POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Prova Finale

Progetto di Reti Logiche
Ingegneria Informatica

Studente: **Yanfeng Li**

Codice Persona: 10650552
Prof: Prof. Fabio Salice, Prof. Cassano Luca Maria
Anno Accademico: 2021-2022

Contents

Contents **i**

1 Introduction **1**

1.1 Project Purpose . . . 1
1.2 General Specifications . . . 1
1.3 Component Interface . . . 2
1.4 Data and Memory Description . . . 3
1.5 Examples . . . 3
1.6 Deduction from Specifications . . . 4

2 Architecture **5**

2.1 Datapath . . . 5
2.1.1 Reading Component . . . 6
2.1.2 Counting Component . . . 7
2.1.3 Output Production and Writing Component . . . 8
2.1.4 Address Management Component . . . 9
2.2 Control Unit . . . 10
2.2.1 State S0 . . . 11
2.2.2 State S1 . . . 11
2.2.3 State S2 . . . 12
2.2.4 State S3 . . . 12
2.2.5 State S4 . . . 12
2.2.6 State S44 . . . 12
2.2.7 State S45 . . . 12
2.2.8 State S5 . . . 12
2.2.9 State S6 . . . 12
2.2.10 State S7 . . . 12
2.2.11 State S8 . . . 13

3 Test Results **15**

3.1 Synthesis . . . 15

3.1.1 Utilization Report . . . 15

3.1.2 Timing Report . . . 15

3.2 Simulations . . . 16

3.2.1 Functionality Test . . . 16

3.2.2 Double Equal Test . . . 16

3.2.3 Maximum Length Sequence Test . . . 16

3.2.4 Minimum Length Sequence Test . . . 17

3.2.5 Multi-Flow Test with Reset . . . 17

4 Conclusion **19**

1 | Introduction

1.1. Project Purpose

The purpose of the project is to describe in VHDL and synthesize the HW component that implements the required specification, interfacing with a memory where the data is stored and where the final result will be written.

1.2. General Specifications

The component to be implemented must receive as input a continuous sequence of W words, each of 8 bits, and return as output a continuous sequence of Z words, each of 8 bits. Each input word is serialized, generating a continuous 1-bit stream U. A 1/2 convolutional code is applied to this stream (each bit is encoded with 2 bits) according to the scheme shown in the figure; this operation generates a continuous output stream Y. The stream Y is obtained by alternately concatenating the two output bits. Using the notation in the figure, the bit uk generates the bits p1k and p2k, which are then concatenated to generate a continuous stream yk (1-bit stream). The output sequence Z is the parallelization, into 8 bits, of the continuous stream yk.

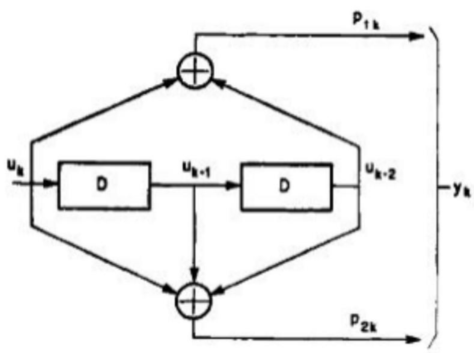

Figure 1.1: Convolutional encoder with a 1/2 transmission rate

1.3. Component Interface

The component to be described must have the following interface.

```
entity project_reti_logiche is
     port (
          i_clk     : in std_logic;
          i_rst     : in std_logic;
          i_start   : in std_logic;
          i_data    : in std_logic_vector(7 downto 0);
          o_address : out std_logic_vector(15 downto 0);
          o_done    : out std_logic;
          o_en      : out std_logic;
          o_we      : out std_logic;
          o_data    : out std_logic_vector (7 downto 0)
     );
end project_reti_logiche;
```

In particular:

- The module name must be `project_reti_logiche`.
- `i_clk` is the input CLOCK signal generated by the TestBench.
- `i_rst` is the RESET signal that initializes the machine, ready to receive the first START signal.
- `i_start` is the START signal generated by the TestBench.
- `i_data` is the input signal (vector) coming from the memory following a read request.
- `o_address` is the output signal (vector) that sends the address to the memory.
- `o_done` is the output signal that indicates the end of processing and the output data written to memory.
- `o_en` is the ENABLE signal to be sent to the memory to enable communication (both read and write).
- `o_we` is the WRITE ENABLE signal to be sent to the memory (=1) to enable writing. For reading from memory, it must be 0.
- `o_data` is the output signal (vector) from the component to the memory.

1.4. Data and Memory Description

Both input and output data will be written to a RAM memory with byte addressing and a 16-bit address bus. The sequence of bytes read as input is transformed into the bit stream U to be processed. The number of words W to be encoded is stored at address 0; the first byte of the sequence W is stored at address 1. The output stream Z must be stored starting from address 1000 (0000001111101000 in binary). The maximum size of the input sequence is 255 bytes, meaning 255 words to be processed; in memory cell 1, the value 255 (11111111 in binary) will be written.

1.5. Examples

The following sequence of numbers shows an example of the memory content at the end of a processing operation. The values represented here in decimal are stored in memory with their equivalent 8-bit unsigned binary encoding.

W: 10100010 01001011
Z: 11010001 11001101 11110111 11010010

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	2	\\ Byte lunghezza sequenza di ingresso
1	162	\\ primo Byte sequenza da codificare
2	75	
[...]		
1000	209	\\ primo Byte sequenza di uscita
1001	205	
1002	247	
1003	210	

Figure 1.2: Sequence of length 2

W: 01110000 10100100 00101101
Z: 00111001 10110000 11010001 11110111 00001101 00101000

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	3	\\ Byte lunghezza sequenza di ingresso
1	112	\\ primo Byte sequenza da codificare
2	164	
3	45	
[...]		
1000	57	\\ primo Byte sequenza di uscita
1001	176	
1002	209	
1003	247	
1004	13	
1005	40	

Figure 1.3: Sequence of length 3

1.6. Deduction from Specifications

P1(t) = U(t) xor U(t-2)
P2(t) = U(t) xor U(t-1) xor U(t-2)
Y(t) è la concatenazione tra P1(t) e P2(t)

Figure 1.4: Logical expressions

U(t-2)	U(t-1)	U(t)	P1(t)	P2(t)
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	0	1

Figure 1.5: Truth table

2 | Architecture

When the input signal **i_start** is set to 1, the developed component begins processing, transitioning from state **S0** to the first computation state. Once the computation is completed, after writing the result to memory, the component raises the **o_done** signal. The test bench responds by lowering **i_start**, and consequently, the component sets **o_done** to 0. The component then returns to state **S0**, waiting for the **i_start** signal to go high again. The component also includes a **i_rst** signal, which, along with the other signals listed in the previous chapter, led us to define a **FSM(D)**, a finite state machine with a data path, combining a standard FSM with typical sequential circuits. In the following sections, we will describe both the FSM and the sequential part of the machine that manages the registers used.

2.1. Datapath

The entire datapath is described by several basic components, including registers, multiplexers, adders, and subtractors. Overall, we can identify 4 groups by categorizing the components based on their functions: the reading component, the counting component, the writing component, and the addressing component. These will be described in detail below.

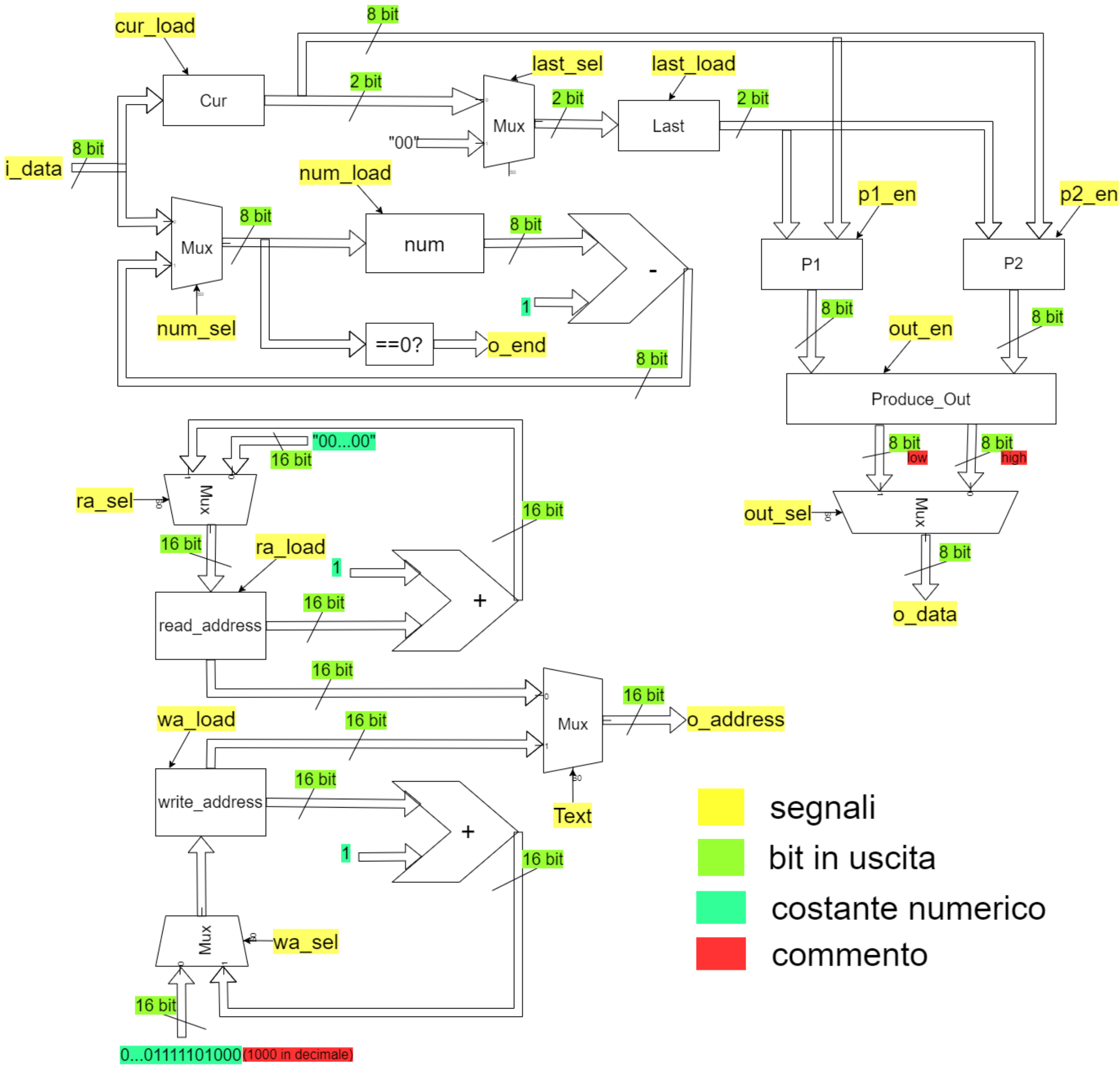

Figure 2.1: Datapath

2.1.1. Reading Component

The reading component consists of two registers and a multiplexer, specifically:

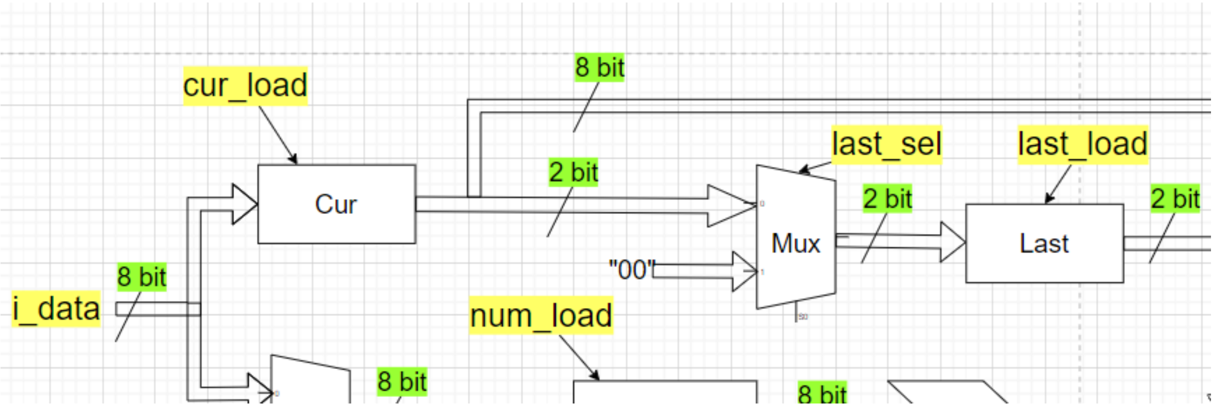

Figure 2.2: Reading component

- The **cur** register saves the input data at the current instant.
- The **last** register initially stores "00" in binary and later stores the last two bits of the content in the **cur** register.
- The **multiplexer**, with its output connected to the **last** register, is used to select the input value.

2.1.2. Counting Component

The counting component is built using a register, a multiplexer, a subtractor, and a comparator.

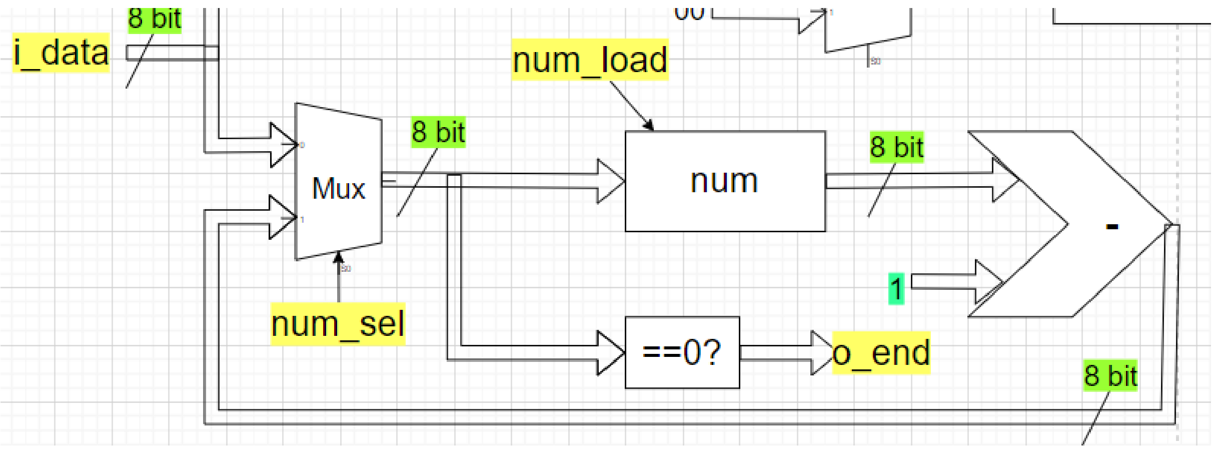

Figure 2.3: Counting component

- The **num** register stores the number of words to be read.
- The **multiplexer**, with its output connected to the **num** register, selects the input from either **i_data** or the subtractor.

- The **subtractor** decrements the count stored in the register by one and returns the result.
- The **comparator**, placed at the output of the multiplexer, changes the value of the **o_end** signal as soon as it detects that the multiplexer's output is 0.

2.1.3. Output Production and Writing Component

The output production and writing component consists of 3 registers with an embedded output calculation algorithm and a multiplexer.

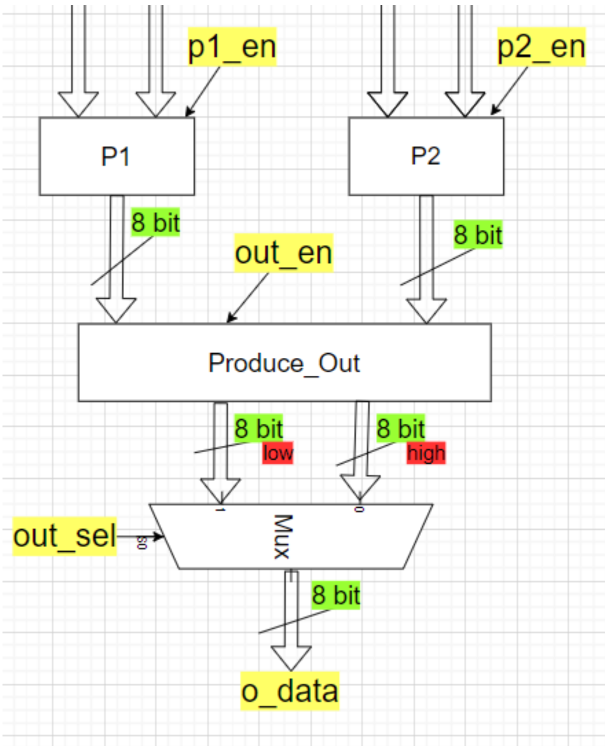

Figure 2.4: Output component

- The **P1** register takes the current input and the last two bits of the previous input and generates the result following the process specified in the requirements.

```
if(p1_en = '1') then
    temp(7) := o_cur(7) xor o_last(1);
    temp(6) := o_cur(6) xor o_last(0);
    for i in 5 downto 0 loop
        temp(i) := o_cur(i) xor o_cur(i+2);
    end loop;
    o_p1 <= temp;
```

Figure 2.5: P1(t) = U(t) xor U(t-2)

- The **P2** register has the same input as **P1** but generates a different output.

```
if(p2_en = '1') then
    temp(7) := (o_cur(7) xor o_last(0)) xor o_last(1);
    temp(6) := (o_cur(6) xor o_cur(7)) xor o_last(0);
    for i in 5 downto 0 loop
      temp(i) := (o_cur(i) xor o_cur(i+1)) xor o_cur(i+2);
    end loop;
    o_p2 <= temp;
```

Figure 2.6: P2(t) = U(t) xor U(t-1) xor U(t-2)

- The **Produce_out** register receives the outputs of **P1** and **P2**, each of 8 bits, and concatenates them according to the process specified in the requirements.

```
if(out_en = '1') then
    j := 15;
    for i in 7 downto 0 loop
        temp(j) := o_p1(i);
        temp(j-1) := o_p2(i);
        j := j-2;
    end loop;
    o_out <= temp;
```

Figure 2.7: OUT(t) = P1(t) concatenated with P2(t)

2.1.4. Address Management Component

The address management component is built using 3 multiplexers, 2 registers, and 2 adders. This component can be further divided into two parts: one for the write address and the other for the read address.

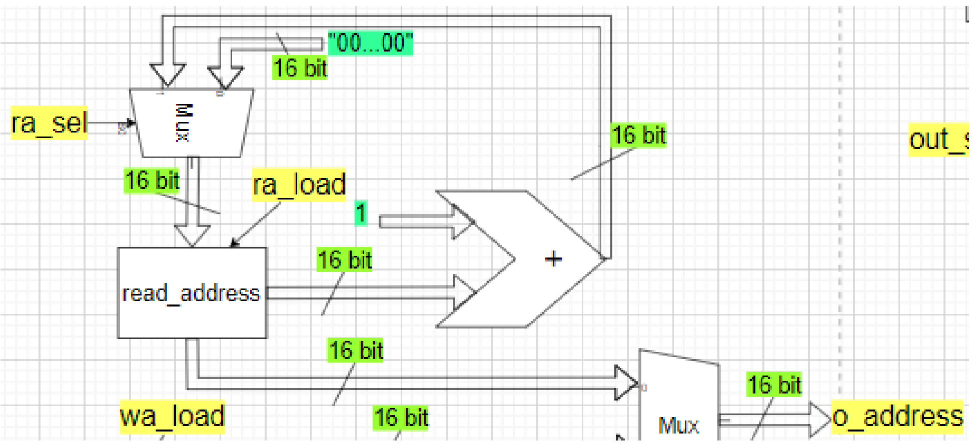

Figure 2.8: Read address register

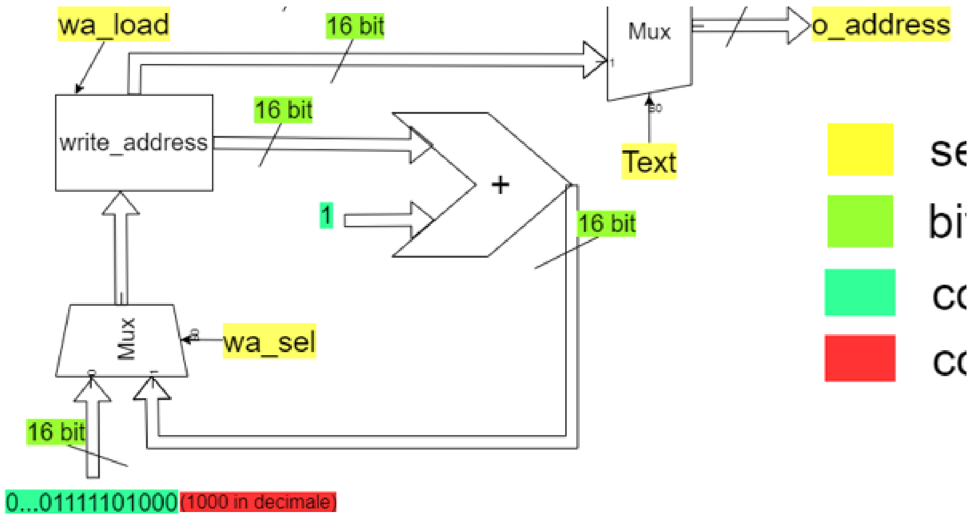

Figure 2.9: Write address register

- The **read/write_address** register stores the read/write address.
- The **adder** increments the read/write address by 1 and returns it to the read/write registers.
- The **multiplexer** selects between the initial address and subsequent addresses for the read/write registers.

2.2. Control Unit

The control unit manages the input and output control signals of the datapath for the proper functioning of the project. It is modeled as an 11-state Moore FSM.

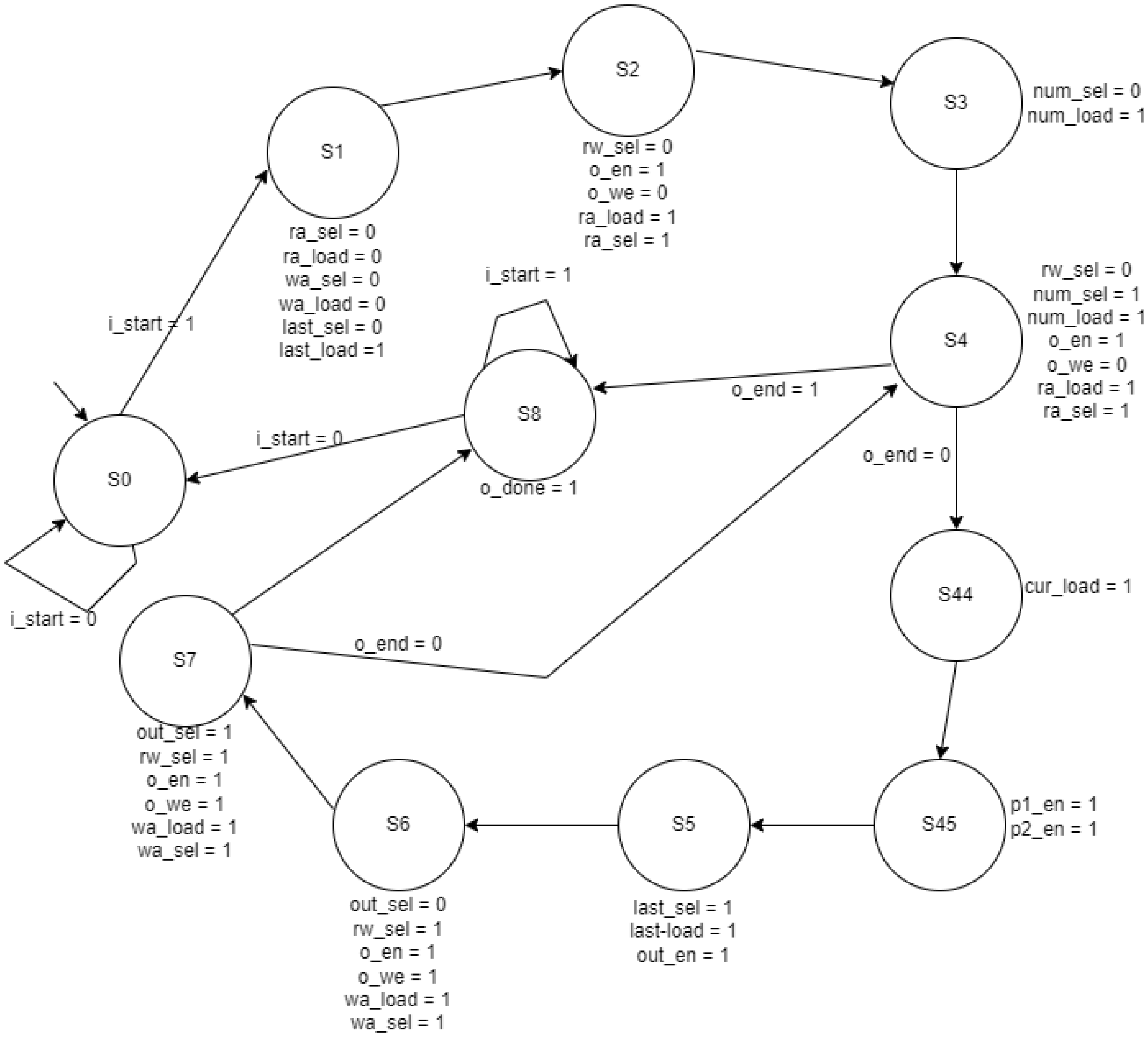

Figure 2.10: FSM

2.2.1. State S0

This is the initial state where the machine waits for the start signal.

2.2.2. State S1

In state **S1**, the initial values are initialized in the **read_address, write_address, last** registers.

2.2.3. State S2

Enables memory for reading and increments the read address. The value read from **i_data** will be ready in the next clock cycle.

2.2.4. State S3

Enables the **num** register for writing. The input value will be written and visible starting from the next clock cycle.

2.2.5. State S4

In state **S4**, memory is enabled for reading, the value in the **num** register is decremented, and the read address is incremented. Additionally, the transition to the next state depends on the **o_end** signal. If it is low, **S4** transitions to **S44**; otherwise, it transitions to **S8**.

2.2.6. State S44

Enables the **cur** register for writing. The input value (**i_data**) will be written and visible starting from the next clock cycle.

2.2.7. State S45

Enables the **P1, P2** registers for producing and writing intermediate results.

2.2.8. State S5

Enables the **produce_out** register to generate the result to be transmitted to memory and saves it by receiving data from the **P1, P2** registers. It also enables the **last** register for writing and shifts its input to the output of the **cur** register using a multiplexer.

2.2.9. State S6

In state **S6**, memory is enabled for writing, the write address is incremented, and the high part of the generated result is written to memory.

2.2.10. State S7

In state **S7**, memory is enabled for writing, the write address is incremented, and the low part of the generated result is written to memory. Additionally, the transition to the next state depends on the **o_end** signal. If it is low, **S7** transitions to **S4**; otherwise, it transitions to **S8**.

2.2.11. State S8

In state **S8**, the **o_end** signal is raised, and the machine remains in this state as long as the **i_start** signal remains high. Otherwise, it returns to state **S0**.

3 | Test Results

To ensure the correct functioning of the synthesized component, it is subjected to various tests to cover all possible cases the machine may encounter by traversing different paths. Below are the post-synthesis reports and the results of the tests performed.

3.1. Synthesis

Post-synthesis was performed using Vivado software, with the target FPGA being the Xilinx xc7a200tfbg484-1 device.

3.1.1. Utilization Report

The utilization report shows that the designed component is fully synthesizable, with no inferred latches.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	92	0	134600	0.07
LUT as Logic	92	0	134600	0.07
LUT as Memory	0	0	46200	0.00
Slice Registers	86	0	269200	0.03
Register as Flip Flop	86	0	269200	0.03
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figure 3.1: Utilization report

3.1.2. Timing Report

The Worst Negative Slack is approximately 96ns, indicating the time the machine remains at the completion of the clock cycle in the worst case. This is also below the specified requirement of 100ns.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	95.921 ns	Worst Hold Slack (WHS):	0.153 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	164	Total Number of Endpoints:	164	Total Number of Endpoints:	87

All user specified timing constraints are met.

Figure 3.2: Timing report, synthesis, 100ns clock

3.2. Simulations

Among the various tests performed, some were for verifying normal behavior and others for edge cases. All tests performed on the project yielded positive results in both behavioral and post-synthesis simulations.

3.2.1. Functionality Test

The test provides a wide range of input words to cover as many cases as possible for result production.

```
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:07 . Memory (MB): peak = 1699.035 ; gain = 0.000
run all
Failure: Simulation Ended! TEST PASSATO
Time: 1067600 ps  Iteration: 0  Process: /project_tb/test  File: D:/OneDrive - Politecnico di Milano/2021-2022/
```

Figure 3.3: Result

3.2.2. Double Equal Test

In this test bench, the same input is provided to the memory twice. The intention is to verify if the designed component can process identical sequences and produce identical results.

```
Failure: Simulation Ended! TEST PASSATO
Time: 5350 ns  Iteration: 0  Process: /project
```

Figure 3.4: Result

3.2.3. Maximum Length Sequence Test

Here, the project is subjected to the maximum input length of 255 words.

```
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:08 . Memory (MB): peak = 1719.422 ; gain = 0.000
run all
Failure: Simulation Ended! TEST PASSATO
Time: 154050100 ps  Iteration: 0  Process: /project_tb/test  File: D:/OneDrive - Politecnico di Milano/2021-2022/
```

Figure 3.5: Result

3.2.4. Minimum Length Sequence Test

The project must be able to process a null sequence. Here, 0 words are provided as input.

```
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:06 . Memory (MB): peak = 1726.328 ; gain = 0.000
run all
Failure: Simulation Ended! TEST PASSATO
Time: 1150100 ps  Iteration: 0  Process: /project_tb/test  File: D:/OneDrive - Politecnico di Milano/2021-2022
```

Figure 3.6: Result

3.2.5. Multi-Flow Test with Reset

This test checks the project's ability to process multiple input flows of different lengths, each preceded by a reset.

```
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:09 . Memory (MB): peak = 1719.422 ; gain = 0.000
run all
Failure: Simulation Ended! TEST PASSATO
Time: 11050100 ps  Iteration: 0  Process: /project_tb/test  File: D:/OneDrive - Politecnico di Milano/2021-2022
```

Figure 3.7: Result

4 | Conclusion

The component passed all the tests listed in the previous chapter according to the requirements. In this conclusion, possible optimizations for the already completed and functional project can be added. One idea is to merge the **cur, last** registers into a single register with 8+2 bits output. This way, at each input read, an 8-bit shift would suffice. Additionally, the output production process could be simplified into a single process.