OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Sequential Circuits in VHDL: Example of the Sequence Recognizer

Marta Bagatin, marta.bagatin@unipd.it

Degree Course in Information Engineering

Academic Year 2023-2024

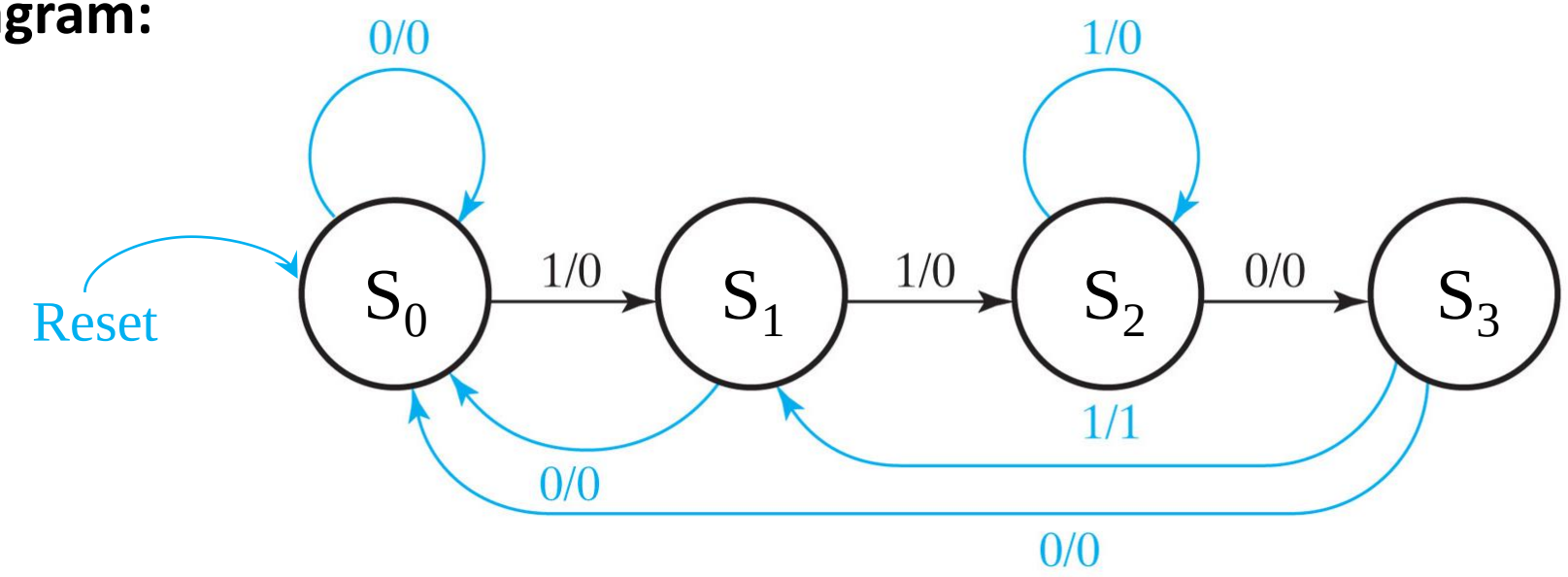
Purpose of the Lesson

- Learn how to describe a **synchronous sequential circuit** in VHDL
- We will see a possible design and testbench of the sequence recognizer (same example we used to study the synthesis of sequential circuits)

Sequence Recognizer

- **Input (1 bit): X Output (1 bit): Z Recognizes the sequence 1101**
  - Reset brings the system to the state «no symbol of the sequence was recognized»
  - Z = 1 if the previous 3 inputs were ‘110’ and the current input is ‘1’
  - Z = 0 otherwise
- The system uses D-type flip-flops with asynchronous reset

State diagram:

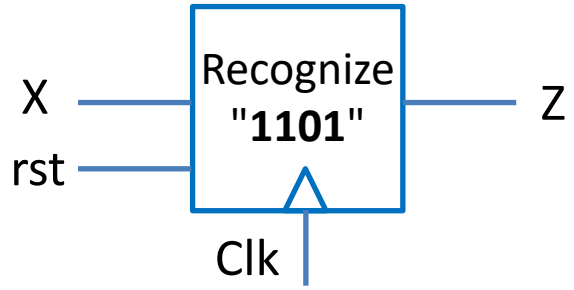


VHDL Description of a Sequential Circuit

- We will describe the circuit with three distinct processes
  - **Process 1**: describes the **update of the state** of the system
  - **Process 2**: calculates the **next state** as a function of present state and input
  - **Process 3**: calculates the **output** as a function of present state and input
- Process 1 implements the sequential part of the circuit. Processes 2 and 3 describe purely combinational logic
- Recall that multiple processes in the architecture run in parallel
- We will see a **behavioral description** of the circuit

Sequence Recognizer: VHDL (1/4)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity seq_rec is
   port (clk, rst, X : in std_logic;
                   Z : out std_logic);
end seq_rec;

architecture beh of seq_rec is
   type state_type is (S0, S1, S2, S3);
   signal state, next_state: state_type;
begin
-- Process 1: Updates state and implements asynchronous reset
   state_register: process (rst, clk)
   begin
      if (rst = '1') then
         state <= S0;
      elsif (clk'event and clk = '1') then
         state <= next_state;
      end if;
   end process;
```



Keywork **type** is used to define a new type of data

State is stored in D-flip-flops with asynchronous reset. Upon reset, the system is brought to state S0

Type

Syntax:

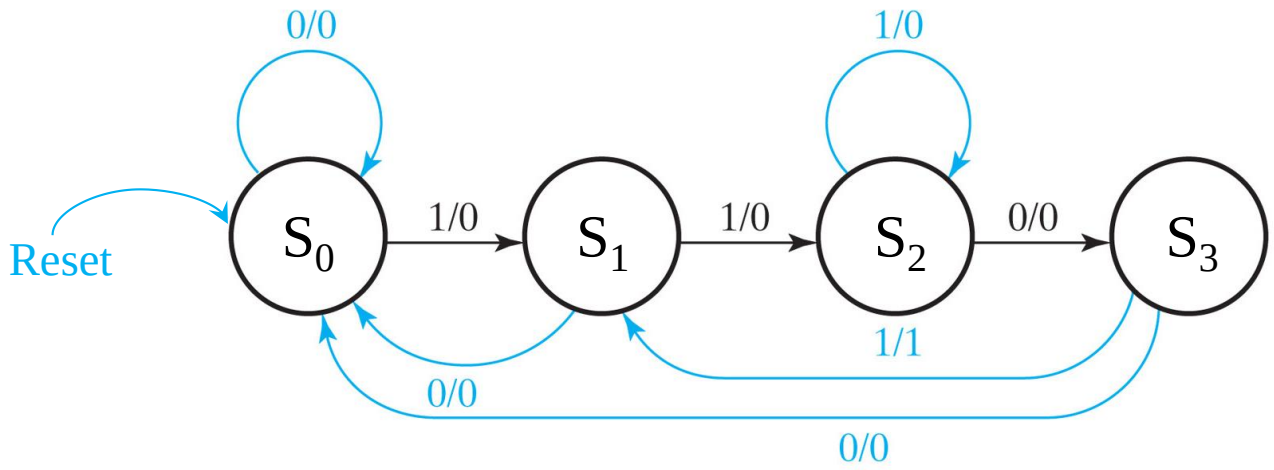
```
type type_name is (value1, value2, value3, ...);
```

- The keyword `type` defines new data types
- For example, we can define a new type by listing all its values between brackets and separated by commas (enumeration). This is useful to define signals that can take a finite number of values (identified by names, not by numbers, to improve the readability of the code)
- Note: the type std_logic we use for logic signals is an enumeration type, which can take 9 distinct values: '0', '1', 'X', '-', 'U', 'Z', 'W', 'L', 'H'
- Once a type has been declared within a design, signals of that type can be used with the usual declaration:

```
signal signal_name: type_name;
```

Sequence Recognizer: VHDL (2/4)

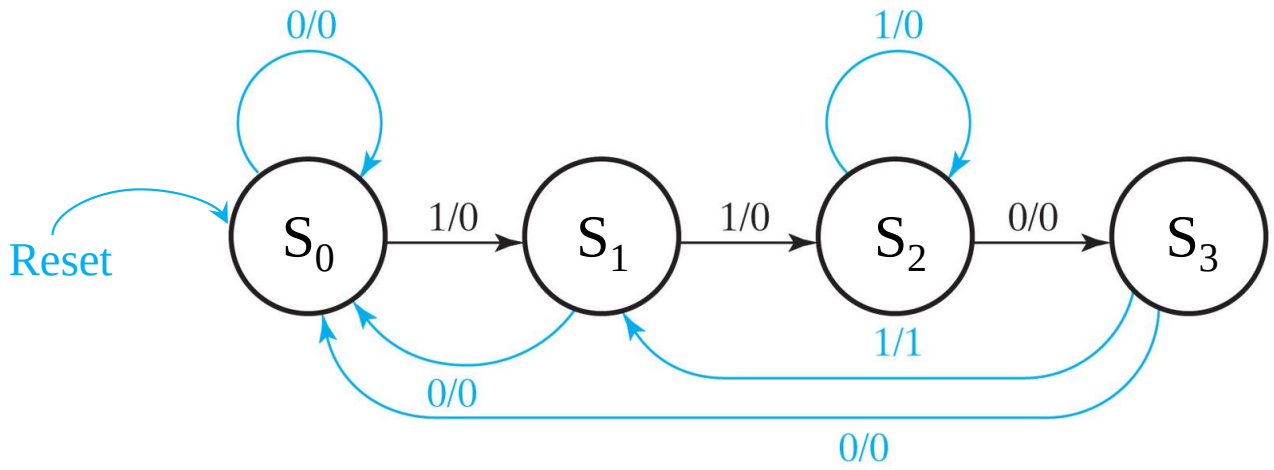
```
-- Process 2: Computes the next state as a combinational function of input and
present state values

    next_state_comb: process (X, state)
    begin

        case state is
            when S0 =>
                if X = '0' then
                    next_state <= S0;
                else
                    next_state <= S1;
                end if;

            when S1 =>
                if X = '0' then
                    next_state <= S0;
                else
                    next_state <= S2;
                end if;
```



Next state is defined with a **case statement** (sequential brother of with-select), based on present state and input

Sequence Recognizer: VHDL (3/4)

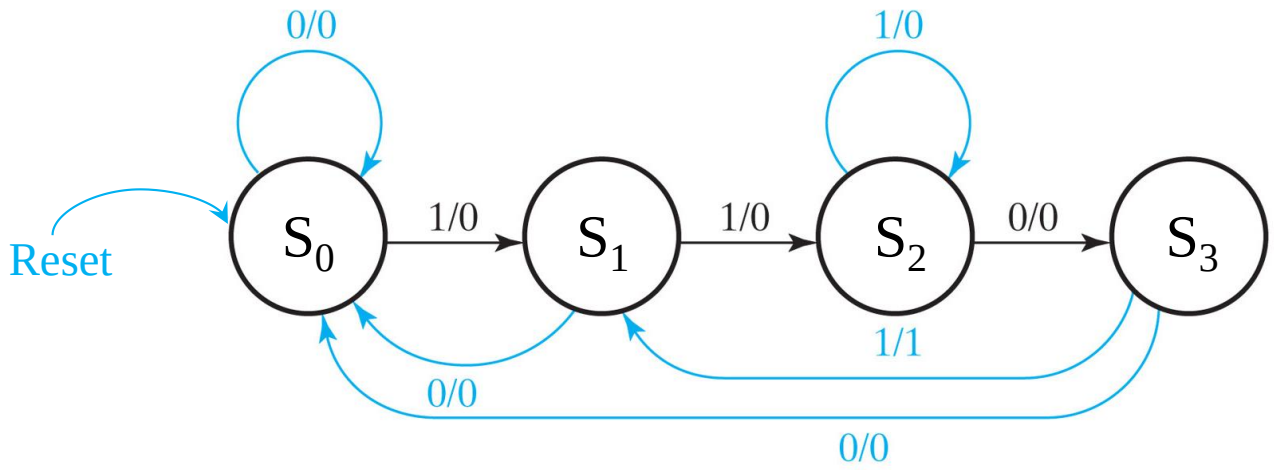
```
-- Process 2 (continued)

        when S2 =>
            if X='0' then
                next_state <= S3;
            else
                next_state <= S2;
            end if;

        when S3 =>
            if X='0' then
                next_state <= S0;
            else
                next_state <= S1;
            end if;
    end case;

end process;
```



We don't need to include 'when others', since we have already enumerated all the possible values that the state type can take

Sequence Recognizer: VHDL (4/4)

```
-- Process 3: calculates the output as combinational function of input and state
   output_comb: process (X, state)
   begin
      case state is
         when S0 =>
            Z <= '0';
         when S1 =>
            Z <= '0';
         when S2 =>
            Z <= '0';
         when S3 =>
            if X='0' then
               Z <= '0';
            else
               Z <= '1';
            end if;
      end case;
   end process;
end beh;
```



Output is defined with another **case statement**, based on the present state and input values (Mealy model)

Remarks

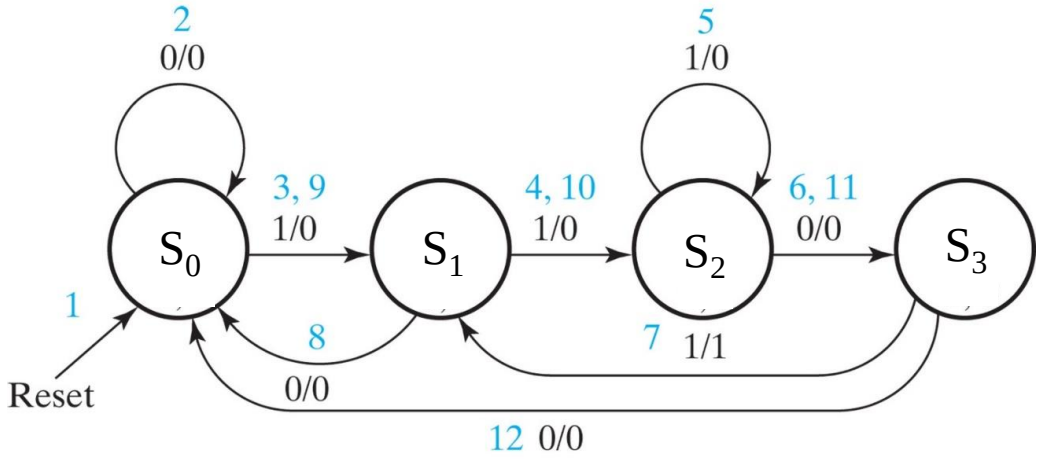
- The **three processes** are **performed in parallel** and they **interact with each other**, having signals in common
- We used a **behavioral description**: we have not suggested in any way how to implement the system (how to encode the states, how to implement the combinational circuits that calculate the next state and output). We only provided a description of the behavior of the machine
- With simple structures, commonly used automatic synthesis tools are able to determine the optimized implementation of the system
- If we wanted to describe a Moore machine instead of a Mealy machine
  - The sensitivity list of the process used to describe the output would contain only the present state and not the input
  - The case statement used to describe the output would contain only the state and not the input

Simulation of Synchronous Sequential Logic

- It is necessary to **change the inputs earlier enough (at least 1 setup time, t_s) than the active clock edge** (e.g. during the falling edge, in the hypothesis of positive-edge-triggered FF). In this way, the combinational logic has time to calculate the output the and next state before the next active edge of the clock
- We will make the testbench with **two processes**, working in parallel:
  1) The first process **generates the clock**. This process is never explicitly stopped and runs continuously, ensuring that the clock keeps oscillating
  2) The second process **generates the sequence of input signals**, including the reset, with proper synchronization, for example at the falling edges of the clock. This process is stopped with the command "`std.env.stop`", which ends the simulation

Sequence Recognizer: Testbench (1/2)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test_seq_rec is
end test_seq_rec;

architecture test of test_seq_rec is

signal clock, reset, X, Z: std_logic;
signal test_sequence: std_logic_vector (0 to 10):= "01110101100";

constant PERIOD: time := 100 ns;

component seq_rec is
   port (clk, rst, X : in std_logic;
                   Z : out std_logic);
end component;

begin
   DUT: seq_rec port map (clock, reset, X, Z);
```



Definition of a **constant** of type time

Test vector chosen to pass in order through all the states of the diagram

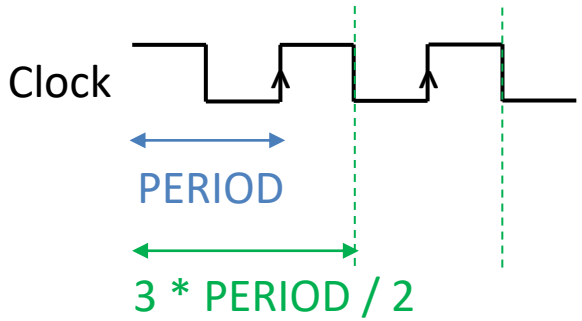
Instantiation of the DUT (circuit to be simulated)

Sequence Recognizer: Testbench (2/2)

```
    generate_clock: process begin
        clock <= '1';
        wait for PERIOD/2;
        clock <= '0';
        wait for PERIOD/2;
    end process;

    apply_inputs: process begin
        reset <= '1';
        X <= '0';
        wait for 3*PERIOD/2;
        reset <= '0';
        for i in 0 to 10 loop
            X <= test_sequence(i);
            wait for PERIOD;
        end loop;
        std.env.stop;
    end process;
end test;
```

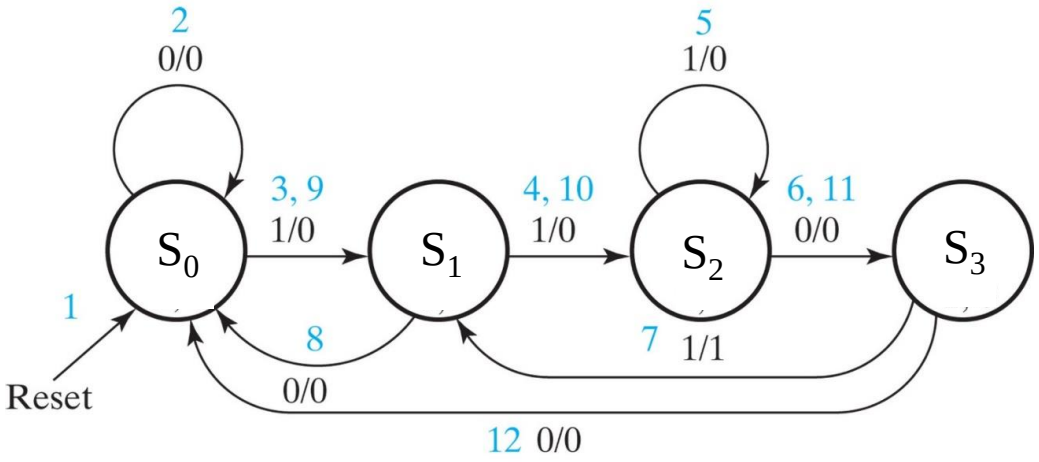
Process to **generate a clock** with period equal to PERIOD

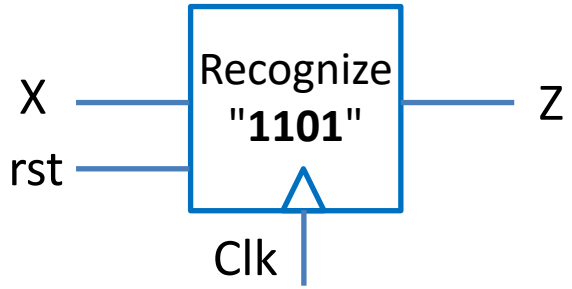
Process to **apply reset and then the sequence of inputs** contained in test_sequence



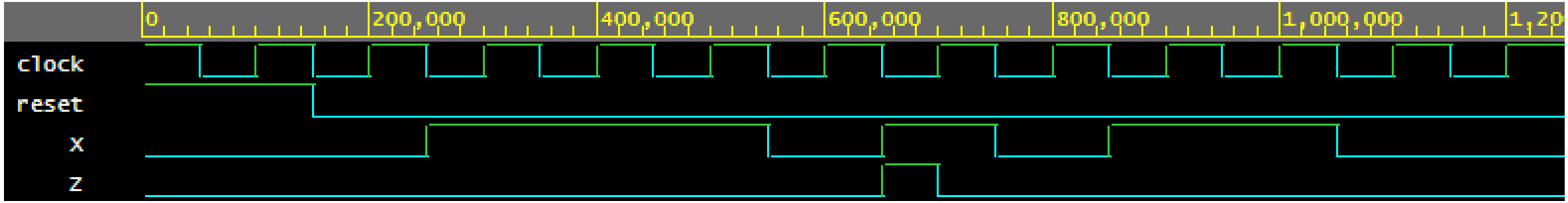
Inputs are changed at the clock falling edges

Sequence Recognizer: Simulation





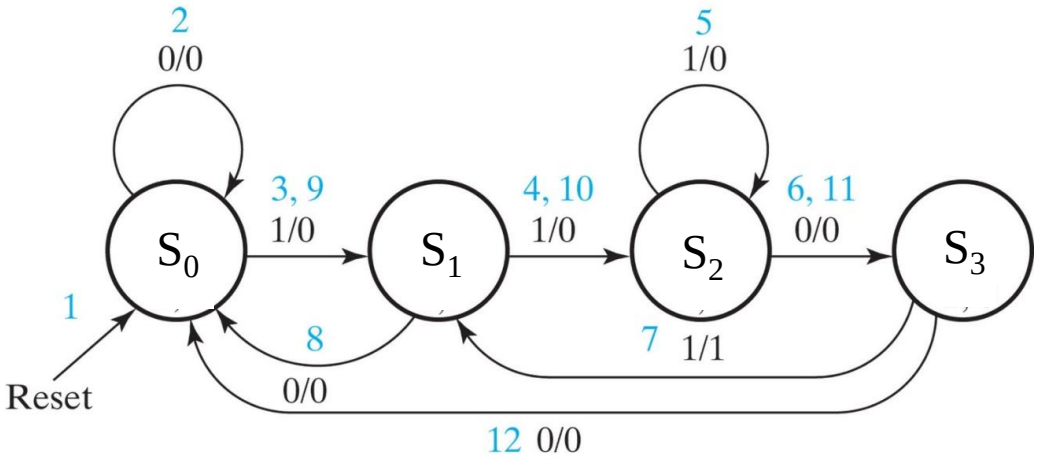
Test sequence: “01110101100”

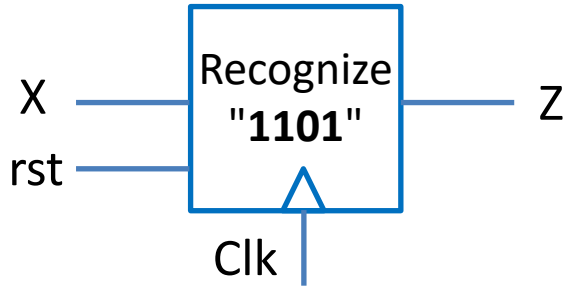


Initial reset: the system starts from a known state

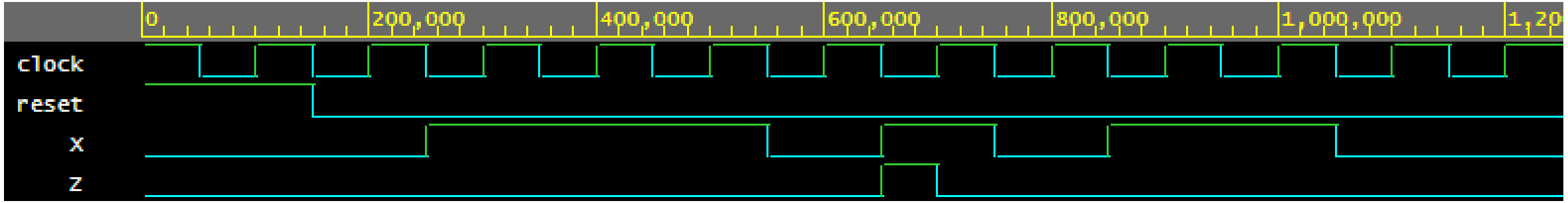
The output becomes 1 at the arrival of the last symbol of the correct sequence (Mealy model: output reacts immediately at the input change)

Sequence Recognizer: Simulation





Test sequence: “01110101100”



Note: In a **Mealy machine**, the output can change also not in correspondence to the active edge of the clock (following a change of the input). This cannot happen in a **Moore machine**, where the output depends only on the state, which is stored in a FF, whose output can change only at active clock edges

Generation of Storage in VHDL

- If we want to make a combinational circuit and forget to specify one (or more) conditions within an if-then-else statement, an unwanted memory element is realized!
- This is intentional when we intend to realize a memory element (latch or flip-flop)
  - Example (PET D-FF):

```
...
process (CLK)
   begin
   if (CLK'event and CLK = '1') then
      if (RESET = '1') then
         Q <= '0';
      else
         Q <= D;
      end if;
   end if;
end process;
```

Only the active clock edge condition is specified on purpose. In all the other conditions, no action is specified. This implicitly means Q <= Q (memory)

Generation of Storage in VHDL

Inputs			Action
RESET = 1	CLK = 1	CLK' event	
FALSE	FALSE	FALSE	Unspecified
FALSE	FALSE	TRUE	Unspecified
FALSE	TRUE	FALSE	Unspecified
FALSE	TRUE	TRUE	Q <= D
TRUE	—	—	Q <= '0'

Memory

Q <= Q

Unwanted Memory Elements

- Let's consider a combinational circuit identifying 3-bit prime numbers. The output is equal to 1 if the input is prime. Suppose we use a process with an «if» statement to describe the circuit
  - Circuit with 1 input (3 bits) and one output (1 bit)

```
architecture implIf of Prime is
begin process (input)
begin
   if (input = 3d"1" OR input = 3d"2" OR input = 3d"3" OR input = 3d"5" OR input = 3d"7")
then isPrime <= '1';
   else isPrime <= '0';
   end if;
end process;
end implIf;
```

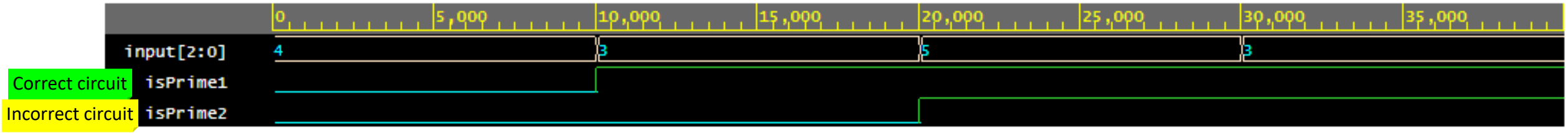
Correct: the behaviour of the circuit is specified in all conditions (with the «else» line)!

Unwanted Memory Elements

- If we forget an «else» condition, we unintentionally create a memory element
  - Example (3-bit prime number):

```
architecture implIf of Prime is
begin process (input)
begin
    if (input = 3d"1" OR input = 3d"2" OR input = 3d"5" OR input = 3d"7")
then isPrime <= '1';
    elsif (input = 3d"4" OR input = 3d"6") then isPrime <= '0';
    end if;
end process;
end implIf;
```

If we forget to specify a case (3d"3"), we create a latch: transitioning to input 3, the system keeps the previous output: isPrime <= isPrime



If ‘3’ (no output specified) is preceded by ‘4’ (isPrime <= ‘0’): the output keeps value ‘0’
If ‘3’ (no output specified) is preceded by ‘5’ (isPrime <= ‘1’): the output keeps value ‘1’

Disclaimer

Figures from*Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano |Kime| Martin