Hardware Modeling [VU] (191.011)
– WS25 –
Sequential Circuit Example: LFSR

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

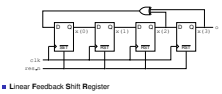
In the previous lecture you learned how different flavors of latches and flip-flops can be described in VHDL. However, while these elements enable the construction of sequential circuits within the synchronous paradigm, you have not yet seen how such a circuit can actually be described. This is what we'll do in this lecture.

Hardware Modeling [VU] (191.011)
– WS25 –
Sequential Circuit Example: LFSR

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

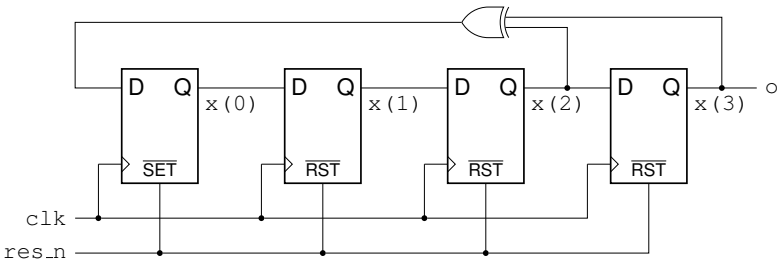
WS 2025/26

Modified: 2025-11-11, 14:41 (f8a58e9)

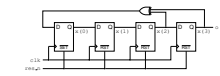


On the slide you can find the schematic of our example circuit. In particular, this circuit is a so-called linear-feedback-shift-register, abbreviated as LFSR.

Example: 4-bit LFSR



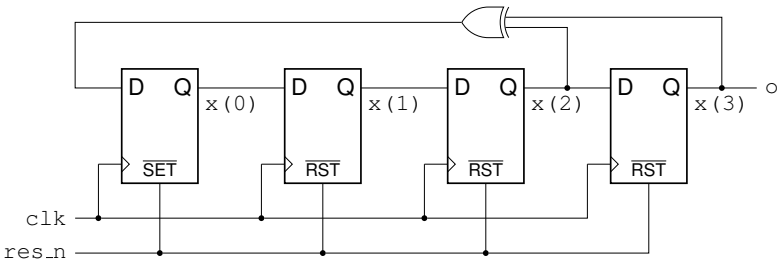
- **L**inear **F**eedback **S**hift **R**egister



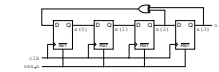
- Linear Feedback Shift Register
- Chain of FFs (*shift-register*) fed by linear combination of its current state

As the name suggests, such circuits revolve around a so-called *shift register*, fed by a linear combination of its current state. A shift register itself is nothing more than a chain of flip-flops, where each flip-flop samples the output of its predecessor. This essentially shifts the currently stored bits by one flip-flop per clock period, hence the name.

Example: 4-bit LFSR



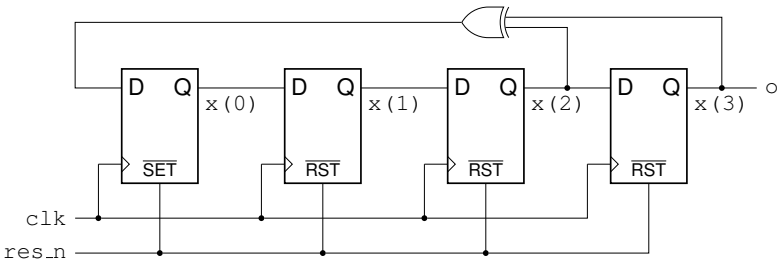
- **L**inear **F**eedback **S**hift **R**egister
- Chain of FFs (*shift-register*) fed by linear combination of its current state



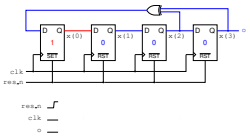
- Linear Feedback Shift Register
- Chain of FFs (*shift-register*) fed by linear combination of its current state
- Pseudo-random sequence of bits

Note that this type of circuit is not some artificial example, but actually something commonly encountered in circuits. In particular, LFSRs are often used as pseudo-random number generators. This means that they can generate a sequence of bits that looks like it is random while it actually is not. We will now discuss how the circuit operates.

Example: 4-bit LFSR

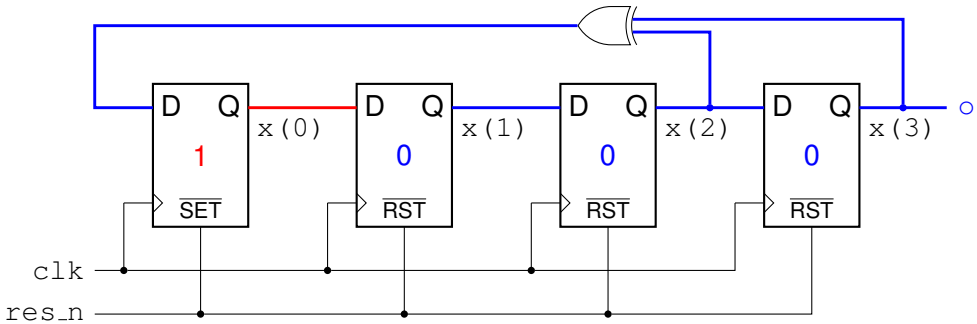


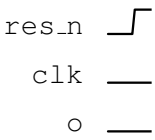
- **L**inear **F**eedback **S**hift **R**egister
- Chain of FFs (*shift-register*) fed by linear combination of its current state
- Pseudo-random sequence of bits

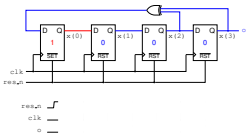


To demonstrate the circuit's operation, we highlight the data wires of the circuit in blue when propagating a logical 0, and in red for logical 1. We also show the logical value currently held by each flip-flop and a timing diagram of the inputs and outputs underneath the circuit.

LFSR - Circuit Operation Principle

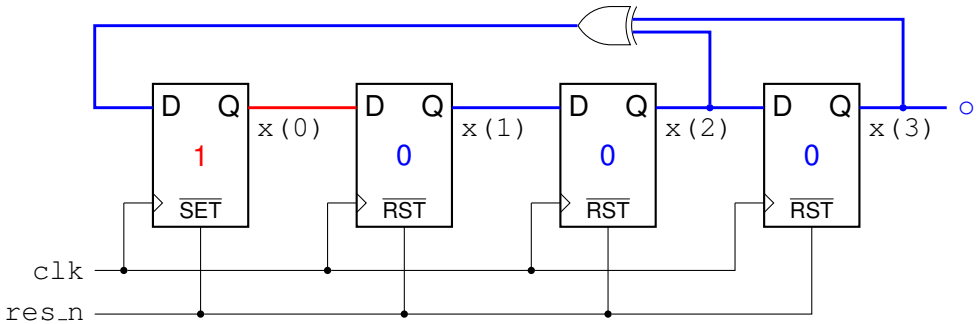


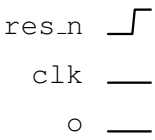


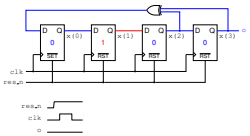


As you can observe in the wave diagram, the first thing that is done is to reset the circuit – just as we discussed in the previous lecture. As a result, all flip-flops, except the first one, hold a zero. This initial value is vital, as all flip-flops being zero would result in the output being constantly zero as well. And that's not very random.

LFSR - Circuit Operation Principle

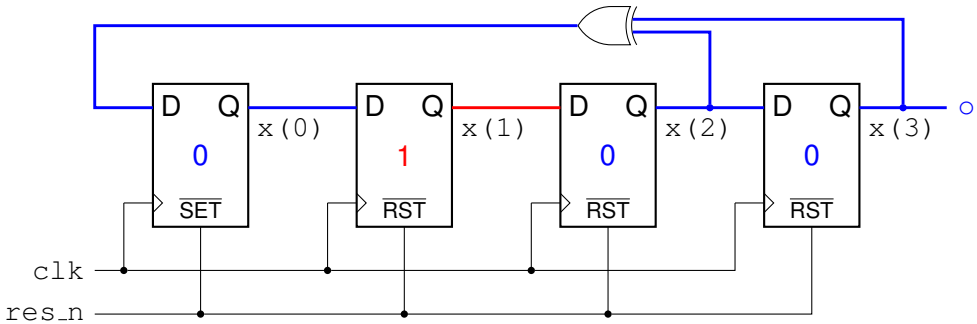


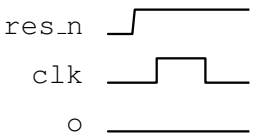


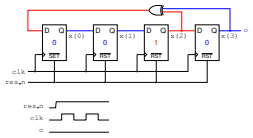


At the first rising clock edge, all flip-flops sample their inputs. This leads to the stored 1 being shifted to the right. The left-most flip-flop samples the output of the XOR gate, which is 0 since the two right-most flip-flops both hold a 0.

LFSR - Circuit Operation Principle

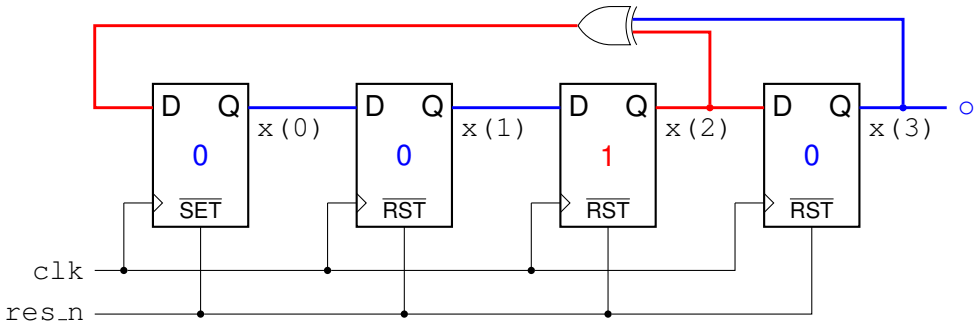


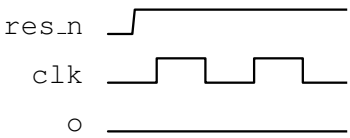


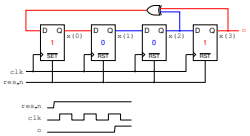


At the next clock edge, the current register bits are shifted to the right again. Note how this makes the feedback path high.

LFSR - Circuit Operation Principle

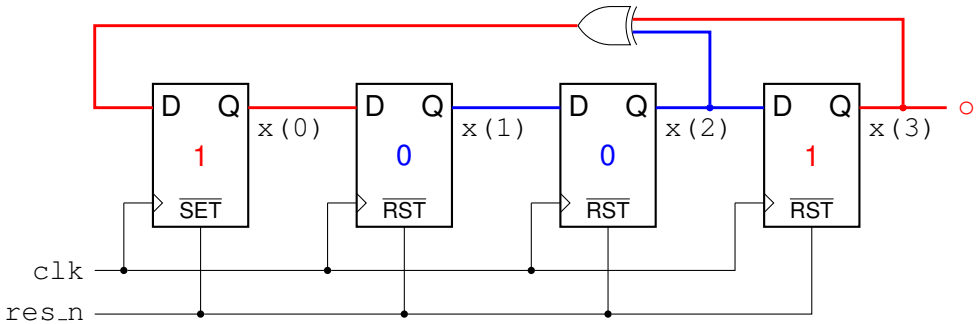


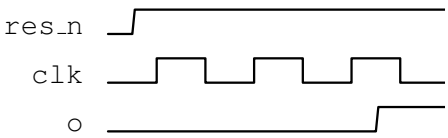


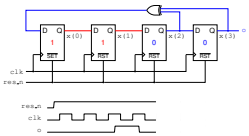


The first flip-flop now samples the applied 1 from the feedback path, while the already stored 1 is shifted into the last flip-flop As a result, the output signal is asserted, as can also be observed in the wave diagram.

LFSR - Circuit Operation Principle

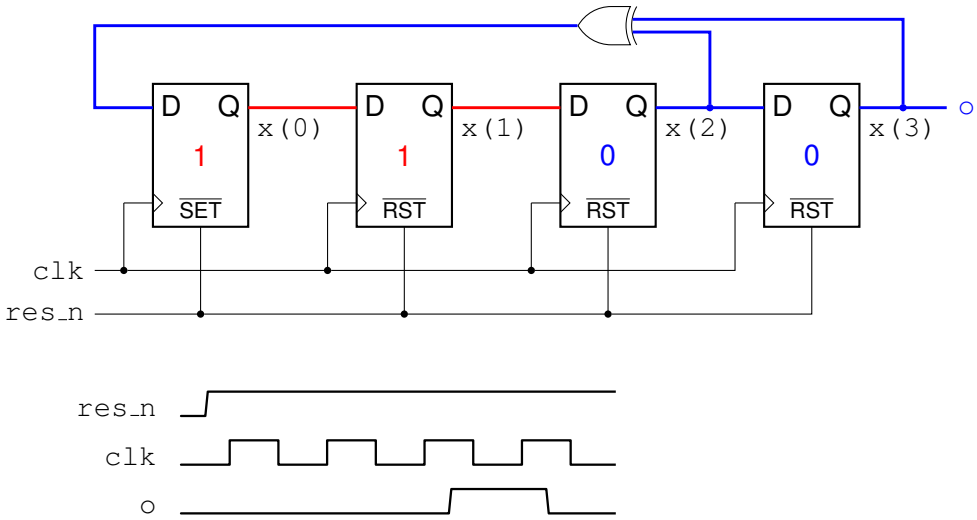


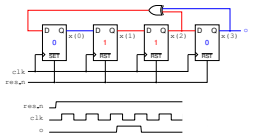




The shifting of bits continues, making the output become low again.

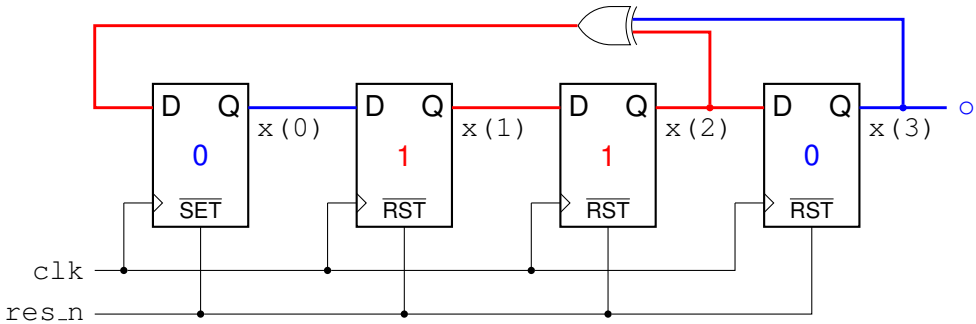
LFSR - Circuit Operation Principle

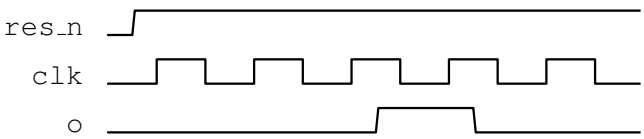


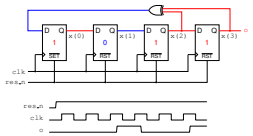


The LFSR continues in this fashion of sampling the feedback and shifting its internal state, producing a sequence of 0s and 1s at its output.

LFSR - Circuit Operation Principle

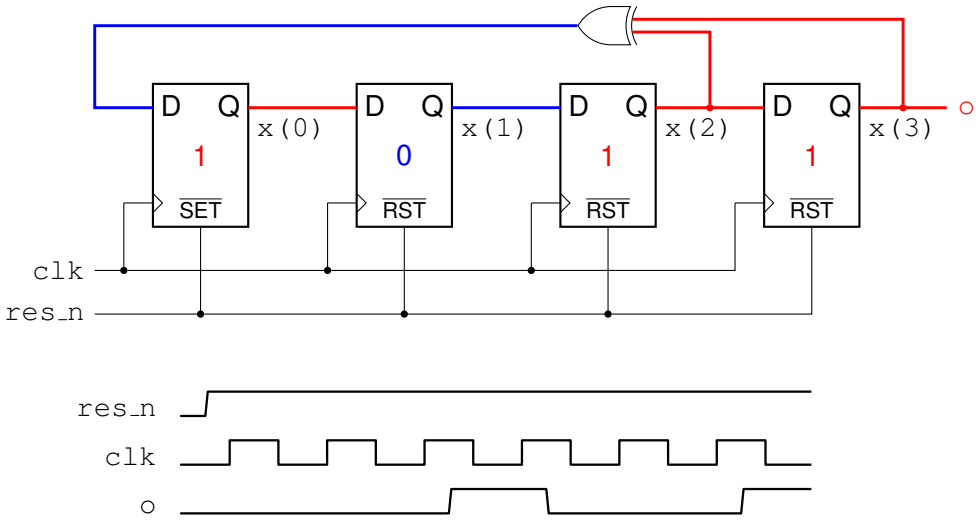


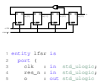




The shifting now leads to the output being set for two consecutive clock cycles. However, by now you have likely gotten the hang of it, and we can continue by discussing how such a circuit can be described in VHDL.

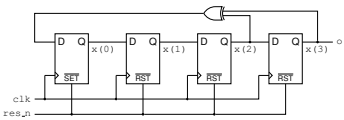
LFSR - Circuit Operation Principle



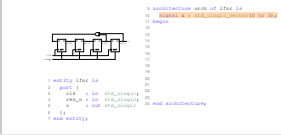


On the slide you can find an entity declaration for the LFSR circuit. The ports should not be too surprising, containing a clock, reset and output. Let us now look at the accompanying architecture.

LFSR - VHDL Design

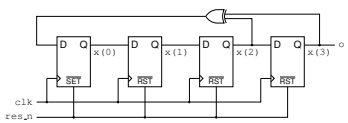



```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```



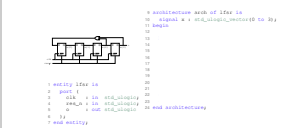
In its declarative part, the architecture declares a single vector signal for the register `x`.

LFSR - VHDL Design



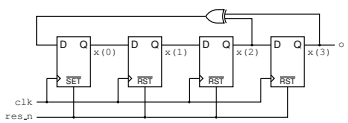

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin




end architecture;
```



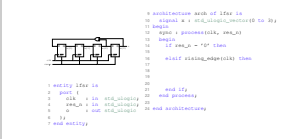
As shown in the previous lecture, to model this register, we can use a process that asynchronously resets this signal and updates it at rising clock edges.

LFSR - VHDL Design



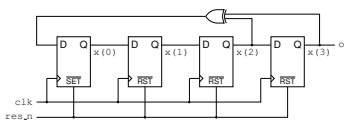

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin









end architecture;
```



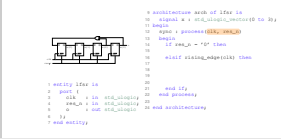
Hence, we add a respective process with the respective `if`-structure added inside the process. Note that we assume that the flip-flops are reset asynchronously.

LFSR - VHDL Design



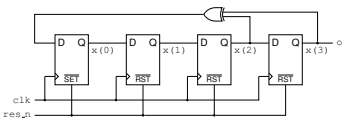

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin
  sync : process(clk, res_n)
  begin
    if res_n = '0' then

    elsif rising_edge(clk) then









    end if;
  end process;

end architecture;
```



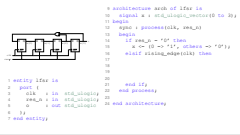
Recall that this also means that the process must be sensitive to both the reset, and the clock.

LFSR - VHDL Design



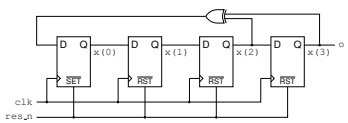

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin
  sync : process(clk, res_n)
  begin
    if res_n = '0' then

    elsif rising_edge(clk) then









    end if;
  end process;

end architecture;
```



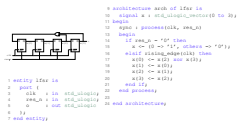
During the asynchronous reset, all bits of `x`, except for the one at index 0, are reset to `'0'` via an aggregate expression. The first bit is set to `'1'` to model the asynchronous set of the left-most flip-flop we mentioned before.

LFSR - VHDL Design



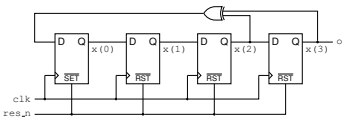

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin
  sync : process(clk, res_n)
  begin
    if res_n = '0' then
      x <= (0 => '1', others => '0');
    elsif rising_edge(clk) then




    end if;
  end process;

end architecture;
```



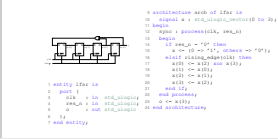
At rising clock edges, the register is updated such that it implements the introduced LFSR. In particular, this means that the distinct bits of the register are connected to form a chain of flip-flops, where each flip-flop samples the output of its predecessor. The left-most flip-flop samples the XOR of the bits 2 and 3.

LFSR - VHDL Design



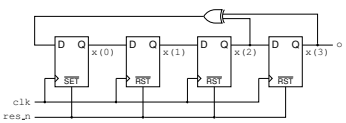

```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin
  sync : process(clk, res_n)
  begin
    if res_n = '0' then
      x <= (0 => '1', others => '0');
    elsif rising_edge(clk) then
      x(0) <= x(2) xor x(3);
      x(1) <= x(0);
      x(2) <= x(1);
      x(3) <= x(2);
    end if;
  end process;

end architecture;
```



Finally, we use a concurrent signal assignment to connect the output of the right-most flip-flop to the output port o.

LFSR - VHDL Design




```vhdl
entity lfsr is
  port (
    clk   : in  std_ulogic;
    res_n : in  std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of lfsr is
  signal x : std_ulogic_vector(0 to 3);
begin
  sync : process(clk, res_n)
  begin
    if res_n = '0' then
      x <= (0 => '1', others => '0');
    elsif rising_edge(clk) then
      x(0) <= x(2) xor x(3);
      x(1) <= x(0);
      x(2) <= x(1);
      x(3) <= x(2);
    end if;
  end process;
  o <= x(3);
end architecture;
```

To conclude this example, let us discuss how a testbench can be written for the LFSR. On the slide you can already find a part of the architecture for such a testbench. It declares the signals required by the unit under test, and instantiates it.

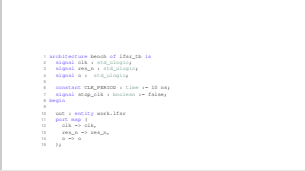
LFSR - Testbench


```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6
7
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```

Obviously, a synchronous design like this one requires a clock to simulate it. But where does this clock signal come from? As for all inputs of the unit-under-test, the testbench has to generate and apply it. This is what we'll address now, as you previously only had to test combinational designs.

LFSR - Testbench

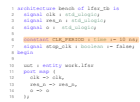

```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6
7
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```



First, we declare an additional constant and a signal.

LFSR - Testbench

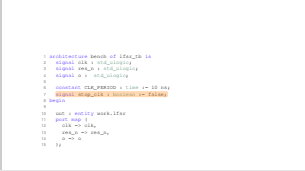

```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```



The constant holds the desired clock period, in this case `10 ns`. Often, the exact value of the clock period is not too important. However, in general, a design is specified for a range or a particular clock frequency. The testbench should then also adhere to this. An exception where setting the right clock period is paramount are *postlayout* simulations, but more on that at another time.

LFSR - Testbench

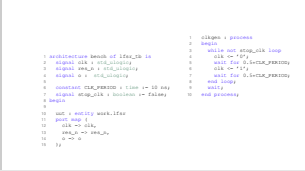

```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```



The purpose of the `stop_clk` signal will become clear in a moment.

LFSR - Testbench


```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```

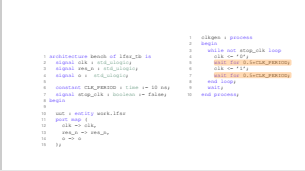


TO deal with the clock, we start by creating a dedicated clock generation process. The purpose of this **non-synthesizable** code is to periodically toggle the clock signal.

LFSR - Testbench


```vhdl
architecture bench of lfsr_tb is
  signal clk : std_ulogic;
  signal res_n : std_ulogic;
  signal o :  std_ulogic;

  constant CLK_PERIOD : time := 10 ns;
  signal stop_clk : boolean := false;
begin

  uut : entity work.lfsr
  port map (
    clk => clk,
    res_n => res_n,
    o => o
  );
```

```vhdl
clkgen : process
begin
  while not stop_clk loop
    clk <= '0';
    wait for 0.5*CLK_PERIOD;
    clk <= '1';
    wait for 0.5*CLK_PERIOD;
  end loop;
  wait;
end process;
```

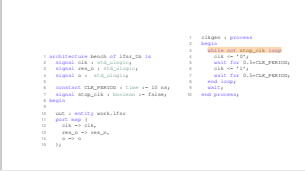


The desired clock period is achieved by waiting for half the respective constant after each of the two assignments.

LFSR - Testbench


```
architecture bench of lfsr_tb is
  signal clk : std_ulogic;
  signal res_n : std_ulogic;
  signal o :  std_ulogic;

  constant CLK_PERIOD : time := 10 ns;
  signal stop_clk : boolean := false;
begin

  uut : entity work.lfsr
  port map (
    clk => clk,
    res_n => res_n,
    o => o
  );
```

```
clkgen : process
begin
  while not stop_clk loop
    clk <= '0';
    wait for 0.5*CLK_PERIOD;
    clk <= '1';
    wait for 0.5*CLK_PERIOD;
  end loop;
  wait;
end process;
```

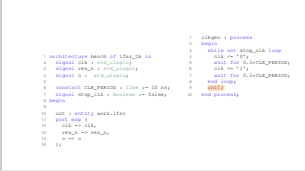


However, if we want the simulation to terminate automatically, we must ensure that all signals become stable eventually. Naturally, this includes the clock signal. Therefore, we wrap the assignments to the clock signal in a while-loop that runs until the `stop_clk` signal becomes `true`.

LFSR - Testbench


```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```

```
1   clkgen : process
2   begin
3     while not stop_clk loop
4       clk <= '0';
5       wait for 0.5*CLK_PERIOD;
6       clk <= '1';
7       wait for 0.5*CLK_PERIOD;
8     end loop;
9     wait;
10  end process;
```

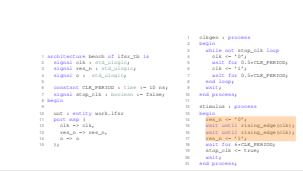


After that, the clock remains stable, and the clock generation process will `wait` indefinitely. Note that the stop signal will be set by the stimulus process once it is done with testing the unit-under-test. We will now look at this process.

LFSR - Testbench


```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```

```
1   clkgen : process
2   begin
3     while not stop_clk loop
4       clk <= '0';
5       wait for 0.5*CLK_PERIOD;
6       clk <= '1';
7       wait for 0.5*CLK_PERIOD;
8     end loop;
9     wait;
10  end process;
```

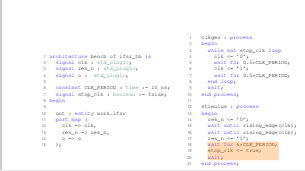


The first task of the stimulus process is to reset the UUT. As already mentioned, otherwise the design might be in an arbitrary state which prohibits proper testing. Note that the reset should be applied for some time, ideally more than a clock cycle, to ensure that everything is properly reset. In this case we activate the reset until two rising clock edges were observed.

LFSR - Testbench


```vhdl
architecture bench of lfsr_tb is
  signal clk : std_ulogic;
  signal res_n : std_ulogic;
  signal o :  std_ulogic;

  constant CLK_PERIOD : time := 10 ns;
  signal stop_clk : boolean := false;
begin

  uut : entity work.lfsr
  port map (
    clk => clk,
    res_n => res_n,
    o => o
  );
```

```vhdl
clkgen : process
begin
  while not stop_clk loop
    clk <= '0';
    wait for 0.5*CLK_PERIOD;
    clk <= '1';
    wait for 0.5*CLK_PERIOD;
  end loop;
  wait;
end process;

stimulus : process
begin
  res_n <= '0';
  wait until rising_edge(clk);
  wait until rising_edge(clk);
  res_n <= '1';
  wait for 6*CLK_PERIOD;
  stop_clk <= true;
  wait;
end process;
```



Afterwards, we let the LFSR run for six clock periods and then stop the clock. The simulation will then be able to automatically terminate because all signals are stable. Finally, we want to remark that the general structure of this testbench can be used for any of the synchronous designs you will encounter in this course.

LFSR - Testbench


```
1  architecture bench of lfsr_tb is
2    signal clk : std_ulogic;
3    signal res_n : std_ulogic;
4    signal o :  std_ulogic;
5
6    constant CLK_PERIOD : time := 10 ns;
7    signal stop_clk : boolean := false;
8  begin
9
10   uut : entity work.lfsr
11   port map (
12     clk => clk,
13     res_n => res_n,
14     o => o
15   );
```

```
1   clkgen : process
2   begin
3     while not stop_clk loop
4       clk <= '0';
5       wait for 0.5*CLK_PERIOD;
6       clk <= '1';
7       wait for 0.5*CLK_PERIOD;
8     end loop;
9     wait;
10  end process;
11
12  stimulus : process
13  begin
14    res_n <= '0';
15    wait until rising_edge(clk);
16    wait until rising_edge(clk);
17    res_n <= '1';
18    wait for 6*CLK_PERIOD;
19    stop_clk <= true;
20    wait;
21  end process;
```

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!