic clk,
    input  logic rst_n,
    input  logic start,
    input  logic [7:0] m [0:3],
    output logic [7:0] d [0:3],
    output logic done
);
```

Module instances:

```
logic [7:0] IV [0:3] = '{8'h34, 8'h55, 8'h0F, 8'h14};

round round_inst (.H_in(H_intermediate), .IV(IV),
.H_out(H_out), .state(state));

FPX fpx_inst (.H(H_fpx), .IV(IV), .d(d));
```

- `round_inst`: This instantiates the `round` module, which handles one round of the hash computation. It takes as input the intermediate hash value (`H_in`) and the initialization vector (`IV`), statically assigned inside the module, and outputs the transformed hash value (`H_out`) and the state (`state`) after the round computation. Note that the H_in wire, as the design specifications says, in sequential logic will be connected via a mux once to the original message while for subsequent rounds to the intermediate state (see Block Diagram Description 3.8).
- `fpx_inst`: This instantiates the `FPX` module. It takes the final hash value from the previous module (`H_fpx`) and the initialization vector (`IV`), and produces the final digest (`d`).

Sequential Logic:
The following code snippet implements the core sequential logic of the design. It is primarily responsible for managing the state transitions of the hash algorithm and updating the internal registers, such as the intermediate hash value and the round counter, based on the clock signal and reset conditions.

```
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        round <= 0;
    end else begin
        state <= next_state;
        if (state == IDLE && start)
            H_intermediate <= m;
        else if (state == CALC_SA)
        begin
            H_intermediate <= H_out;
            round <= round + 1;
        end
        else if (state == CALC_ROUND)
        begin
            round <= round + 1;
            H_intermediate <= H_out;
            if(round == 23)
                H_fpx <= H_out;
        end
        else
            round <= 0;
    end
end
```

Below a detailed explanation of each part:

- `always_ff @(posedge clk or negedge rst_n)`: This process is triggered on the rising edge of the clock or the falling edge of the reset signal (`rst_n`). The sequential logic is implemented within this block to ensure that the state machine and registers are updated correctly on every clock cycle.

- `if (!rst_n) begin ... end`: If the reset signal (`rst_n`) is low (active-low reset), the state is set to `IDLE` and the `round` counter is reset to 0. This initializes the state machine when the system is reset.

- `else begin ... end`: When the reset is not active, the state is updated to the next state, and different actions are performed based on the current state:

  - `if (state == IDLE && start)`: When the state is `IDLE` and the `start` signal is asserted, the input message (`m`) is loaded into the `H_intermediate` register. The `start` is an input flag driven to `1'b1`, when the input message (`m`) on the corresponding input port is valid and stable.

  - `else if (state == CALC_SA)`: In the `CALC_SA` state, the output of the SA function (`H_out`) is stored in `H_intermediate`, and the `round` counter is incremented. This prepares the intermediate hash value for the next round of computation.

- `else if (state == CALC_ROUND)`: During the `CALC_ROUND` state, the hash value is updated by storing the output of the current round into `H_intermediate`, and the round counter is incremented. If the round counter reaches 23 (indicating the last round), the final hash value is stored in `H_fpx`, which will be used for the final permutation.
  Lastly, as we said before, `H_intermediate` has this dual task of initially storing the message in the `CALC_SA` state and subsequently in the `CALC_ROUND` storing the value of the previous round.
- `else round <= 0`: If the state machine is in any other state, the round counter is reset to 0. This was added to ensure multiple sequence computations.

In this sequential block, non-blocking assignments (‘<=‘) are used. These assignments ensure that all the updates to the state and registers, such as `H_intermediate` and `round`, happen simultaneously at the next clock edge.

Combinational Logic:
The following code snippet implements the core combinational logic of the design, specifically handling the state transitions of the hash algorithm.

```
always_comb begin
    next_state = state;
    done = 0;
    case (state)
        IDLE: begin
            if (start)
                next_state = CALC_SA;
        end
        CALC_SA: begin
            next_state = CALC_ROUND;
        end
        CALC_ROUND: begin
            if (round >= 1 && round < 23) begin
                next_state = CALC_ROUND;
            end else begin
                next_state = CALC_FINAL;
            end
        end
        CALC_FINAL: begin
            next_state = DONE;
        end
        DONE: begin
            done = 1;
            next_state = IDLE;
        end
    endcase
end
```

This combinational logic block is responsible for determining the next state of the system based on the current state and inputs. It does not rely on the clock signal, which distinguishes it from sequential logic. Instead, it reacts immediately to changes in the current state or inputs, making it a purely combinational process.

Here's a summary of the most important parts:

- *Initial Setup*: At the start of the `always_comb` block, the next state is initialized to the current state, and the `done` signal is set to 0. This ensures that if no transition conditions are met in the case statement, the state remains unchanged, and the `done` signal is reset.
- *State Transitions*: The state transitions are handled through a case statement that checks the current state and determines the next state:
  - `IDLE`: If the system is in this state and the `start` signal is asserted, the next state is set to `CALC_SA`, initiating the hash computation.
  - `CALC_SA`: After processing the State Array function, the state move to `CALC_ROUND`, where the rounds of transformation will be executed.
  - `CALC_ROUND`: In this state, the system remains here as long as the round counter is between 1 and 22. Once the 23rd round is completed, the state transitions move to `CALC_FINAL`. It may be confusing that the counter goes from 1 to 22; this is because the first round (0) is done in `CALC_SA`, while the last round (23) is done in `CALC_FINAL`.
  - `CALC_FINAL`: This state performs the final operations of the hash algorithm, after which the state transitions to `DONE`.
  - `DONE`: In this state, the `done` signal is asserted to indicate the completion of the hash operation. Afterward, the system returns to the `IDLE` state, ready for a new computation. As the input flag `start` previously indicated the beginning of the operation, now `done` is an output flag driven to `1'b1` when the output digest on the corresponding output port is ready and stable.

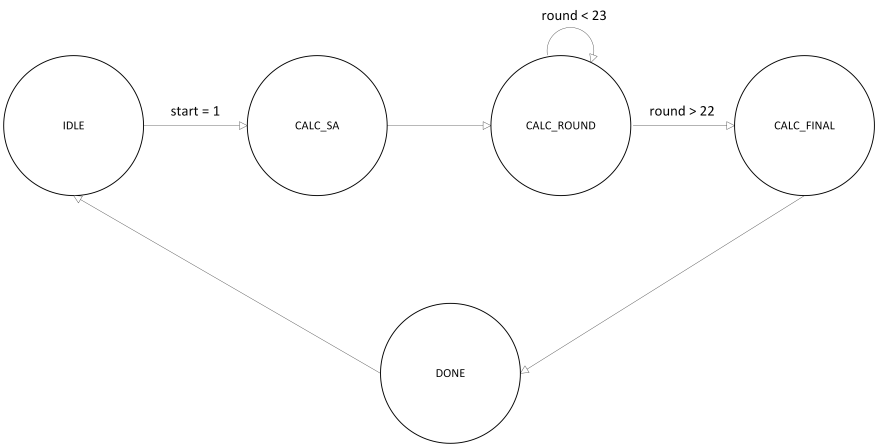

Figure 3.1: *State Machine Diagram for Hash Algorithm*

Figure 3.1 illustrates the state machine used in this design, showing the transitions between the states and the conditions under which they occur. In this combinational block, blocking assignments (‘=‘) are used. Unlike non-blocking assignments, blocking assignments execute statements sequentially, in the order they appear in the code. This is appropriate for combinational logic, where the order of execution is important and the result of one assignment can immediately affect subsequent assignments.

3.3 Round Module: Hash_round.sv

The *Hash_round.sv* module performs the core transformation in each round of the algorithm. It takes the current state as input, applies the State Array (SA) function, followed by the Theta and Rho transformations, and outputs the new state.

```
module round(
    input  logic [7:0] H_in [0:3],
    input  logic [7:0] IV [0:3],
    input logic [2:0] state,
    output logic [7:0] H_out [0:3]
);

    SA sa_inst (.m(H_in), .IV(IV), .H(H_intermediate_SA));
    XOR_Module xor_inst (.H_in(H_in), .IV(IV),
    .H_out(H_intermediate_XOR));

    Theta theta_inst (.H_in(H_intermediate),
    .H_out(H_intermediate_Theta));

    rho rho_inst (.H_in(H_intermediate_Theta),
    .H_out(H_intermediate_rho));
```

Combinational Logic:
The following block of combinational logic acts as a multiplexer, selecting the appropriate value to assign to the `H_intermediate` register based on the current state of the system.

```
always_comb begin
case (state)
    CALC_SA:    begin
                H_intermediate = H_intermediate_SA;
                end
    CALC_ROUND: begin
                H_intermediate = H_intermediate_XOR;
                end
    default:    for (int i = 0; i < 4; i++)
        begin
        H_intermediate[i] = 8'b0;
        end
endcase
```

```
end
assign H_out = H_intermediate_rho;
```

From this point forward, the submodules will not be commented in detail, as they have been thoroughly discussed in the previous sections. These modules follow similar structures and logic, already explained above.

3.4 State Array Module: SA.sv

```
module SA(
    input  logic [7:0] m [0:3],
    input  logic [7:0] IV [0:3],
    output logic [7:0] H [0:3]
);
    always_comb begin
        for (int i = 0; i < 4; i++) begin
            H[i] = m[i] ^ IV[i];
        end
    end
endmodule
```

3.5 Theta and Rho Modules: Theta.sv - Rho.sv

```
module Theta(
    input  logic [7:0] H_in [0:3],
    output logic [7:0] H_out [0:3]
);
    // Implementazione di permutazione custom
    always_comb begin
        for (int i = 0; i < 4; i++) begin
            H_out[i] = H_in[3-i];
        end
    end
endmodule

module rho(
    input  logic [7:0] H_in [0:3],
    output logic [7:0] H_out [0:3]
);
    always_comb begin
        for (int i = 0; i < 4; i++) begin
            H_out[i] = (H_in[i] + 8'h85) % 8'hFD;
        end
    end
endmodule
```

3.6 Final Permutation and XOR Module: FPX.sv

```
module FPX(
    input  logic [7:0] H [0:3],
    input  logic [7:0] IV [0:3],
    output logic [7:0] d [0:3]
);
    always_comb begin
        for (int i = 0; i < 4; i++) begin
            d[i] = H[3-i] ^ IV[i];
        end
    end
endmodule
```

3.7 XOR Module: Xor_module.sv

```
module XOR_Module(
    input  logic [7:0] H_in [0:3],
    input  logic [7:0] IV [0:3],
    output logic [7:0] H_out [0:3]
);
    always_comb begin
        for (int i = 0; i < 4; i++) begin
            H_out[i] = H_in[i] ^ IV[i];
        end
    end
endmodule
```

3.8 Block Diagram Description

In this section, we provide an overview of the block diagrams representing the structure and data flow of the hash algorithm. These diagrams illustrate how the main components and submodules are interconnected to achieve the desired functionality.

3.8.1 Block Diagram

The main block diagram, as shown in Figure 3.2, provides a high-level overview of the overall architecture. Only the most important parts will be presented below.
Description

- *Round Logic:* The core logic that processes the input message in multiple rounds to produce the intermediate hash values. It takes in the current state, the intermediate hash value (`H_intermediate`), and the Initialization Vector (`IV`), and produces the output for the current round. The round logic is also responsible for updating the state based on the current round and controlling the flow of data through the rounds.

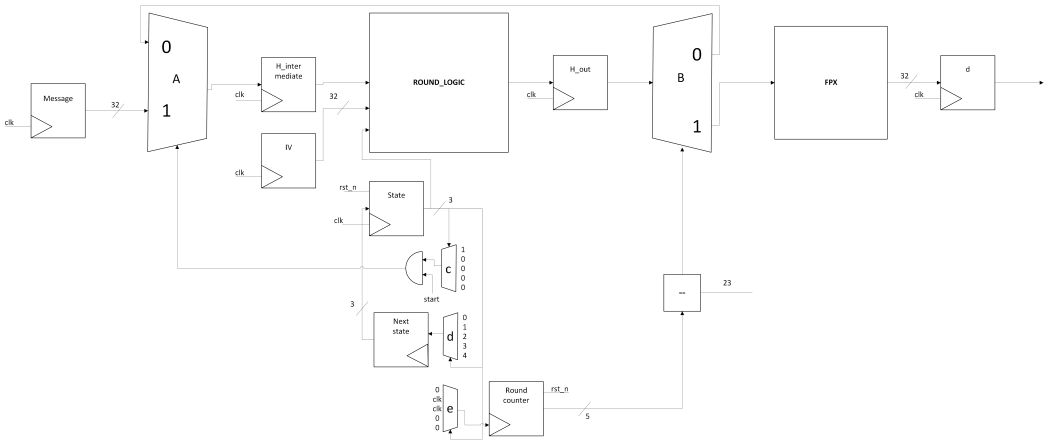

Figure 3.2: *Main Block Diagram of the Hash Algorithm*

- *FPX Module:* After all rounds are completed, the final permutation and XOR operations are performed in the FPX module. The output of this module is the final hash digest (`d`).

- *Multiplexer A:* The A mux is in charge of managing what to input into the `round_logic`. It is driven by a `and (&)` between the `start` signal and the `state`. In fact, if the state is `IDLE` represented by the logical value 1 in the mux c and `start` is active, then the `round_logic` will have the message as input. Otherwise, in all other states or if start is off `round_logic` will have the feedback of the output processed by the round.

- *Multiplexer B:* The B mux instead, is driven by the comparator between the `round_counter` and 23. This because, if the round counter has reached the end of its cycle it must connect the round output to FPX, otherwise it will connect the round output in retroactive feedback.

3.8.2 Round Block Diagram

The round block diagram, shown in Figure 3.3, details the specific logic applied during each round of the hash computation.

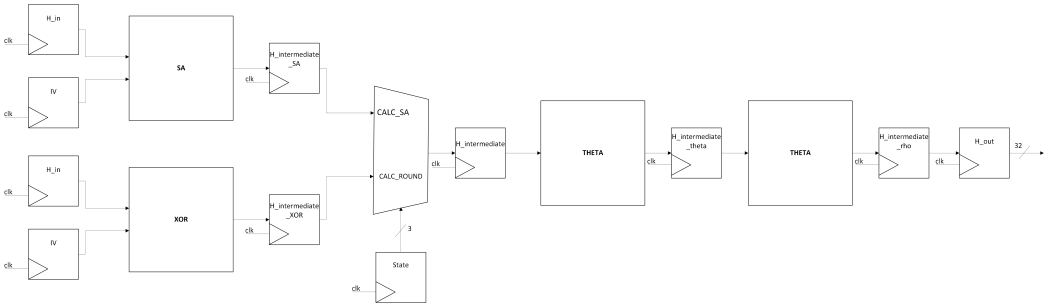

Figure 3.3: *Round Block Diagram of the Hash Algorithm*

Description

- *Multiplexer:* This mux selects between the outputs of the SA and XOR modules (`H_intermediate_SA` and `H_intermediate_XOR`, respectively), based on the current state of the system. The selected value is then directed to the next stage of the computation.

3.9 Architectural Choice: Single-Inter-Round-Pipelined

In this project, we opted for the **Single-inter-round-pipelined architecture** for implementing the hash algorithm.

3.9.1 Why Single-Inter-Round-Pipelined?

The Single-inter-round-pipelined architecture offers a balance between resource utilization and performance. Specifically, it has the following key advantages:

- **Resource Efficiency**: The architecture consumes relatively low resources (A_R) because only one round's logic is instantiated. This is beneficial for designs where area (e.g., ALMs in FPGA) is a constraint.
- **High Operating Frequency**: By using a short combinational path (the propagation delay of a single round), this architecture allows for a higher operating frequency.
- **Scalability**: This architecture can easily scale with the number of rounds, as the logic remains consistent regardless of the number of iterations. The state is passed from one round to the next without significant resource overhead.

However, this architecture does have some drawbacks, mainly:

- **Increased Latency**: The latency increases linearly with the number of rounds. For example, if the algorithm requires 24 rounds, the total latency will be 24 clock cycles.

3.9.2 Comparison with Other Architectures

To further justify the choice of the Single-inter-round-pipelined architecture, we compare it to other possible architectures:

Unrolled Architecture

In the unrolled architecture, all rounds are implemented sequentially in a single clock cycle. This results in:

- **Low Latency**: The entire algorithm completes in one clock cycle.
- **Long Combinational Path**: Since all rounds are executed within one cycle, the combinational logic path becomes very long, potentially reducing the maximum operating frequency.
- **High Resource Consumption**: Each round's logic is instantiated, leading to significant resource consumption.

While the unrolled architecture offers the advantage of low latency, it is less suited for resource-constrained environments due to its high area and lower achievable clock frequency.

Cascaded Single-Inter-Round-Pipelined Architecture

The cascaded single-inter-round-pipelined architecture places registers between each round, allowing for one round to be completed per clock cycle but requiring significant resources due to the multiple registers and combinational logic blocks. It has:

- **Low Clock-Per-Data (CPD)**: Each clock cycle produces a new output, resulting in a high throughput.
- **High Resource Usage**: Since every round has its own register, the area consumption is higher than in the single-inter-round-pipelined architecture.

While this architecture provides higher throughput, it is not as resource-efficient as the Single-inter-round-pipelined architecture, making it less suitable for this project.

3.9.3 Final Thoughts and Design Choices

The **Single-inter-round-pipelined architecture** strikes a balance between resource usage and performance. By utilizing a register between rounds, it allows for higher frequencies without the excessive resource consumption seen in unrolled architectures. Though it incurs higher latency compared to unrolled designs, the trade-off is justified by the significant savings in area and the ability to achieve higher clock frequencies, which are critical for this project.

The only notable downside of our choice is the increased latency. However, as mentioned earlier, the latency is directly proportional to the number of rounds, and in this algorithm the number of rounds is 24, not excessively high, and therefore the latency remains within an acceptable range. Given this, we determined that the Single-inter-round-pipelined architecture was the optimal choice for our design.

To further substantiate our choice, we performed a comparative analysis between the different architectures, focusing on key metrics such as area, operating frequency, latency, throughput and efficiency. The calculations are summarized in the tables below 3.1.

Table 3.1: *Architectural Comparison Calculations*

Architecture	Area	Frequency	Latency	CPD	Throughput	Efficiency
Unrolled	$N \cdot A_R$	$\frac{1}{N \cdot t_R}$	1 cc	1 cc	$\frac{Q}{N \cdot t_R}$	$\frac{Q}{N^2 \cdot t_R \cdot A_R}$
Single-inter-round	A_R	$\frac{1}{t_R}$	N cc	N cc	$\frac{Q}{N \cdot t_R}$	$\frac{Q}{N \cdot t_R \cdot A_R}$
Cascaded	$> N \cdot A_R$	$\frac{1}{t_R}$	N cc	1 cc	$\frac{Q}{t_R}$	$\sim \frac{Q}{N \cdot t_R \cdot A_R}$

Commentary on Results

The results presented in the last table 3.2 provide a clear comparison of the different architectural choices based on our specific implementation metrics.

Table 3.2: *Calculated Values for Our Specific Implementation*

	Area	**Frequency**	**Latency**	**CPD**	**Throughput**	**Efficiency**
Single-inter-round	149	51.49 MHz	24 cc	24 cc	68.7 Mbps	0.46 Mbps/ALM
Unrolled	3576	2.1 MHz	1 cc	1 cc	68.7 Mbps	19 Kbps/ALM
Cascaded	> 3576	51.49 MHz	24 cc	1 cc	1.65 Gbps	19 Kbps/ALM

- **Area:** The area values correspond to the number of Adaptive Logic Modules (ALMs) reported by Quartus. The Single-inter-round architecture requires only 149 ALMs, which is significantly lower than the Unrolled and Cascaded architectures. This makes it the most resource-efficient option in terms of area consumption.
- **Frequency:** The clock frequency was calculated based on the clock period (19.42 ns on Quartus), with a resulting frequency of 51.49 MHz for the Single-inter-round and Cascaded architectures. The Unrolled architecture, however, achieves only 2.1 MHz due to its long combinational path, which limits the maximum operating frequency.
- **Latency:** The latency directly corresponds to the number of rounds, which is 24 for both the Single-inter-round and Cascaded architectures. In contrast, the Unrolled architecture has a latency of just 1 clock cycle.
- **Cycles per Data (CPD):** This metric reflects the number of clock cycles required to process a single unit of data. For the Single-inter-round architecture, the CPD is directly proportional to the latency, meaning it requires 24 clock cycles to complete the processing of one data block. In contrast, the Unrolled and the Cascaded architecture processes all rounds in one clock cycle, resulting in a CPD of 1.
- **Throughput:** The throughput for the Single-inter-round architecture is calculated using $Q = 32$, as we are working with 32-bit data. Both the Single-inter-round and Unrolled architectures achieve 68.7 Mbps, but the Cascaded architecture, with its ability to process data every clock cycle, reaches a much higher throughput of 1.65 Gbps.
- **Efficiency:** When considering the efficiency, defined as the throughput per area, the Single-inter-round architecture demonstrates a balanced performance, achieving 0.46 Mbps/ALM. While the Cascaded architecture also achieves 0.46 Mbps/ALM, it does so with a much higher area and resource cost. The Unrolled architecture, although efficient in terms of cycles per data, achieves only 19 Kbps/ALM due to its low operating frequency and high area consumption.

In conclusion, even with the values calculated specifically for our implementation, the Single-inter-round architecture emerges as the optimal choice. It provides a good balance between area, frequency, and throughput while maintaining a reasonable level of efficiency. The Unrolled architecture, while fast in terms of CPD and latency, suffers from low frequency and high area requirements. The Cascaded architecture, although offering high throughput, is not as resource-efficient, making it less attractive given the higher area costs. Therefore, the Single-inter-round architecture remains the best option for our design, offering the most balanced trade-offs.

CHAPTER 4

Interface Specifications and Expected Behavior

4.1 Introduction

This chapter shows how to drive inputs and outputs of Light Hash algorithm v4 to obtain an expected behaviour. In particular, this section will display the module ports, the correct timing and some use case.

4.2 Module Ports

The module has four logic inputs and two logic outputs.

4.2.1 Input Ports

As input the module takes:

- clk: The clock signal
- rst_n: The asynchronus, active low reset signal
- $start$: The input flag that ensures that the input data are ready to be ridden
- m: The four byte register that contains the input message

4.2.2 Output Ports

The two corresponding outputs are:

- d: The four byte register that contains the resulting hashed data
- $done$: The output flag that ensure that the computation is done and the output data are ready

4.3 Timing and Expected Behaviour

4.3.1 Input Logic Timing

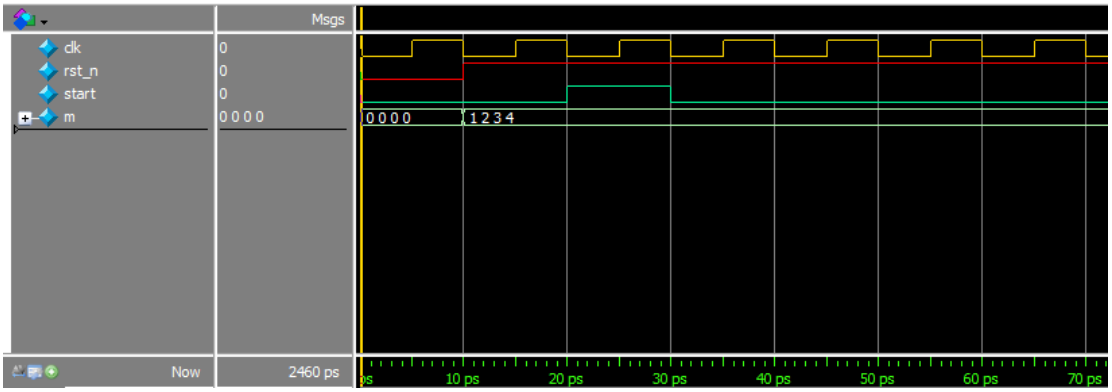

Figure 4.1: *Input logic waveform*

The expected behaviour of the input signals is presented in figure 4.1. The first input is the clock signal. It changes polarity from zero to one and vice versa every 5 ps in this model. Then there's the reset signal `rst_n` that is active for the first 10 ps and after is removed.
After that there' s the activation of `start` for which the module assume that the message is already there and it's ready to be ridden, thus the input `m` , must be correctly sets before 20ps, the moment in which the start signal is activated.

4.3.2 Output Logic Timing

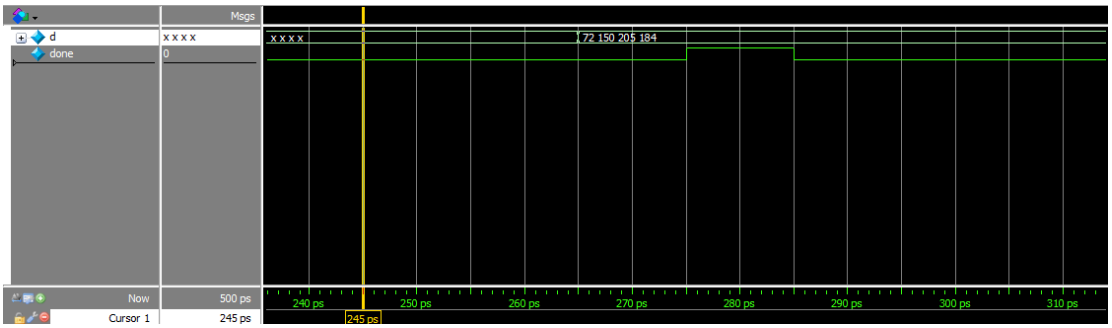

Figure 4.2: *Output logic waveform*

Concerning about the output logic waveform, showed in figure 4.2, `done` became active after 275ps from the start of the simulation. That means that, as the input data, the output `d` must be correctly sets before 275ps and , as the picture demonstrates, it's ready at 265ps, so the output timing in this example is respected.

4.4 Use Case: Main case

The main case consist of a standard configuration of input that drives the module with an expected behaviour.

As presented in figure 4.1 , the standard case consist of a clock that change every 5 ps, a reset signal disabled after 10 ps, the message m, ridden after the deactivation of `rst_n`, the start signal activated after 20 ps and disabled after 30ps. Specifically, `m` is "1 2 3 4".
The expected outputs for this main case are that `d` is ready after 265ps and it's equal to "72 150 205 184". Instead `done` is activated after 275ps to guarantee that the output is truly ready.

4.5 Use Case: Corner Case

In this model five corner cases have been identified and the use cases for each will be shown below.

4.5.1 Reading after start

This case manifests when the module's inputs are bad driven. In particular, the problem consist of reading the message `m` after the activation of `start`. This results in a useless reading, because the computation will start with the old value of the message.

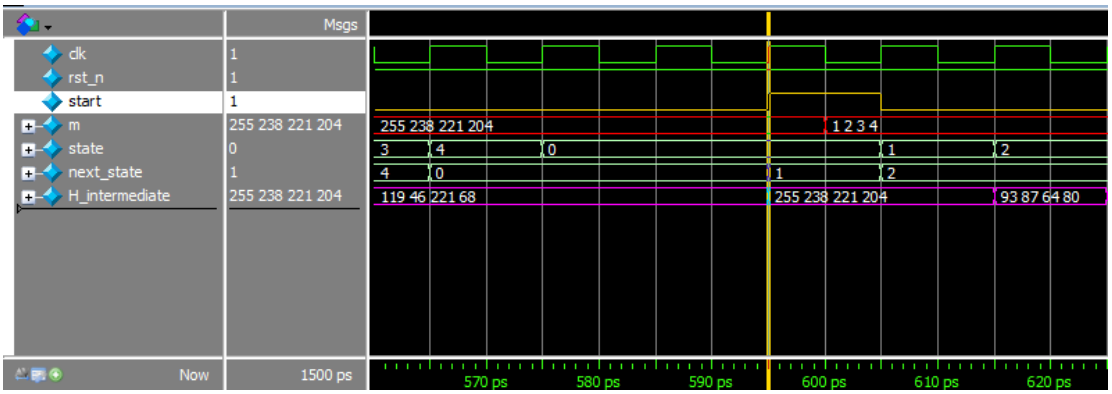

Figure 4.3: *Input of first corner case*

As presented in figure 4.3 , the message input `m` (red signal) changes after that `start` (gold signal) is activated, but the input register for the computation `H_intermediate` (magenta signal) contains the old value of `m`, meaning that the reading of "1 2 3 4" is ignored.

4.5.2 Reset activation during computation

This case happens when, during a computation `rst_n` is activated.
Starting from the main case, it's enough to activate the reset signal and then start another computation to see that the module is well designed and it restart the computation without any bad value.

4.5.3 Input larger then four bytes

The third corner case consists of providing an input larger than the one that is expected. As explained in chapter 5, test vectors have been used to insert the message. To verify this behaviour, the message m reads *Text_Vector3.hex*, in which there's an input of 8

bytes. The result is the same of the main case, because `m` is defined to be a 4 byte register and the reading operation only reads 4 bytes.

4.5.4 Activation of start during computation

In this case, after a normal computation has been started,`start` is accidentally reactivated and left active. As showed in figure 4.4 , `start` (gold) is re-activated after 80ps

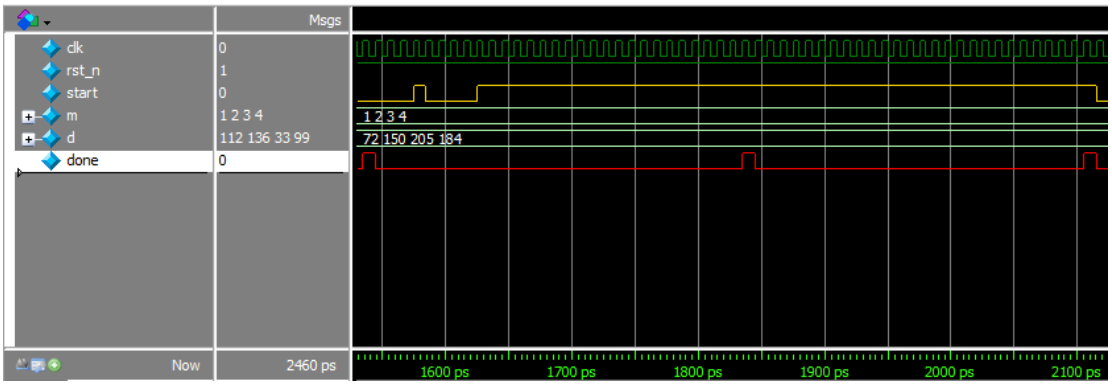

Figure 4.4: *Waveform of fourth corner case*

and left active. Firstly it is ignored and the computation ended like the main case. After the done signal has been activated, another computation is triggered due to the fact that the start signal is on. If `start` is not deactivated, it will cause an infinite triggering of new computation.

4.5.5 Start and Reset activation

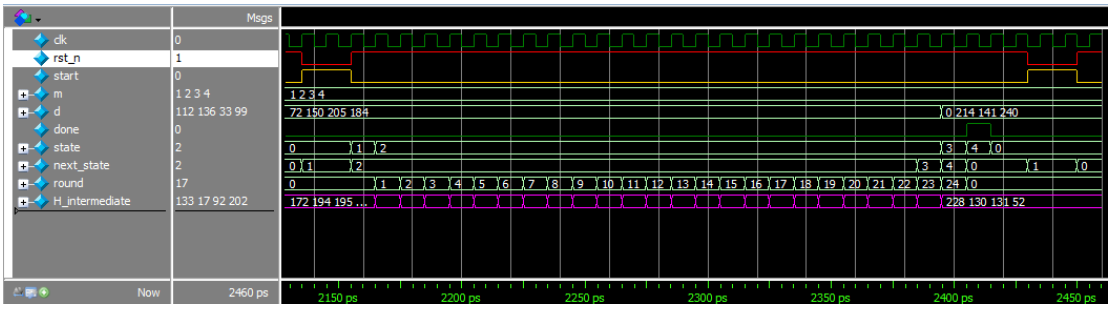

Figure 4.5: *Start and reset disabling*

This last corner case occurs when `start` and `rst_n` are simultaneously activated. In this scenario, two different behaviors can occur, depending on when the signals are disabled.

Signals Disabled on the Rising Edge of the Clock

When the reset signal is activated, the state and the round counter are reset, but the combinational logic causes the "next state" to change. If the signals are disabled at the rising edge of the clock, a problem arises because, even if the state evolves correctly to `CALC_SA`, the intermediate register `H_intermediate` does not have

enough time to read the input of `m`, leading to a computation based on the old value of `H_intermediate` rather than the actual input (first part of figure 4.5).

Signals Disabled on the Falling Edge of the Clock

If the signals are disabled at the falling edge of the clock, the sequential logic does not detect the start signal activation. As a result, it only recognizes the reset action, and the module will remain inactive (last part of figure 4.5).

CHAPTER 5

Functional Verification

5.1 Intro

This chapter shows how the module has been tested. The testbench is composed of different tests, and it takes 2460ps to reach the end of the simulation. Below, the structure of the testbench is presented, starting from basic assignments to the corner cases explained in Chapter 4.

5.2 Basic definitions and assignments

```
// Input Parameters
logic clk;
logic rst_n;
logic start;
logic [7:0] m [0:3];

//  Output Parameters
logic [7:0] d [0:3];
logic done;

//DUT (Device Under Test)
Hash_light_top dut (
.clk(clk),
.rst_n(rst_n),
.start(start),
.m(m),
.d(d),
```

```
.done(done)
);
```

The module *hash_light_tb.sv* starts with the definition of the logic that the Device Under Control will take as input/output. That logic consist in the clock `clk`, the reset signal `rst_n`, the start signal `start` and the four bytes register `m`. Then there's the definition of the four bytes register `d` and the definition of the flag `done`.
After that there's the declaration of an instance of the module, with a proper connection of inputs and outputs.Next is the always section that assigns a period of 10ps at the clock.

```
// Clock generation
always #5 clk = ~clk;

initial begin
clk = 0;
rst_n = 0;
start = 0;
m[0] = 8'h00;
m[1] = 8'h00;
m[2] = 8'h00;
m[3] = 8'h00;

// Reset deassertion
#10;
rst_n = 1;
```

The last basic elements are inside the initial block, in which there's specified that all the input starts with a value equal to zero. To permits the tests `rst_n` is disabled after 10 ps.
Following the initial block, there are seven test that have been used to test the module. In each one of those tests, the value of m has been assigned using test vectors.

5.3 Tests with known values

The first two tests consist of two standard computations with known values.

```
// Test Case 1 --> standard model
// Test with known values
$readmemh("../modelsim/tv/Test_Vector1.hex",m);
// starting hash computation
#10;
start = 1;
#10;
start = 0;

// waiting for the end of computation
wait(done) @ (posedge clk);
```

```
// shows digest output
$display("Digest: %h %h %h %h", d[0], d[1], d[2], d[3]);
```

In the case presented, `m` reads the value inside *Test_Vector1.hex* and it's equal to "1 2 3 4". After 10ps `start` is activated and it still active for other 10ps. Then the operation *wait*, hold for the done signal and after the test displays in the terminal the resulting digest.
The same structure has been used for the second test, but the message `m` reads the value from *Test_Vector2.hex*.

5.4 First corner case

```
// Test Case 3: 1st Corner Case
// reading of known value but after the rise of start signal

// starting hash computation
#10;
start = 1;
#5;
$readmemh("../modelsim/tv/Test_Vector1.hex",m);
#5
start = 0;


// waiting for the end of computation
wait(done) @ (posedge clk);

// shows digest output
$display("Digest: %h %h %h %h", d[0], d[1], d[2], d[3]);
```

According to the use case specified in chapter 4, the first corner case consist of reading the value of `m` from the test vector after the activation of `start`. As showed in the code, the operation of *readmemh* has been done 5ps after the activation of the start signal. The result are the ones obtained in chapter 4

5.5 Second corner case

```
// Test Case 4: 2nd Corner Case
// Reset activation during a round and then restart
#10;
$readmemh("../modelsim/tv/Test_Vector1.hex",m);
// starting hash computation
#10;
start = 1;
#10;
start = 0;
```

```
#50
rst_n = 0;
#10
rst_n =1;

#10;
start = 1;
#10;
start = 0;
 // waiting for the end of computation
wait(done) @ (posedge clk);
// shows digest output
$display("Digest: %h %h %h %h", d[0], d[1], d[2], d[3]);
```

The second corner case consist of the activation of `rst_n` during a computation. To test that, after 50ps from the deactivation of the start signal, `rst_n` has been sets to zero. To restart the computation the reset signal has been disabled and `start` has been activated again for 10ps.

5.6 Third corner case

In the third corner case, the objective was to test what happens if the message `m` takes an input larger than four bytes.
The testbench module for this case is equal to the one presented in the standard case, except for the fact that `m` takes as input the value contained in *Test_Vector3.hex*.

5.7 Fourth corner case

```
// Test Case 6: 4th Corner Case
// Start signal reactivated and left active
#10;
$readmemh("../modelsim/tv/Test_Vector1.hex",m);
// starting hash computation
#10;
start = 1;
#10;
start = 0;
#40;
start = 1 ;
// waiting for the end of computation
wait(done)@ (posedge clk);
#10;
wait(done)@ (posedge clk);
start=0;
// shows digest output
$display("Digest: %h %h %h %h", d[0], d[1], d[2], d[3]);
```

The fourth corner case consists of activating the start signal during a computation and leaving that signal active. As showed in the code, the testing of this case is made by the reactivation of `start` after 40ps. Next there are two *wait* operations to show that if the start signal remains active, another computation will start after the first one. To stop that and to going on with the lasts test, the signal is disabled after the second *wait*.

5.8 Fifth corner case

In the last corner case, there is simultaneous activation of `start` and `rst_n`, and the behavior changes depending on when these two signals are disabled. For this reason, there are two different tests that differ only in the timing of when the signals are disabled. In the first test, the signals are disabled at the rising edge of the clock, while in the second test, they are disabled at the falling edge of the clock.

```
 #10;
rst_n = 0;
start = 1;
#20;
rst_n=1;
start = 0;
```

Figure 5.1: *Signals disabled on the clock's rising edge*

```
#5;
rst_n = 0;
start = 1;
#20;
rst_n=1;
start = 0;
```

Figure 5.2: *Signals disabled on the clock's falling edge*

CHAPTER 6

FPGA Implementation Results

6.1 Introduction

This chapter describes the steps adopted on Quartus to implements the model designed at Register Transfer Level into a FPGA board. Specifically, the FPGA device used is the **5CGXFC9D6F27C7**.
The most important steps taken are: **Virtual Pins Assignment**,**Logical and Physical analysis**, **Static Time Analysis**.

6.2 Virtual Pins Assignment

Before starting the logical and physical analysis, Quartus needs to know whether the pins are all physical or if there are any virtual pins, i.e. pins connected to other ALMs rather than directly to the FPGA's physical pins. As showned in 6.1, in this model there are 67 virtual pins and only 1 physical pin that is the clock signal.

6.3 Logical and Physical Analysis

The first two analyses carried out are the logical, made by the Analysis and Synthesis step of Quartus compilation design, and the physical, made by the Fitter phase.
After those, it's possible to see in 6.1 that there are 74 total register used across all of the 109 ALMs needed.
It's also possible to notice in 6.2 that from 109 ALMs implemented, 34 are not available because they are used for the connection of virtual pins. Of the remaining 75, 20 are used for both LUT logic and registers, 39 only for LUT logic and 16 only for registers.

Flow Status	Successful - Thu Sep 5 12:06:59 2024
Quartus Prime Version	23.1std.1 Build 993 05/14/2024 SC Lite Edition
Revision Name	hash_light_top
Top-level Entity Name	Hash_light_top
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	109 / 113,560 (< 1 %)
Total registers	74
Total pins	1 / 378 (< 1 %)
Total virtual pins	67
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 %)

Figure 6.1: *Flow Summary after Fitting Phase*

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	109 / 113,560	< 1 %
2	ALMs needed [=A-B+C]	109	
1	[A] ALMs used in final placement [=a+b+c+d]	75 / 113,560	< 1 %
1	[a] ALMs used for LUT logic and registers	20	
2	[b] ALMs used for LUT logic	39	
3	[c] ALMs used for registers	16	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	0 / 113,560	0 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	34 / 113,560	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	34	
3			
4	Difficulty packing design	Low	

Figure 6.2: *Fitter resources allocation*

6.4 Static Time Analysis

This final phase aims to understand if the model respects the time constraints as specified in the constraints file *time_constr_template.sdc* inside the Quartus directory. This analysis divides the model into register to register paths and checks that the constraints are met according to the maximum clock frequency. To satisfy the constraints, the maximum clock frequency has been sets equal to 51.49 MHz that corresponds to a clock period of 19.42ns.These values were chosen considering the two models `Slow 1100mv 85°C` and `Slow 1100mv 0°C` and selecting the lowest frequency value to ensure the model would work even in the worst conditions.