Prova Finale (Progetto di Reti Logiche)

Tesi di Laurea Magistrale in
Computer Engineering

Authors: **Matteo Civitillo, Alessandro Paolo Gianni Callegari**

Student IDs: 984313, 980921
Advisor: Prof. William Fornaciari
Academic Year: 2023-24

Contents

Introduction

In the contemporary world, characterised by relentless advancements in technology, ensuring the reliable transmission of information over noisy channels has become paramount across a wide range of sectors. Error correction code decoders play a pivotal role in this context, facilitating the maintenance of data integrity even under disturbed transmission conditions. Industries such as storage support, cellular networks, and digital communication heavily rely on these devices to ensure the quality and reliability of their operations.

The "Prova Finale (Progetto di Reti Logiche)" for the Academic Year 2023/2024 stipulates the implementation of an error correction device with a focus on the management of data correction within a RAM module. This RAM stores a sequence of K words W, with values ranging from 0 to 255, starting from a designated address. The words are saved at intervals of two bytes, with the intermediate byte set to zero.

The component to be implemented must facilitate the reading of the words from RAM, replace any zeros (which act as "unspecified values") with the last valid value read, and assign a credibility value to the missing byte. The credibility value indicates the reliability of the assigned value for the lost data and is determined based on the sequence of readings. In the event of the component reading a word that differs from zero, the credibility is set to 31 and undergoes a subsequent decrement with each reading of an unspecified value. This value is only reset upon the next reading of a valid value, thereby reflecting the probability of accuracy of the assigned value.The more unspecified values that are read, the more data is lost, increasing the likelihood that the substituted value significantly differs from the originally assigned one.

The following example illustrates the functionality of the system (bold indicates words W):

- Input sequence given to the component (K = 14):

  128 0 **64** 0 **0** 0 **0** 0 **0** 0 **0** 0 **0** 0 **100** 0 **1** 0 **0** 0 **5** 0 **23** 0 **200** 0 **0** 0

- Output sequence from the component:

  128 31 **64** 31 **64** 30 **64** 29 **64** 28 **64** 27 **64** 26 **100** 31 **1** 31 **1** 30 **5** 31 **23** 31 **200** 31 **200** 30

1 | Architecture

1.1. Overview

A state machine has been designed to implement the required functionalities without the need for external components, thereby minimizing signal propagation delay as much as possible. The machine manages the "start" and "reset" inputs, generates the "o_done" control signal to indicate the completion of execution, and directly communicates with the memory (RAM) through the corresponding signals.

For greater clarity, the states have been categorized into four distinct groups:

1. States responsible for handling the reading of the first value, verifying whether it is zero, and managing any subsequent occurrences of zero (highlighted in purple).
2. States S2 and S7, which evaluate the value just read from RAM and determine the next state based on it (highlighted in blue).
3. States dedicated to processing the reading of a non-zero value (highlighted in yellow).
4. States responsible for replacing the zero in memory and decrementing the credibility value (highlighted in pink).

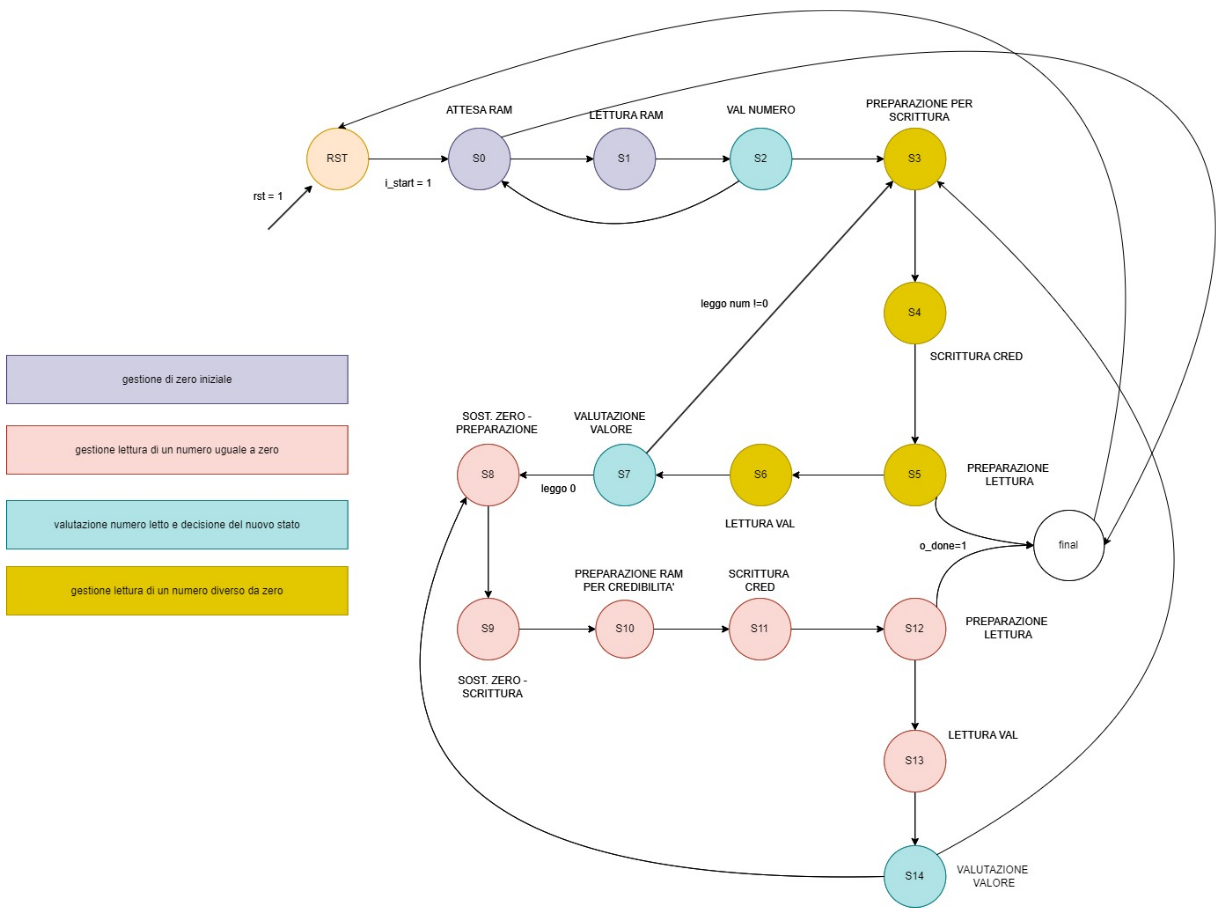


Figure 1.1: State machine

1.2. State Description

1.2.1. Rst (Reset State)

1. The machine has received the reset signal and has been reinitialized for a new reading.
2. It remains in this state as long as $i\_start = 0$.
3. When $i\_start = 1$, it enables the RAM ($o\_mem\_en = 1$), starts storing the first address to be read ($curr\_addr = i\_add$), and initializes the counter ($counter = i\_k$).
4. All signals are also reset to zero.

1.2.2. S0 (Waiting for RAM)

1. The machine waits for the RAM to be ready for reading the first value.
2. Meanwhile, it provides the first address that will be read ($o\_mem\_addr = curr\_addr$).

3. If $counter = 0$, it transitions to the **final** state, setting $o\_done = 1$; otherwise, it moves to **S1**.

1.2.3. S1 (Searching for the Value in RAM at the Given Address)

1. This state is used by RAM to output the value stored at the previously provided address. Meanwhile, the counter is decremented ($counter = counter - 1$).
2. The system transitions to **S2**.

1.2.4. S2 (Value Evaluation)

1. If the read value is zero ($i\_mem\_data = 0$):
   - Increment $curr\_addr$ by 2 ($curr\_addr = curr\_addr + 2$).
   - Return to state **S0**.
2. If the read value is different from zero:
   - Increment the address by 1 ($curr\_addr = curr\_addr + 1$).
   - Initialize the credibility value to 31 ($mem\_c = 31$).
   - Store the value as the last valid value read ($ultimo\_valido = i\_mem\_data$).
   - Transition to **S3**.

1.2.5. S3 (Preparing RAM for Writing)

1. The write signal is enabled ($o\_mem\_we = 1$).
2. The RAM is provided with the address to write to ($o\_mem\_addr = curr\_addr$) and the content to be written ($o\_mem\_data = mem\_c$).
3. The system transitions to **S4**.

1.2.6. S4 (Writing Credibility to RAM)

1. Clock cycle used by the RAM to write the credibility value.
2. The address is incremented for the next read operation ($curr\_addr = curr\_addr + 1$).
3. The system transitions to **S5**.

1.2.7. S5 (Preparing for Reading)

1. The RAM is prepared for reading ($o\_mem\_we = 0$) and provided with the next reading address ($o\_mem\_addr = curr\_addr$).

2. If $counter = 0$, the system transitions to the **final** state, setting $o\_done = 1$; otherwise, it moves to **S6**.

1.2.8. S6 (RAM Reading Process)

1. The counter is decremented ($counter = counter - 1$).
2. The system transitions to **S7**.

1.2.9. S7 (Value Evaluation)

1. If the read value is zero ($i\_mem\_data = 0$):
   - Decrease the credibility ($mem\_c = mem\_c - 1$).
   - Transition to **S8**, where the case of a zero value is handled.
2. If the read value is different from zero:
   - Increment the address by 1 ($curr\_addr = curr\_addr + 1$).
   - Store the value as the last valid value read ($ultimo\_valido = i\_mem\_data$).
   - Transition to **S3**.

1.2.10. S8 (Zero Replacement in Memory - Writing Preparation)

1. The RAM is enabled for writing ($o\_mem\_we = 1$).
2. The content to be written is passed to RAM ($o\_mem\_data = ultimo\_valido$).
3. The system transitions to **S9**.

1.2.11. S9 (Zero Replacement in Memory - Writing)

1. Clock cycle used by the RAM to write.
2. The address is incremented for the next credibility writing operation ($curr\_addr = curr\_addr + 1$).
3. The system transitions to **S10**.

1.2.12. S10 (Preparing RAM for Writing Credibility After Reading Zero)

1. The RAM is provided with the address to write to ($o\_mem\_addr = curr\_addr$) and the content to be written ($o\_mem\_data = mem\_c$).
2. The system transitions to **S11**.

1.2.13. S11 (Writing Credibility to RAM)

1. Clock cycle used by the RAM to write the credibility value.
2. The address is incremented for the next reading operation ($curr\_addr = curr\_addr + 1$).
3. The system transitions to **S12**.

1.2.14. S12 (Preparing for Reading)

1. The RAM is prepared for reading, resetting $o\_mem\_we = 0$, and provided with the new address ($o\_mem\_addr = curr\_addr$).
2. If $counter = 0$, the system transitions to the **final** state, setting $o\_done = 1$; otherwise, it moves to **S13**.

1.2.15. S13 (Searching for Value in RAM)

1. This state is used by RAM to output the value stored at the previously provided address. Meanwhile, the counter is decremented ($counter = counter - 1$).
2. The system transitions to **S14**.

1.2.16. S14 (Value Evaluation)

1. If the read value is another zero, the system transitions back to **S8**, decrementing credibility.
2. If the read value is different from zero, the system transitions to **S3**, updating the credibility to 31 ($mem\_c = 31$), incrementing the address ($curr\_addr = curr\_addr + 1$), and storing the new value as the last valid one ($ultimo\_valido = i\_mem\_data$).

1.2.17. Final (Resetting Signals and Entering Standby Mode)

1. If $i\_start = 0$:
   - Set $o\_done = 0$.
   - Transition to **Rst**, waiting for a new start or reset signal.
2. If $i\_start = 1$, remain in the **final** state until the start signal goes to zero.

1.3. Component Implementation and Signal Management

To implement the component, all operations have been incorporated within a single process. Given that the state machine consists of a relatively small number of states, this approach allows for efficient state transitions and facilitates modifications when necessary.

Within the component, four signals have been employed to manage information:

- `signal mem_c: std_logic_vector(7 downto 0);`
  - Used to store the last credibility value written to memory.
- `signal counter: std_logic_vector(9 downto 0);`
  - Used to count the number of words remaining to be read.
- `signal address_curr: std_logic_vector(15 downto 0);`
  - Stores the current address from which the next value is to be read.
- `signal ultimo_valido: std_logic_vector(7 downto 0);`
  - Stores the value of the last valid word read.

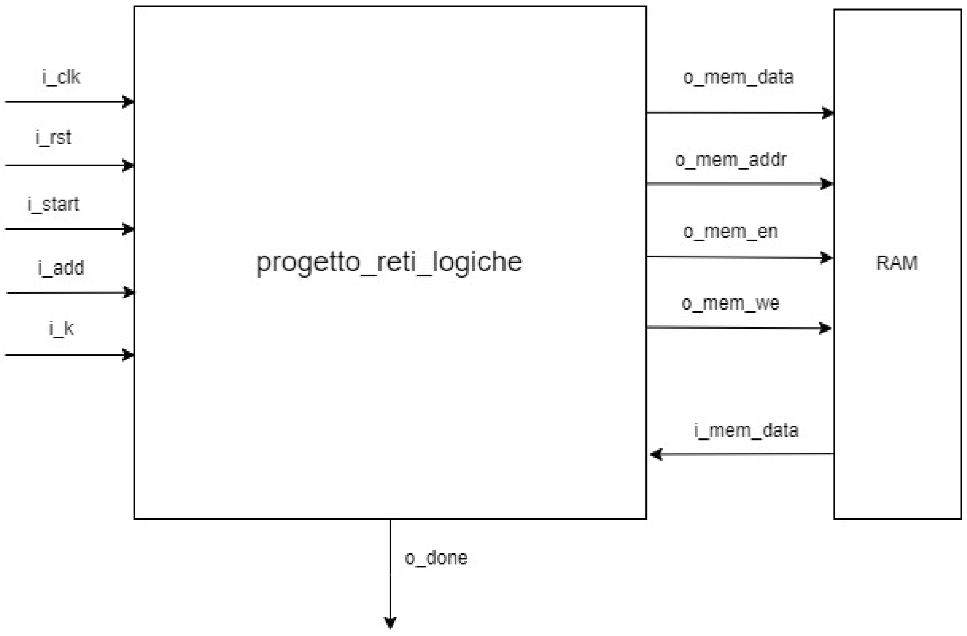


Figure 1.2: Components and signals

2 | Experimental Results

2.1. Testbench

The following section presents the results of the post-synthesis tests conducted for the most significant edge cases. Additionally, the first test executed (not reported here for conciseness) was crucial in identifying that the RAM required an additional clock cycle to output the data read from the specified address.

2.1.1. Reset without Completing the Previous Read Operation

At **1200 ns**, the reset signal (`rst`) is asserted, and the component is reinitialized (all signals are reset to zero). The system remains in this state until a start signal is received at **1850 ns**, at which point a new read address and a new word count are assigned.

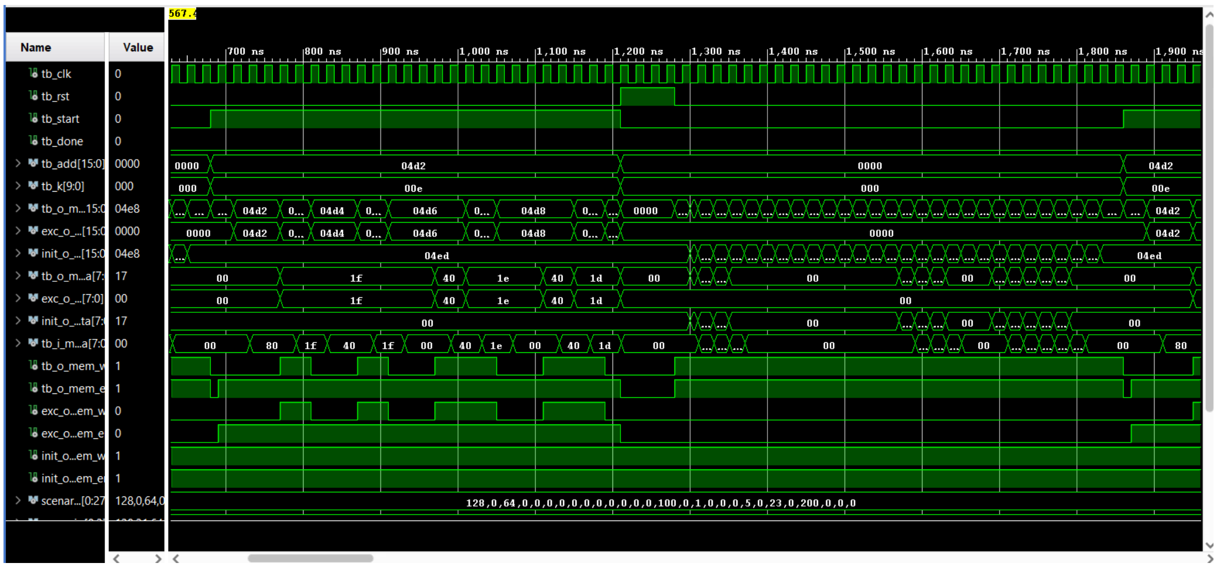


Figure 2.1: Reset without Completing the Previous Read Operation

2.1.2. Reading a Sequence Beginning with Multiple Zeros

When the sequence starts with consecutive zeros, the read address can be incremented by 2 to directly access the next value, as neither the word nor the credibility value needs to be replaced.

As shown in the example (where the sequence begins with two consecutive zeros), the component skips from address `0x04D2` to `0x04D4`, locating the first valid word at `0x04D6` (the read value in this case is `0x80`).

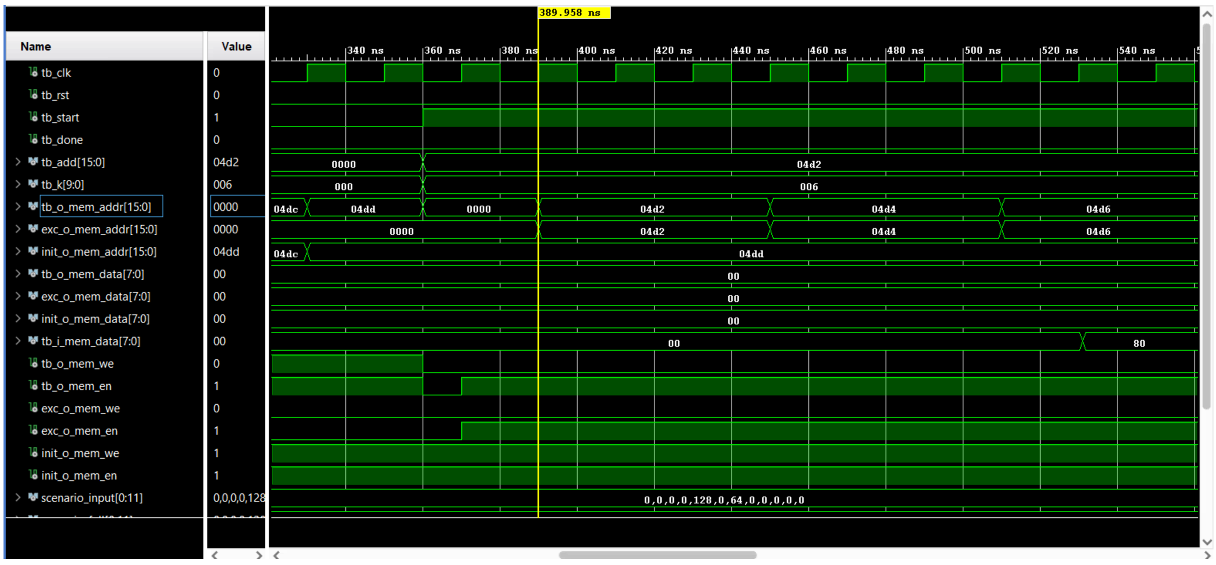


Figure 2.2: Reading a Sequence Beginning with Multiple Zeros

2.1.3. Credibility Value Decremented to Zero

It is observed that once the credibility value reaches zero (`o_mem_data` reaches zero at **5910 ns**), it remains at zero for all subsequent cycles.

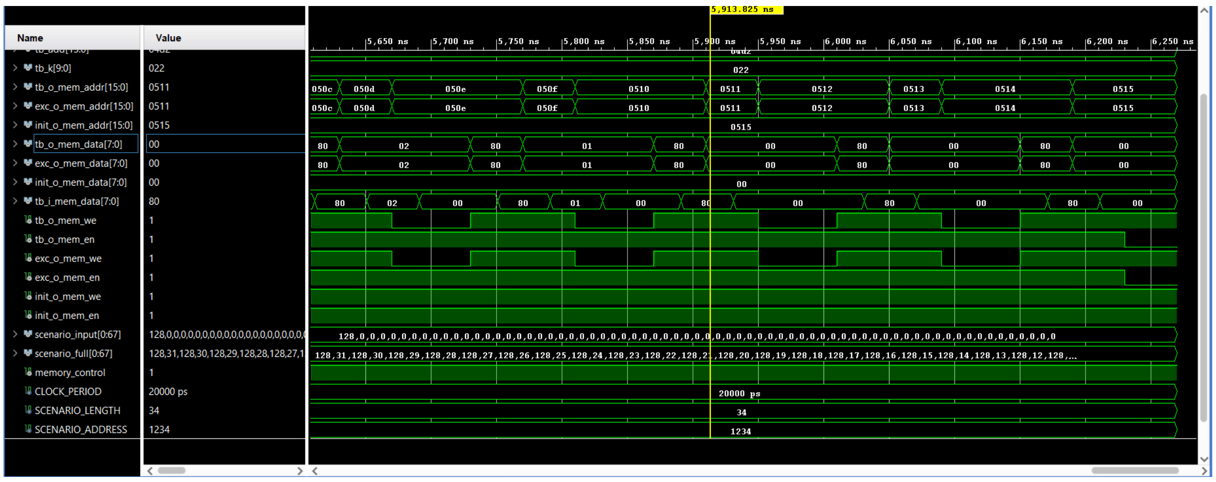


Figure 2.3: Credibility Value Decremented to Zero

2.1.4. k = 1 (Single Read Operation)

Even when the sequence consists of a **single word**, the component behaves correctly. Execution begins at **160 ns** when the start signal is received along with the first read address (`tb_add`) and the sequence length (`tb_k`). At **190 ns**, the read address is passed to the RAM, which at **210 ns** outputs the first word (0x80 in hexadecimal). At **250 ns**, the credibility value is passed to the RAM for writing, with `o_mem_we` transitioning to **1** (indicating a write operation).

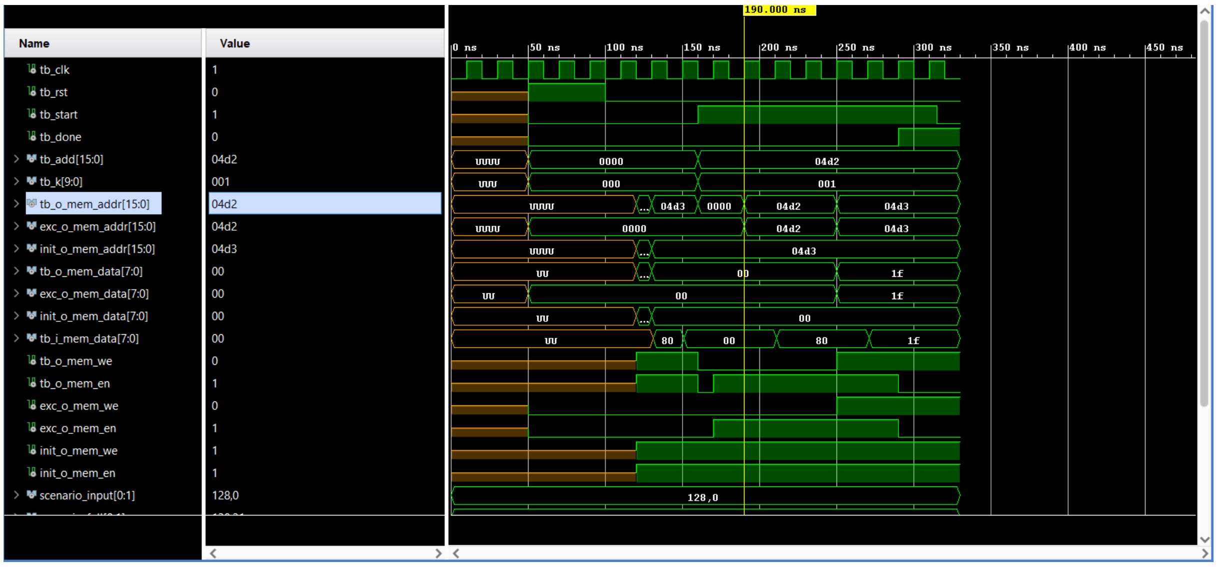


Figure 2.4: k = 1

2.1.5. New Start Signal without Reset

At **2820 ns**, a new start signal is received **without the reset signal** being asserted. Since signals are not reinitialized, a new execution starts, and the RAM is provided with the new read address and the new sequence length.

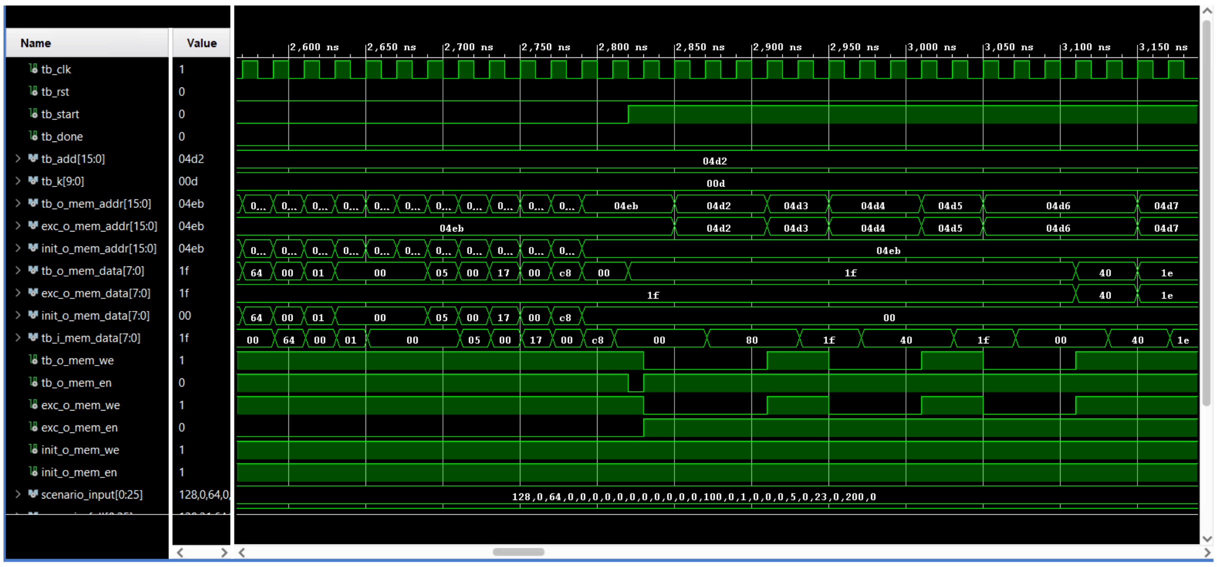
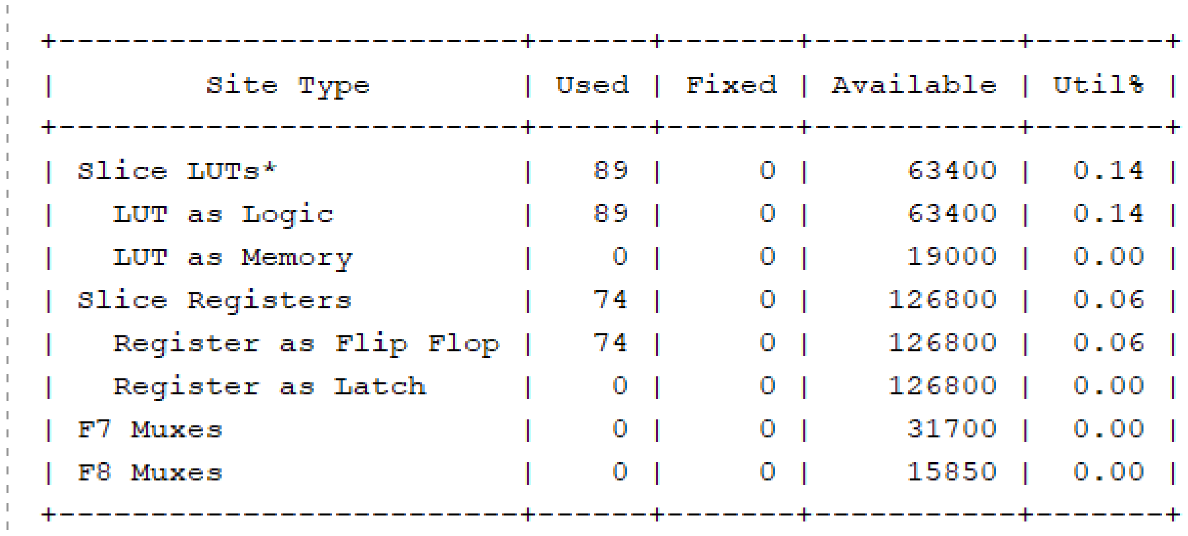


Figure 2.5: New Start Signal without Reset

2.2. Synthesis Report

2.2.1. Utilization Report

It is noteworthy that the number of generated flip-flops is very low and that no latches are used for the component implementation. This ensures that all elements operate synchronously with the clock, preventing situations where the output might change without a corresponding variation in the input.

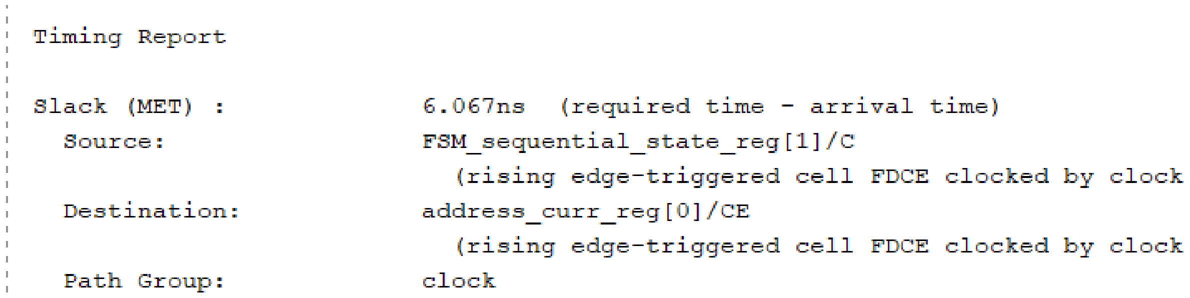


```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   89 |     0 |     63400 |  0.14 |
|   LUT as Logic          |   89 |     0 |     63400 |  0.14 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |   74 |     0 |    126800 |  0.06 |
|   Register as Flip Flop |   74 |     0 |    126800 |  0.06 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Figure 2.6: Utilization Report

2.2.2. Timing Report

A key observation is that the component utilizes only **14 ns** out of the available **20 ns** clock period, making it robust against signal delays.



```
Timing Report

Slack (MET) :             6.067ns  (required time - arrival time)
  Source:                 FSM_sequential_state_reg[1]/C
                            (rising edge-triggered cell FDCE clocked by clock
  Destination:            address_curr_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock
  Path Group:             clock
```

Figure 2.7: Timing Report

3 | Conclusions

The goal of this project was to design and implement a hardware component capable of replacing zero values stored in a RAM module and writing an associated credibility value.

The resulting component fully meets the given requirements and specifications, delivering excellent performance and a significant level of robustness. The `report_timing` analysis confirms that only **14 ns** out of **20 ns** of the available clock cycle are used, while the `report_utilization` reveals that no latches are generated and the number of flip-flops employed is minimized.

Designing the component as a single process managed by a state machine has allowed for a reduction in overall project complexity. This approach eliminated the need for additional components and significantly simplified the underlying logic.