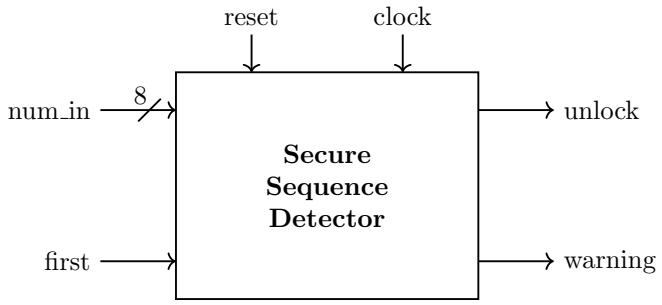
Secure Sequence

Leonardo Giovannoni

March 2024

1 Introduction



It is requested to design and implement a sequence recognizer in VHDL with the following characteristics:

The recognizer is synchronous and the sequence consists of 5 numbers, each represented by 8 bits. Specifically, the sequence is: 36, 19, 56, 101, 73. The first significant number must be accompanied by a *first* signal. The subsequent digits must be presented to the component at the cadence of one clock cycle each. The *first* signal should be high only during the presentation of the first number. If this condition is not met, the recognizer sets the *warning* signal to 1 and stops recognizing sequences until it is reset, for which there is a designated reset pin.

The recognition duration is always 5 clock cycles, even if the first number is incorrect. If the sequence is recognized after 5 clock cycles, the *unlock* pin is set to 1 for one clock cycle following the recognition of the last number. Otherwise, the *warning* pin is set to 1 for one clock cycle. If the recognizer fails to recognize the sequence three times consecutively, the *warning* pin is set to 1 constantly, and the recognizer ceases to function until it is reset.

Upon reset, both the *unlock* and *warning* pins are set to 0. The reset signal can be configured to be active high or active low, based on the designer's choice. Inputs of the block are:

- *num_in* (8-bit wide): This input receives the numbers that are part of the sequence to be detected.
- *first*: This input signal indicates if the current number being input is the first number of the sequence.
- *reset*: This input signal resets the state of the sequence detector.
- *clock*: This input signal provides the clock pulses to synchronize the operations of the sequence detector.

While its outputs are:

- *unlock*: This output signal is set to 1 if the sequence is correctly recognized.
- *warning*: This output signal is set to 1 if an error is detected during the sequence recognition or if the sequence fails to be recognized correctly.

2 Algorithm description

The algorithm can be implemented in various ways. One possible method is to maintain registers to track the state of errors or warnings. Alternatively, a pure state machine approach can be used. The pure state machine approach is advantageous because it can be formally verified and is conceptually straightforward for this problem. However, it requires tracking every possible state of the sequence recognition process, which results in numerous states that are difficult to handle in VHDL code.

Therefore, this approach will utilize some helper registers, such as:

- A counter to keep track of how many numbers in the sequence have been recognized.
- A counter to record the number of errors in each recognition attempt.
- A flag to monitor errors during the recognition of numbers.

The state machine will include only three states:

- *WaitingForFirst*: This is the default state, indicating that the system is waiting for the arrival of the first number. If the first bit is not set, no action is taken.
- *WaitingForNext*: This state is entered after receiving the first number and is for waiting for the subsequent numbers. In this state, the first signal should not be set to 1.
- *Error*: This state is an error state where the system waits to be reset.

3 Test description

The following is an example of a test case for the sequence recognizer module. Tests are created as yaml files and starting from them it will be generated by a program the relative testbench. Each entry in the array represents the inputs and expected outputs for a single clock cycle. The generic form of a test is the following

```
arr:
    - [reset@1T, number@1T, first@1T, [unlock@2T, warning@2T]]
    - [reset@2T, number@2T, first@2T, [unlock@3T, warning@3T]]
    .
    .
    .
```

Where:

- `reset` is a boolean signal (0 or 1) that indicates whether the system should be reset.
- `number` is the 8-bit input number to be processed by the sequence recognizer.
- `first` is a boolean signal (0 or 1) that indicates if the current number is the first number in the sequence.
- `[unlock, warning]` is an array containing two boolean values representing the expected outputs for the `unlock` and `warning` signals, respectively.

Below is presented an example of a test file.

```
arr:
  - [0, 36, 1, [0, 0]]
  - [0, 19, 0, [0, 0]]
  - [0, 56, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [1, 0]]

  - [0, 255, 0, [0, 0]]
  - [0, 255, 0, [0, 0]]

  - [0, 36, 1, [0, 0]]
  - [0, 19, 0, [0, 0]]
  - [0, 55, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [0, 1]]

  - [0, 255, 0, [0, 0]]

  - [0, 36, 1, [0, 0]]
```

Algorithm 1 Sequence Recognizer

```
Define constants:
    SEQUENCE = [36, 19, 56, 101, 73]
Define states:
    State = {WaitingForFirst, WaitingForNext, Error}
Function reset():
    current_state = WaitingForFirst
    error = False
    next_index = 0
    error_count = 0
    unlock = False
    warning = False
Function update(number, first):
if current_state == WaitingForFirst then
  if first then
    if SEQUENCE[0] ≠ number then
      error = True
    end if
    next_index, current_state = 1, WaitingForNext
  end if
  unlock, warning = False, False
else if current_state == WaitingForNext then
  if first then
    current_state, unlock, warning = Error, False, True
  else
    if SEQUENCE[next_index] ≠ number then
      error = True
    end if
    next_index += 1
    if next_index == len(SEQUENCE) then
      next_index = 0
      if error then
        error_count += 1
      end if
      if error_count ≥ 3 then
        current_state = Error
      else
        current_state = WaitingForFirst
      end if
      unlock, warning, error = not error, error, False
    else
      unlock, warning = False, False
    end if
  end if
else if current_state == Error then
  unlock, warning = False, True
end if
```

```
  - [0, 19, 0, [0, 0]]
  - [0, 56, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [1, 0]]

  - [0, 255, 0, [0, 0]]
```

Explanation:

- Each element in the array corresponds to one clock cycle.
- The first value is the *reset* signal (0 = no reset, 1 = reset).
- The second value is the number input, representing the current number in the sequence.
- The third value is the *first* signal (0 = not first number, 1 = first number of the sequence).
- The fourth element is an array with two values:
  - The first value is the expected *unlock* output (0 = no unlock, 1 = unlock).
  - The second value is the expected *warning* output (0 = no warning, 1 = warning).

This format allows for a more simple way to create a test without have to deal with VHDL syntax and boilerplate, focusing on the data. Once that all testbenches will be generated, than it will be run a command into a justfile that will test the code with all testbenches. If an error it is found, it will be printed by GHDL to screen. As an example of the transpilation we report the yaml code

```
arr:
  - [0, 36, 1, [0, 0]]
  - [1, 0, 0, [0, 0]]
```

And the equivalent VHDL code generated

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE std.env.stop;

ENTITY SequenceRecognizer_tb7 IS
END SequenceRecognizer_tb7;

ARCHITECTURE behavior OF SequenceRecognizer_tb7 IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT SequenceRecognizer
        PORT (
            clk : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            number : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            first : IN STD_LOGIC;
            unlock : OUT STD_LOGIC;
            warning : OUT STD_LOGIC
        );
    END COMPONENT;

    -- Inputs
    SIGNAL clk : STD_LOGIC := '0';
    SIGNAL reset : STD_LOGIC := '0';
    SIGNAL number : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
```

```
    SIGNAL first : STD_LOGIC := '0';

    -- Outputs
    SIGNAL unlock : STD_LOGIC;
    SIGNAL warning : STD_LOGIC;

    -- Clock period definition
    CONSTANT clk_period : TIME := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut : SequenceRecognizer PORT MAP(
        clk => clk,
        reset => reset,
        number => number,
        first => first,
        unlock => unlock,
        warning => warning
    );

    -- Clock process definitions
    clk_process : PROCESS
    BEGIN
        clk <= '0';
        WAIT FOR clk_period/2;
        clk <= '1';
        WAIT FOR clk_period/2;
    END PROCESS;

    -- Stimulus process
    stim_proc : PROCESS
    BEGIN
    reset <= '1';
    WAIT FOR CLK_PERIOD;
        reset <= '0';
        number <= x"24";
        first <= '1';
        WAIT FOR CLK_PERIOD;
        ASSERT unlock = '0' REPORT "Unlock error" SEVERITY error;
        ASSERT warning = '0' REPORT "Warning error" SEVERITY error;
        reset <= '1';
        number <= x"00";
        first <= '0';
        WAIT FOR CLK_PERIOD;
        ASSERT unlock = '0' REPORT "Unlock error" SEVERITY error;
        ASSERT warning = '0' REPORT "Warning error" SEVERITY error;
        STOP;
END process;
END behavior;
```

3.1 Test plan

Using just YAML as input simplifies a lot writing testbenches, so we could perform a lot more tests in a faster way. Tests performed are the following:

Based on the provided VHDL testbenches and the project requirements outlined in the uploaded document, here are suggested names for each test that align with typical naming conventions in hardware verification:

1. **Basic Sequence Recognition Test**: Ensures the SequenceRecognizer correctly identifies the sequence 36, 19, 56, 101, 73 with appropriate *first* and *reset* signals.
2. **Sequence Recognition with Reset Test**: Tests the SequenceRecognizer functionality under normal conditions, but includes a reset during the sequence to check if the module resets correctly and starts the sequence recognition again.
3. **Multiple Sequence Recognition Test**: Verifies if the SequenceRecognizer can handle multiple sequences in succession, ensuring the unlock and warning signals behave as expected across multiple sequences.
4. **Incorrect First Signal Test**: Checks the behavior of the SequenceRecognizer when the *first* signal is not set correctly, expecting the warning signal to be asserted.
5. **Immediate Reset Test**: Tests the immediate reset functionality to ensure both *unlock* and *warning* signals are set to 0 immediately after the reset signal.
6. **Warning Signal Lock Test**: Ensures that the warning signal stays asserted after three consecutive sequence recognition failures and that the recognizer stops functioning until a reset is applied.
7. **Sequence with Early Warning Test**: Tests for correct assertion of the warning signal when the sequence starts with *first*, followed by an incorrect sequence.
8. **Repeated Incorrect Sequence Test**: Repeatedly inputs an incorrect sequence to verify if the warning signal remains asserted after three failures.
9. **Stress Test with Multiple Sequences**: Provides multiple sequences, some correct and some incorrect, to ensure the module handles stress conditions and maintains correct behavior.
10. **Continuous Operation Test**: Runs a prolonged test to ensure the module's stability over an extended period with sequences provided continuously.
11. **Edge Case Test**: Tests specific edge cases such as maximum and minimum possible values for the sequence numbers, to verify robustness.

4 VHDL implementation

Let's break down the VHDL code into smaller chunks and explain each part.

```
ENTITY SequenceRecognizer IS
    PORT (
        clk : IN STD_LOGIC;
        reset : IN STD_LOGIC;
        number : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8-bit input number
        first : IN STD_LOGIC; -- Boolean flag for first number
        unlock : OUT STD_LOGIC; -- Unlock output
        warning : OUT STD_LOGIC -- Warning output
    );
END SequenceRecognizer;
```

This declares the entity *SequenceRecognizer* with inputs for clock (*clk*), reset (*reset*), an 8-bit number (*number*), and a boolean flag (*first*). It also declares outputs for unlock (*unlock*) and warning (*warning*).

```
ARCHITECTURE Behavioral OF SequenceRecognizer IS
    TYPE State_Type IS (WaitingForFirst, WaitingForNext, ErrorState);
    SIGNAL current_state : State_Type;
    SIGNAL next_index : INTEGER RANGE 0 TO 4;
    SIGNAL error : STD_LOGIC;
    SIGNAL error_count : INTEGER RANGE 0 TO 100; -- Arbitrarily large enough range

    TYPE Array_Type IS ARRAY(0 TO 4) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
    CONSTANT CORRECT_SEQUENCE : Array_Type := (
        x"24", x"13", x"38", x"65", x"49"
    );
```

This defines the architecture *Behavioral* for the entity. It declares: - *State_Type*: An enumerated type for FSM states (*WaitingForFirst*, *WaitingForNext*, *ErrorState*). - Internal signals: *current_state*, *next_index*, *error*, *error_count*. - *Array_Type*: An array type for storing the correct sequence of numbers. - *CORRECT_SEQUENCE*: A constant array holding the correct sequence to unlock.

```
BEGIN
    PROCESS (clk, reset)
    BEGIN
        IF reset = '1' THEN
            current_state <= WaitingForFirst;
            next_index <= 0;
            error <= '0';
            error_count <= 0;
            unlock <= '0';
            warning <= '0';
```

The main process starts. If *reset* is high, it initializes the state machine to *WaitingForFirst*, resets the *next_index*, *error*, *error_count*, and output signals (*unlock* and *warning*).

```
        ELSIF rising_edge(clk) THEN
            CASE current_state IS
                WHEN WaitingForFirst =>
                    IF first = '1' THEN
                        IF CORRECT_SEQUENCE(0) /= number THEN
                            error <= '1';
                        END IF;
                        next_index <= 1;
                        current_state <= WaitingForNext;
                    END IF;
                    unlock <= '0';
                    warning <= '0';
```

When the clock rises, the state machine checks the current state. In *WaitingForFirst*, if *first* is high, it checks if the input number matches the first number in *CORRECT_SEQUENCE*. If it doesn't, it sets *error*. Then it moves to the *WaitingForNext* state and increments *next_index*.

```
                WHEN WaitingForNext =>
                    IF first = '1' THEN
                        current_state <= ErrorState;
                        warning <= '1';
                    ELSE
                        IF CORRECT_SEQUENCE(next_index) /= number THEN
                            error <= '1';
                        END IF;
                        IF next_index = 4 THEN
                            next_index <= 0;
                            IF error = '1' THEN
                                error_count <= error_count + 1;
                            END IF;
                            IF error = '1' AND error_count >= 2 THEN
                                current_state <= ErrorState;
                            ELSE
                                current_state <= WaitingForFirst;
                            END IF;
```

```
                    IF error = '0' THEN
                        unlock <= '1';
                        warning <= '0';
                    ELSE
                        unlock <= '0';
                        warning <= '1';
                    END IF;
                    error <= '0';
                ELSE
                    next_index <= next_index + 1;
                END IF;

            END IF;
```

In *WaitingForNext*, if *first* is high again, it transitions to *ErrorState* and sets *warning*. Otherwise, it checks if the input number matches the expected number in *CORRECT_SEQUENCE*. If it doesn't match, *error* is set.

If *next_index* reaches 4, it checks if there were any errors. If there are errors and the error count is 2 or more, it transitions to *ErrorState*. Otherwise, it returns to *WaitingForFirst*.

If no errors, it sets *unlock* high; otherwise, it sets *warning* high. It resets *error* and increments *next_index* for the next number.

```
            WHEN ErrorState =>
                unlock <= '0';
                warning <= '1';

        END CASE;
    END IF;
END PROCESS;
END Behavioral;
```

In *ErrorState*, it keeps *unlock* low and *warning* high, indicating an error condition.

4.1 Test Execution using Justfile

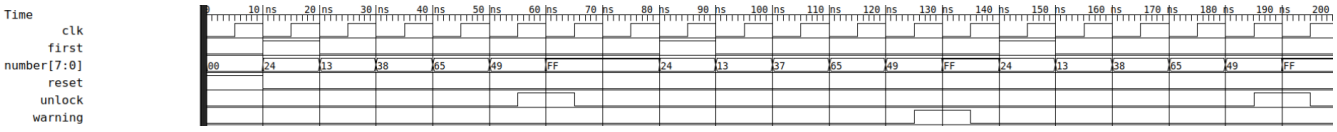
To facilitate the testing process, a *justfile* is used to manage the compilation, elaboration, and execution of the testbenches. The *justfile* provides a set of commands to streamline these tasks, ensuring that the VHDL code is analyzed and tested efficiently. Here is a high-level description of the *justfile* commands:

- **analyze test**: Compiles the VHDL source files in the `src` directory and testbenches in the `testbench` directory.
- **elaborate test**: Elaborates each testbench in the `testbench` directory to prepare for simulation.
- **run test**: Runs the compiled testbenches in the `testbench` directory and generates the output waveforms in the `build/dump` directory.
- **generate_test**: Generates testbenches from the YAML test definitions in the `test` directory using a script in the `test_generator` directory and places them in the `testbench` directory.
- **start_gtkwave**: Launches GTKWave to view the simulation results in the `build/dump` directory.
- **test**: Cleans, compiles, elaborates, and runs all testbenches in the `testbench` directory.
- **test-one** *test_file_name*: Cleans, compiles, elaborates, and runs a specific testbench in the `testbench` directory.
- **clean**: Cleans up the build directories.

The use of this *Justfile* automates the testing process, making it easier to verify the functionality of the sequence recognizer module. If any errors are found during the simulation, they will be printed to the screen by GHDL, allowing for quick identification and debugging.

4.2 Simulation

We don't really need to see graphically what is happening since we are using asserts to verify correctness of output, which can be certainly more precise than visual checking. However, for completeness, we report what a simulation will plot.



In the picture we can see the simulation corresponding to the following test:

```yaml
arr:
  - [0, 36, 1, [0, 0]]
  - [0, 19, 0, [0, 0]]
  - [0, 56, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [1, 0]]

  - [0, 255, 0, [0, 0]]
  - [0, 255, 0, [0, 0]]

  - [0, 36, 1, [0, 0]]
  - [0, 19, 0, [0, 0]]
  - [0, 55, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [0, 1]]

  - [0, 255, 0, [0, 0]]

  - [0, 36, 1, [0, 0]]
  - [0, 19, 0, [0, 0]]
  - [0, 56, 0, [0, 0]]
  - [0, 101, 0, [0, 0]]
  - [0, 73, 0, [1, 0]]

  - [0, 255, 0, [0, 0]]
```

The simulation waveform corresponds to the YAML test case, showing the correct sequence of inputs and expected outputs. The `unlock` and `warning` signals are asserted correctly based on the input sequence, indicating the recognizer's proper functionality. We presented the waveform just for completeness, but we didn't need to show it since each output is asserted to be the correct one.

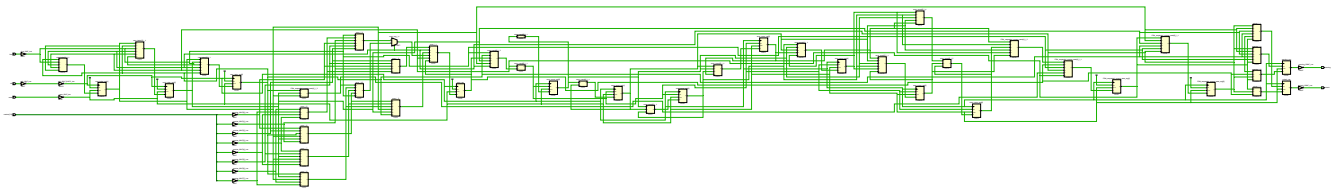
5 Synthesis

The logic synthesis was carried out with Xilinx Vivado 2023.2 without any issues2The synthesis was performed using a Xilinx Zynq-7000 general purpose board (xc7z010clg400-1) as a target SoC with the clock constraint of 8ms (125MHz). After the synthesis we get 1 warning

```
[Synth 8-7080] Parallel synthesis criteria is not met
```

that tells us that project size is not sufficiently large to benefit for parallelization on a multi-core cpu.

Schematic In figure we can see the schematic which results from the synthesis.



Timing The timing summary in the implementation stage is as follows:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.184 ns	Worst Hold Slack (WHS):	0.142 ns	Worst Pulse Width Slack (WPWS):	3.500 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:	24	Total Number of Endpoints:	24	Total Number of Endpoints:	16

All user specified timing constraints are met.

In the summary, the following parameters can be found:

- **Worst Negative Slack**: indicates the worst negative timing discrepancy among signal paths in the circuit. A negative discrepancy indicates insufficient delay to meet timing requirements.
- **Total Negative Slack**: represents the overall sum of negative timing discrepancies across all signal paths in the circuit.
- **Worst Hold Slack**: measures the worst timing discrepancy concerning hold requirements among critical signal paths. It indicates whether the hold time between signals is sufficient.
- **Total Hold Slack**: is the total sum of hold timing discrepancies across all signal paths.
- **Worst Pulse Width Slack**: indicates the worst timing discrepancy in pulse width, crucial for periodic signals like clocks.
- **Total Worst Pulse Width**: represents the total sum of timing discrepancies in pulse width across all signal paths in the circuit.

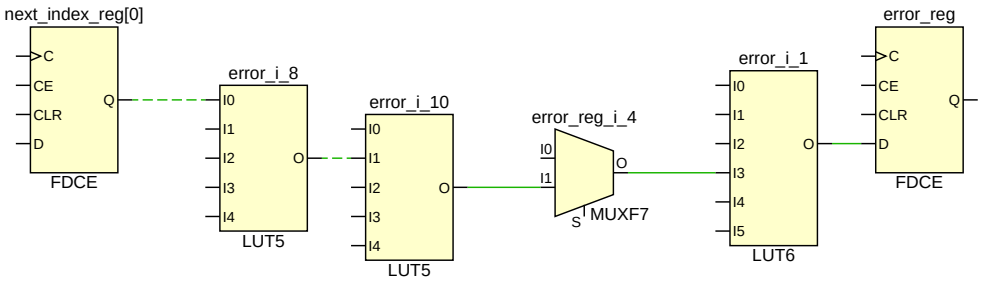
We can calculate the maximum frequency as:

$$f_{max} = \frac{1}{T_{clk} - \text{WNS}} = \frac{1}{(8 - 4.184)10^{-9}} \approx 262\text{MHz}$$

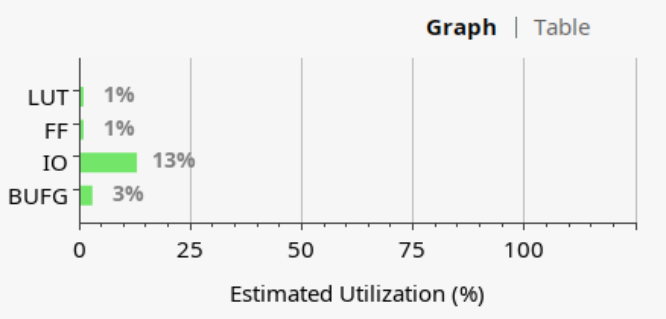
Critical path The WSN is determined by the critical path of the architecture which is shown in the following table:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	4.184	4	5	12	next_index_reg[0]/C	error_reg/D	3.665	1.446	2.219	8.0	clk	clk		0.035
Path 2	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[0]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 3	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[1]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 4	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[2]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 5	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[3]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 6	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[4]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 7	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[5]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 8	4.932	2	3	12	next_index_reg[0]/C	error_count_reg[6]/CE	2.686	0.875	1.811	8.0	clk	clk		0.035
Path 9	5.381	1	2	12	FSM_sequential_current_state_reg[0]/C	unlock_reg/CE	2.237	0.751	1.486	8.0	clk	clk		0.035
Path 10	5.381	1	2	12	FSM_sequential_current_state_reg[0]/C	warning_reg/CE	2.237	0.751	1.486	8.0	clk	clk		0.035

Visualizing the schematic of the critical path reveals that this section of the circuit that exhibits the longest timing delays due to its combinatory logic elements



Utilization



In the picture we can see the estimated utilization of the module:

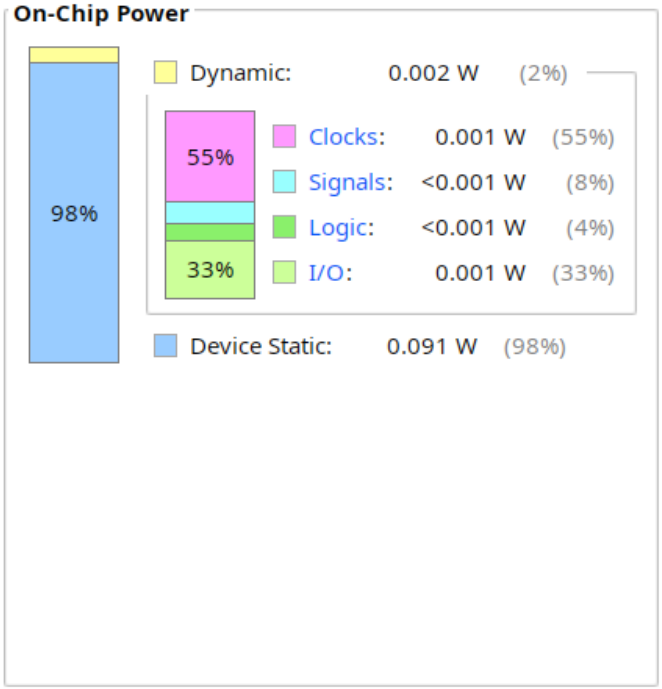
1. **LUT** (Look-Up Tables) - 1% utilization: The design utilizes a very small fraction (1%) of the available LUTs on the FPGA. This indicates that the combinational logic required by the *SequenceRecognizer* is minimal and efficient.

2. **FF** (Flip-Flops) - 1% utilization: Similar to the LUTs, the flip-flop utilization is also very low (1%). This suggests that the sequential logic (registers, state machines, etc.) in the design is quite limited and does not demand many resources.

3. **IO** (Input/Output) - 13% utilization: The IO utilization is at 13%, which is relatively higher compared to LUTs and FFs. This indicates that the design has a moderate number of input and output signals.

4. **BUFG** (Global Clock Buffers) - 3% utilization: The design uses 3% of the available global clock buffers. Global clock buffers are used to distribute clock signals efficiently across the FPGA. A 3% utilization indicates that the design has some clock signals that need to be distributed across different parts of the FPGA.

Power consumption The dynamic power consumption is very low compared with static power consumption. A motivation could be that if the design is relatively small, the dynamic power consumption will naturally be low. Smaller designs use fewer resources, which in turn have fewer transitions, leading to lower dynamic power consumption. Another motivation could be the low frequency of the clock.



6 Conclusions

The project successfully achieved its goals with a highly efficient design in terms of resource utilization and power consumption. The comprehensive testing plan ensured robust functionality and reliable performance. The use of YAML for test definitions and *Justfile* for automation significantly streamlined the testing and verification process, enhancing the project's efficiency. The design is ready for deployment on a Xilinx Zynq-7000 FPGA, with potential applications in secure digital systems requiring sequence-based unlocking mechanisms.