lab 03

Generic modules, components, simulation

Digital design – laboratory exercises

assistant: Nejc Ilc

Generalized modules - generic

- We use the generic clause to parameterize a module during the instantiation, e.g., n-bit counter.
- We specify a default value at declaration and the final value at instantiation.

```
entity counter is

port (
    clock: in  STD_LOGIC;
    reset: in  STD_LOGIC;
    value: out
         STD_LOGIC_VECTOR (3 downto 0);
 );
end counter;
```

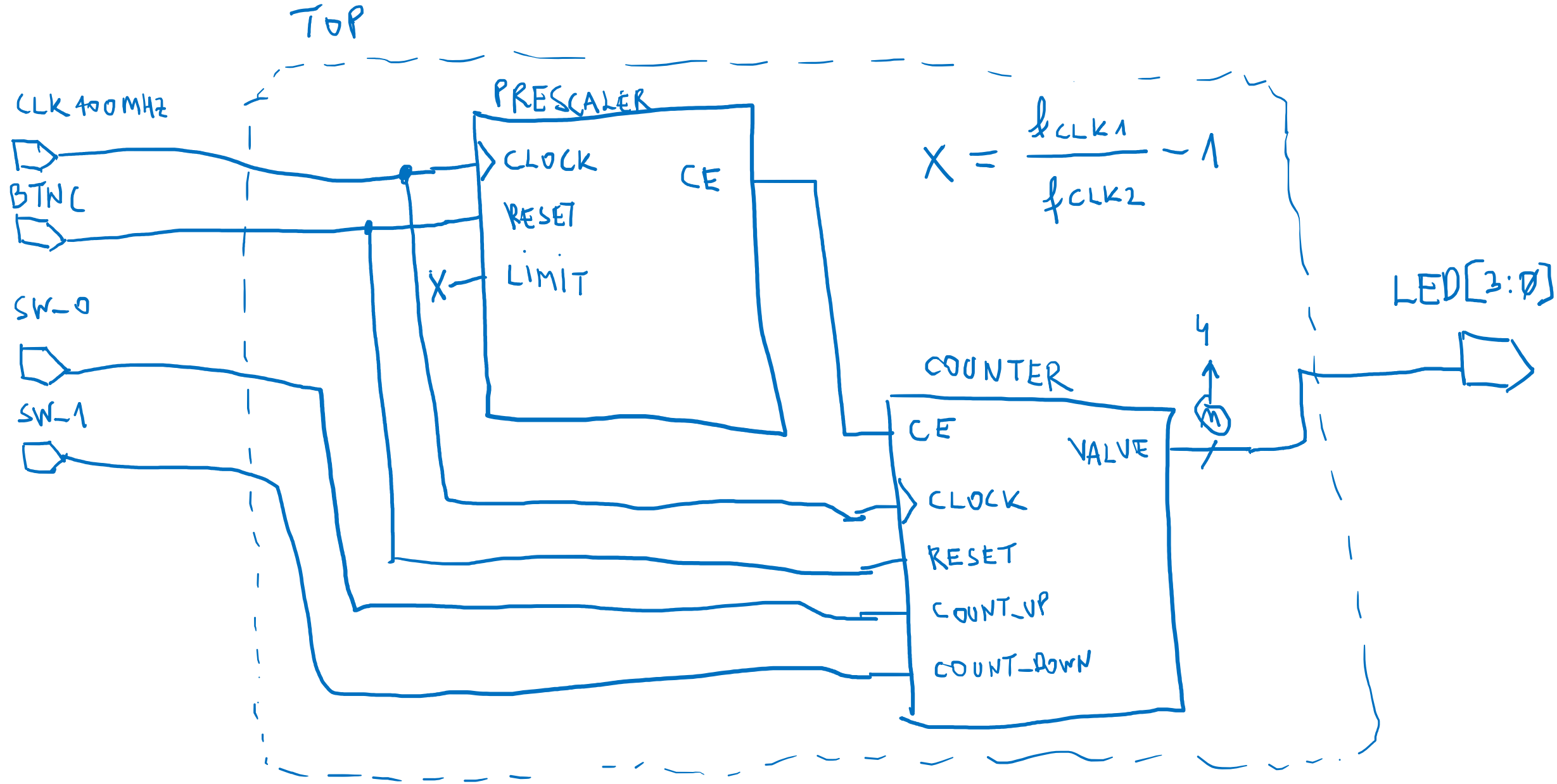
```
entity counter is
generic (n: integer := 8);
port (
    clock: in  STD_LOGIC;
    reset: in  STD_LOGIC;
    value: out
         STD_LOGIC_VECTOR (n-1 downto 0);
 );
end counter;
```

Components

- Our project should have a modular and clean structure.
- We want to reuse already built (generic) modules.
  - Example: we use the description of a n-bit counter for two counters: 4-bit and 8-bit.
- A concept of components enables us to instantiate and connect the already defined modules in a new module.

Example

- Let's build a counter that will de/increment every second.
- We will split the design into three modules (files):
  - `prescaler`
    - inputs: `clock`, `reset`, `limit`
    - outputs: `clock_enable`
  - `counter`
    - inputs : `clock`, `reset`, `clock_enable`, `count_up`, `count_down`
    - outputs : `value`
  - `top`
    - connects the components `prescaler` and `counter` and defines the external interface


TOP
CLK 100MHz
BTNC
SW_0
SW_1
PRESCALER
CLOCK
RESET
LIMIT
X
CE
$X = \frac{f_{CLK1}}{f_{CLK2}} - 1$
COUNTER
CE
VALUE
CLOCK
RESET
COUNT_UP
COUNT_DOWN
4
M
LED[3:0]

Module prescaler

```
entity prescaler is
  generic (
    width: integer := 8; -- the width of a prescaler counter
  );
  port (
    clock:        in  std_logic;
    reset:        in  std_logic;
    limit:        in  integer;
    clock_enable: out std_logic;
  );
end prescaler;
```

Module counter

```
entity counter is
  generic (
    width: integer := 4;
  );
  port (
    clock:        in  std_logic;
    reset:        in  std_logic;
    clock_enable: in  std_logic;
    count_up:     in  std_logic;
    count_down:   in  std_logic;
    value:        out signed(width-1 downto 0);
  );
end counter;
```

Module top

- Top-most module cannot be generic.

```
entity top is
  port (
    CLK100MHZ: in  std_logic;
    BTNC:      in  std_logic;
    SW_0:      in  std_logic;
    SW_1:      in  std_logic;
    LED:       out signed(3 downto 0);
  );
end top;
```

A wiring of components – port maps

- A component is a module that we use in the other module.
- Declaration (before begin in the architecture)

```
component component_name
    port ( signal_name: direction type ...);
end component;
```

- Port mapping

```
<label>: component_name
port map (
    signal_name_comp_1 => signal_name_top_1,
    signal_name_comp_2 => signal_name_top_2,
    ...
);
```

Example: declarations

```
-- Constants
constant f_clk_sys: integer := 100e6; -- 100 MHz
constant f_clk:     integer := 1; -- 1 Hz
constant cnt_width: integer := 4;
constant pr_width:  integer := 27;
constant pr_limit:  integer := f_clk_sys / f_clk - 1;

-- Internal signals
signal CE: std_logic := '0';
```

```
-- Components
component counter is generic (width: integer := 4);
    port (
        clock:        in  std_logic;
        reset :       in  std_logic;
        clock_enable: in  std_logic;
        count_up:     in  std_logic;
        count_down:   in  std_logic;
        value:        out signed (width-1 downto 0));
end component;
component prescaler is generic (width : integer := 27);
     port (
        clock:        in  std_logic;
        reset:        in  std_logic;
        limit:        in  integer;
        clock_enable: out std_logic);
end component;
```

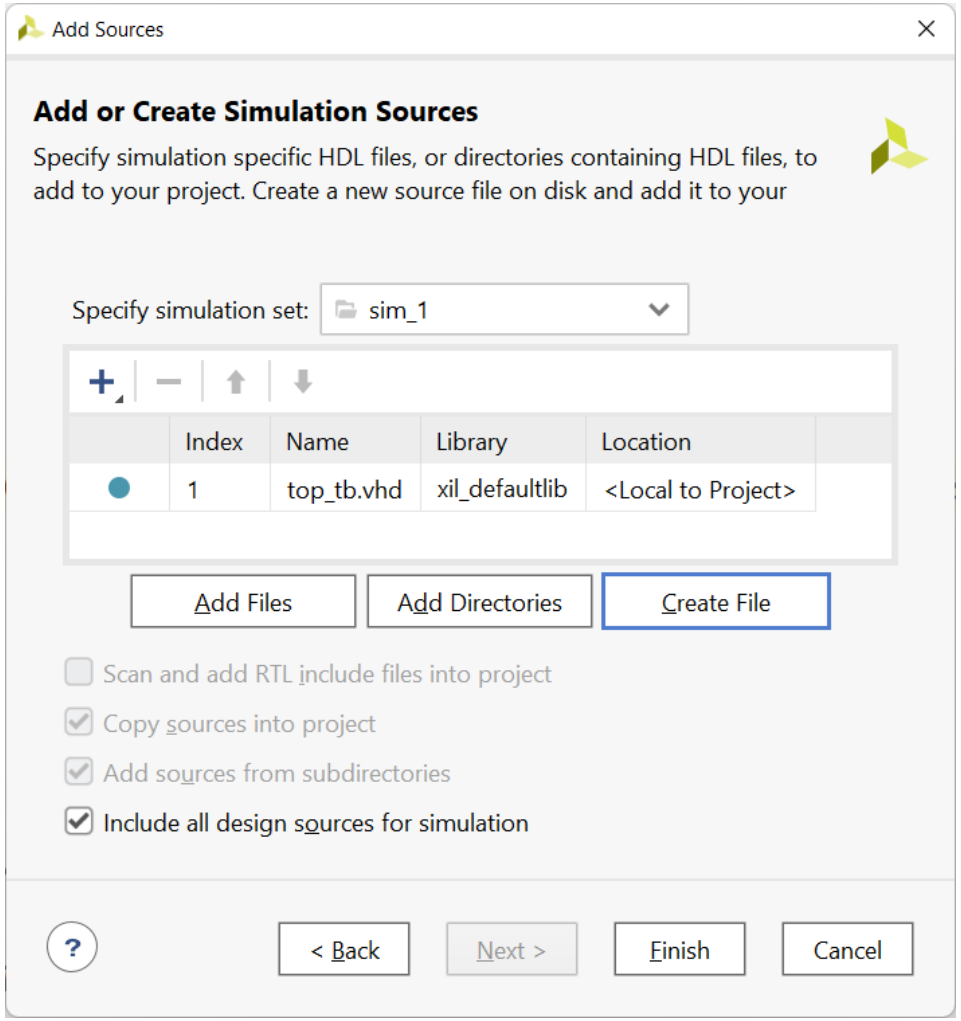
Example: instantiation

```
pr: prescaler
generic map (
  width => pr_width)
port map (
  clock        => CLK100MHZ,
  reset        => BTNC,
  clock_enable => CE,
  limit        => pr_limit
);
```

```
cnt: counter
generic map (
  width => cnt_width)
port map (
  clock        => CLK100MHZ,
  reset        => BTNC,
  clock_enable => CE,
  count_up     => SW_0,
  count_down   => SW_1,
  value        => LED
);
```

Simulation

- We run "Behavioral simulation" before going into synthesis.
- Let's write a testing scenario, i.e., "test bench".
- We define stimuli of the module's inputs and observe its outputs.
- File → Add Sources … → Add or Create Simulation Sources → Create File → <module_name>_tb.vhd



Simulation: test bench

- Entity in a test bench does not have any inputs or outputs.
- In the architecture:
  - we declare and instantiate a component under test (UUT – unit under test),
  - we define stimuli using processes
    - a process for a clock,
    - a process for other stimuli.

Simulation: test bench (2)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top_tb is
end entity;

architecture Behavioral of top_tb is
    constant clock_period: time := 10 ns;
    signal clock:         std_logic := '0';
    signal reset:         std_logic := '0';
    signal count_up:      std_logic := '0';
    signal count_down:    std_logic := '0';
    signal counter_value: signed (3 downto 0);
```

```
component top is
  port ( CLK100MHZ: in  std_logic;
         BTNC:      in  std_logic;
         SW_0:      in  std_logic;
         SW_1:      in  std_logic;
         LED:       out signed (3 downto 0));
end component;

begin
  uut: top
  port map( CLK100MHZ => clock,
            BTNC      => reset,
            SW_0      => count_up,
            SW_1      => count_down,
            LED       => counter_value);
```

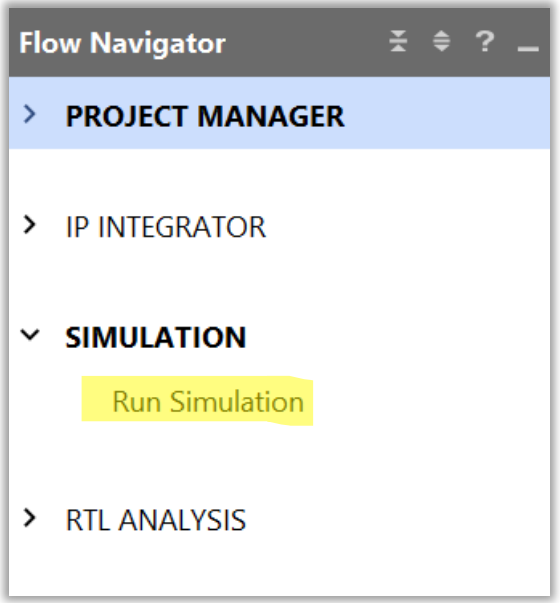
Simulation: test bench (3)

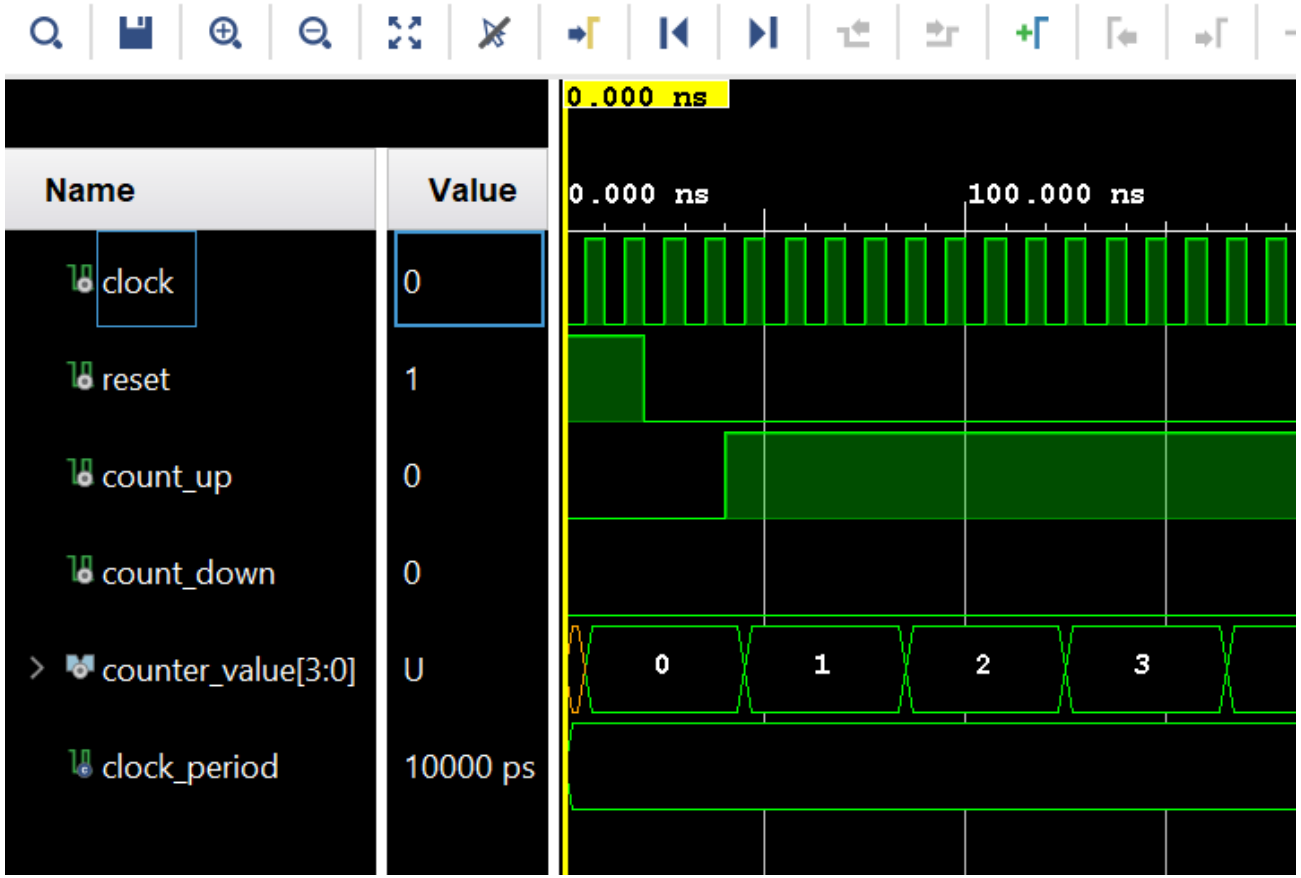
```
clk: process
    begin
        wait for clock_period/2;
        clock <= not clock;
    end process;
```

```
stimuli: process
    begin
        reset <= '1';
        wait for 2*clock_period;
        reset <= '0';
        wait for 2*clock_period;
        count_up <= '1';
        wait for 15*clock_period;
        count_up <= '0';
        count_down <= '1';
        wait; -- wait forever ...
    end process;
end Behavioral;
```

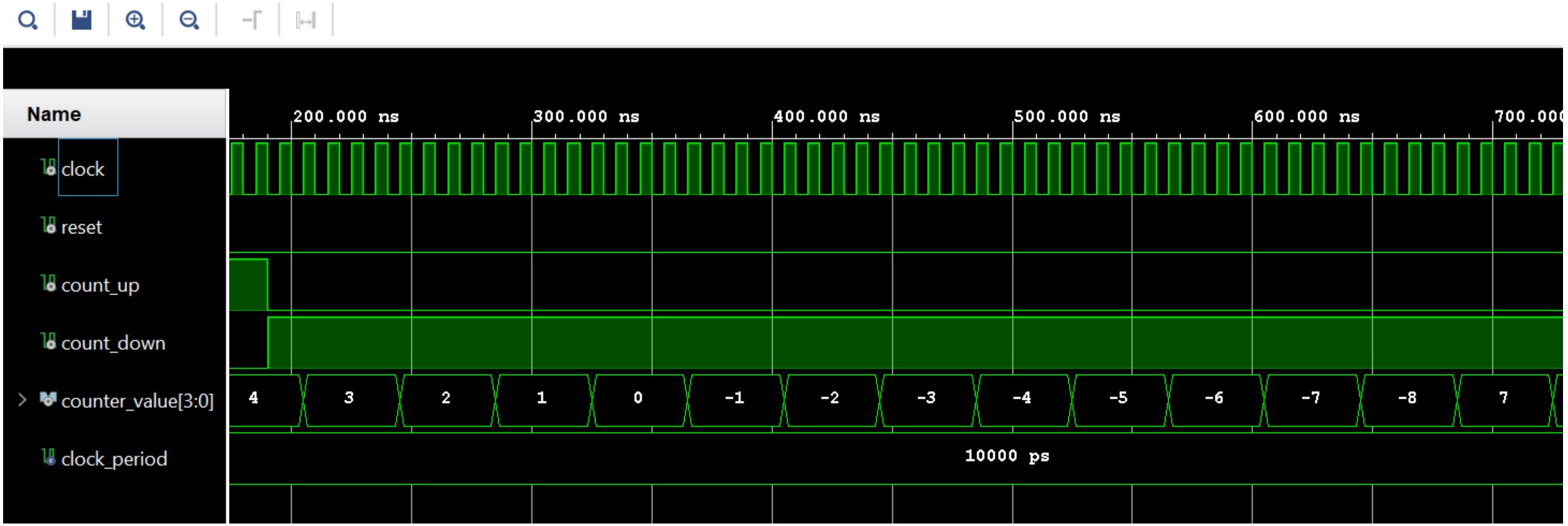
Simulation: waveforms

- Launching a simulation:
  - Flow → Run Simulation → Run Behavioral Simulation





Simulation: waveforms (2)



Note: the value of `counter_value` is displayed as signed decimal number (Radix → Signed Decimal)

Challenge

Solve the challenge 02 (a LED scroller) using components:

- `prescaler`
  - slows down the clock
  - let it have a generic width of a counter register
- `scroller`
  - it creates a scrolling effect on LEDs considering the prescaler "tempo"
  - let it have a generic width of a register that stores the state of LEDs; consequently, it means the number of LEDs we control.
- `top`
  - top module that wires the modules `prescaler` in `scroller` together.