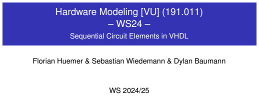


Hardware Modeling [VU] (191.011)
– WS24 –
Sequential Circuit Elements in VHDL

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

This lecture covers how sequential circuit elements, such as flip-flops and latches, are modeled in VHDL, how they differ and how to deal with them in simulation.

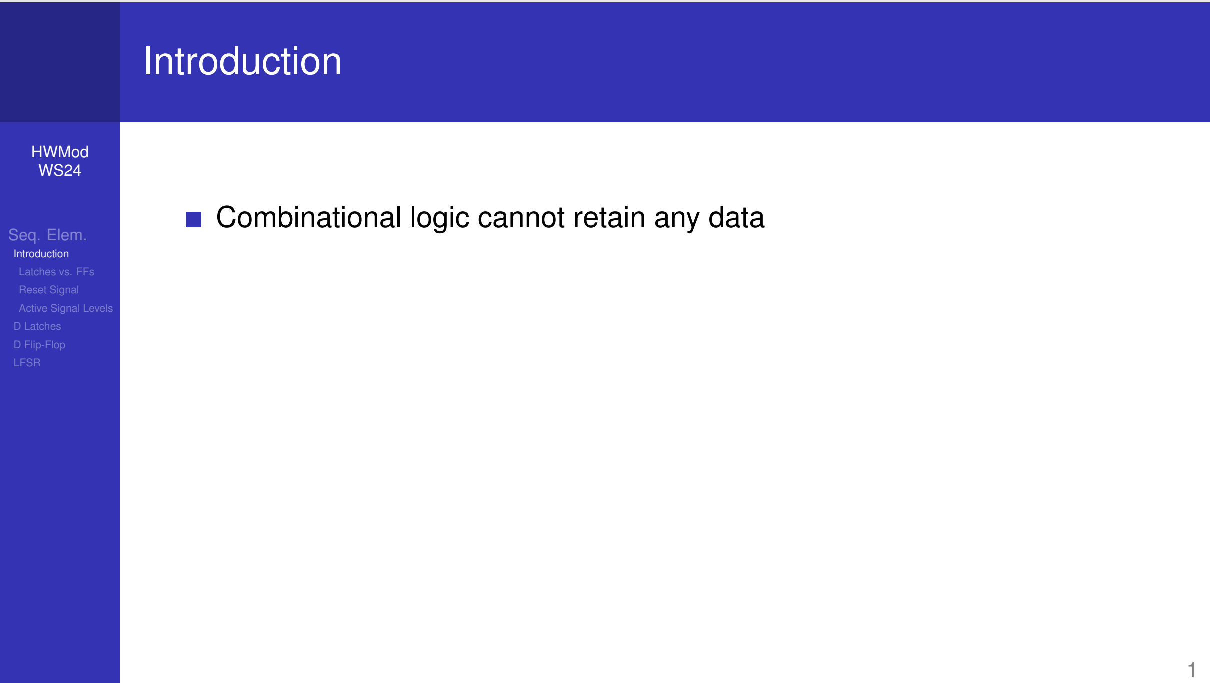
Hardware Modeling [VU] (191.011)
– WS24 –
Sequential Circuit Elements in VHDL

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

- Combinational logic cannot retain any data

So far, we have only dealt with purely combinational VHDL designs. That is, circuits that compute outputs based solely on their current inputs, without any memory or feedback. These circuits do not store state information, meaning their outputs are immediately and directly influenced by changes in inputs, with no dependence on previous input values or timing. However, as we have learned in the lecture about synchronous design, sequential circuits require memory elements, allowing them to store and utilize past states or input values. In synchronous designs, the circuit behavior is coordinated by a clock signal, controlling when state changes occur in memory elements such as latches and flip-flops. In this lecture we will show you how to implement these memory elements in VHDL and how to use them in a circuit.



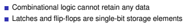
Introduction

- Combinational logic cannot retain any data

Sequential Circuit Elements in VHDL

Introduction

Introduction

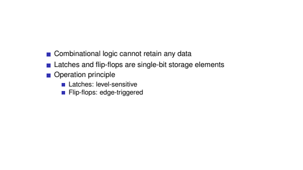


- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements

Both latches and flip-flops are storage elementsthat hold a single bit of data. An upcoming lecture will discuss larger memory elements, such as RAMs, ROMs and FIFOs.

Introduction

- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements

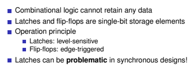


- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered

The major difference between latches and flip-flops is how they react to their input signals and what events cause them to update their internal state. While latches operate based on signal levels, flip-flops only react to rising or falling clock edges.

Introduction

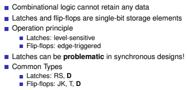
- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered


- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!

The level-sensitive nature of latches makes them somewhat problematic in synchronous designs, as an upcoming lecture will discuss. Thus, they must be strictly avoided in designs during this course. Nevertheless, it is important to cover both concepts as they represent fundamental building blocks of digital systems and to understand the potentially problematic aspects of them.

Introduction

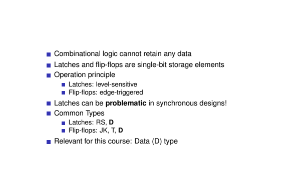
- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!


- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!
- Common Types
  - Latches: RS, **D**
  - Flip-flops: JK, T, **D**

You might have already learned in other courses, that there exists a range of latches and flip-flops, such as RS or D latches, and JK, T or D flip-flops.

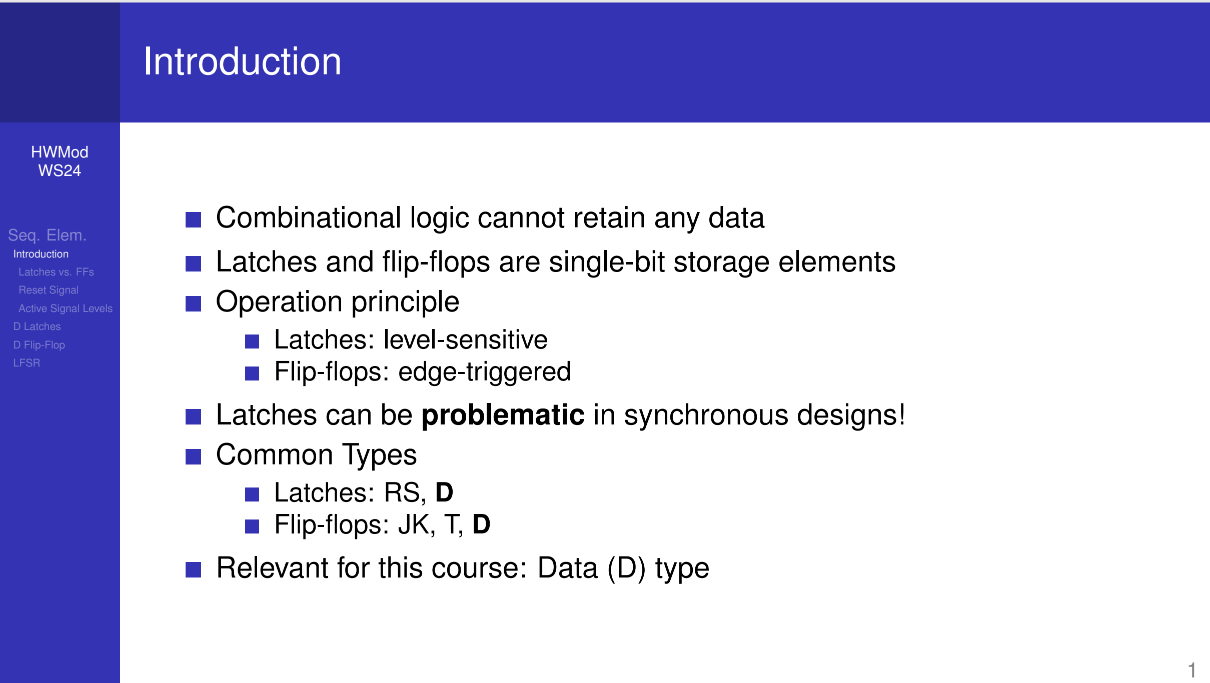
Introduction

- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!
- Common Types
  - Latches: RS, **D**
  - Flip-flops: JK, T, **D**



- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!
- Common Types
  - Latches: RS, **D**
  - Flip-flops: JK, T, **D**
- Relevant for this course: Data (D) type

In this course only the D-type versions are of importance – D standing for data.

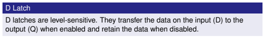


Introduction

- Combinational logic cannot retain any data
- Latches and flip-flops are single-bit storage elements
- Operation principle
  - Latches: level-sensitive
  - Flip-flops: edge-triggered
- Latches can be **problematic** in synchronous designs!
- Common Types
  - Latches: RS, **D**
  - Flip-flops: JK, T, **D**
- Relevant for this course: Data (D) type

OK, let's start with some definitions.

Introduction (cont'd)


D Latch
D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.

A D latch is a level-sensitive single-bit storage device featuring an enable input. Whenever this input is active, the latch continuously transfers the signal level on its input D to its output Q. In this state the latch is also referred to as open, enabled, or transparent, as from the point of view of the input it appears as though the signal passes directly through to the output. When the enable input is inactive, the latch holds the last input value. This is referred to as closed, disabled, or opaque.

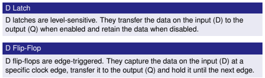
Introduction (cont’d)

HWMod
WS24

Seq. Elem.
Introduction
Latches vs. FFs
Reset Signal
Active Signal Levels
D Latches
D Flip-Flop
LFSR

D Latch

D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.



D Latch
D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.
D Flip-Flop
D flip-flops are edge-triggered. They capture the data on the input (D) at a specific clock edge, transfer it to the output (Q) and hold it until the next edge.

D flip-flops on the other hand only capture their input signal at single moments in time, marked by rising or falling transitions of a clock signal. During all other times the input signals can not affect the output of a D flip-flop. The specific clock edge the flip-flop reacts to is referred to as the active clock edge. Typically, the rising edge is used for this purpose. We will therefore only use this variant in this course.

Introduction (cont'd)

HWMod
WS24

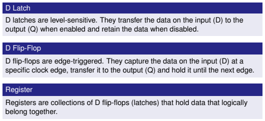
Seq. Elem.
Introduction
Latches vs. FFs
Reset Signal
Active Signal Levels
D Latches
D Flip-Flop
LFSR

D Latch

D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.

D Flip-Flop

D flip-flops are edge-triggered. They capture the data on the input (D) at a specific clock edge, transfer it to the output (Q) and hold it until the next edge.



D Latch
D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.
D Flip-Flop
D flip-flops are edge-triggered. They capture the data on the input (D) at a specific clock edge, transfer it to the output (Q) and hold it until the next edge.
Register
Registers are collections of D flip-flops (latches) that hold data that logically belong together.

Finally, let us define the term register, as it is also important in the context of storage elements. A register is a set of single-bit storage elements – usually flip-flops – that are triggered by the same clock signal and work in unison to store multiple bits of data that logically belong together.

Introduction (cont’d)

HWMod
WS24

Seq. Elem.
Introduction
Latches vs. FFs
Reset Signal
Active Signal Levels
D Latches
D Flip-Flop
LFSR

D Latch

D latches are level-sensitive. They transfer the data on the input (D) to the output (Q) when enabled and retain the data when disabled.

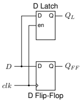
D Flip-Flop

D flip-flops are edge-triggered. They capture the data on the input (D) at a specific clock edge, transfer it to the output (Q) and hold it until the next edge.

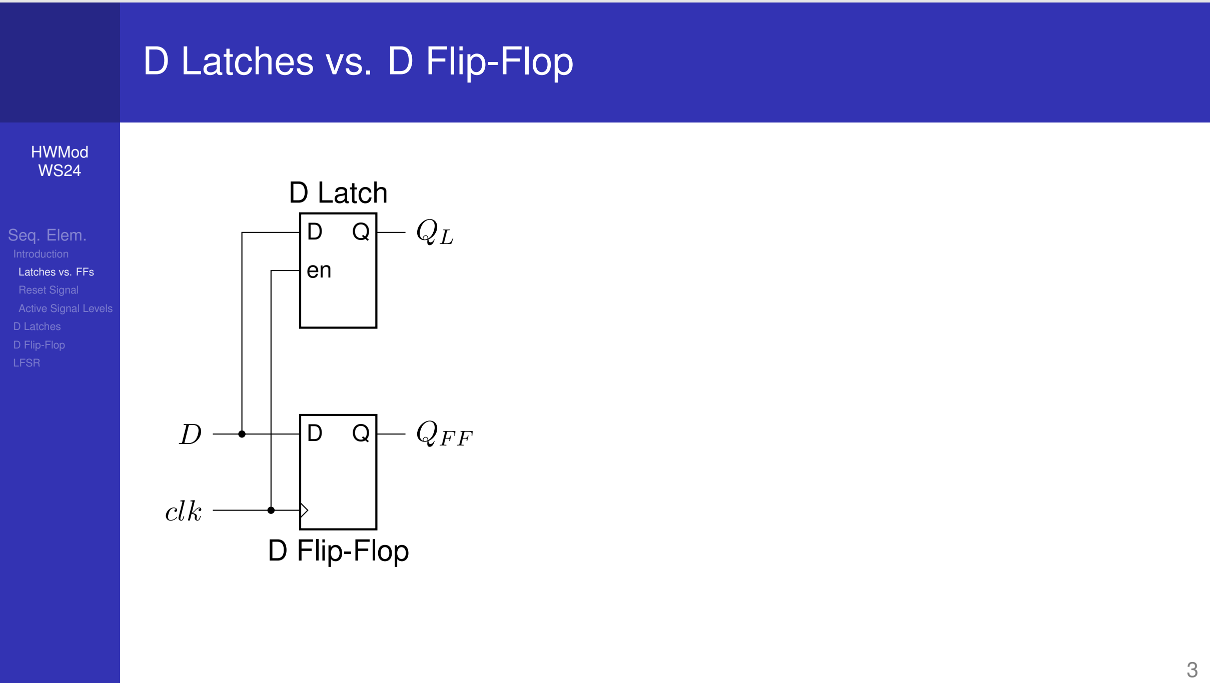
Register

Registers are collections of D flip-flops (latches) that hold data that logically belong together.

└Sequential Circuit Elements in VHDL
└Introduction
└**D Latches vs. D Flip-Flop**



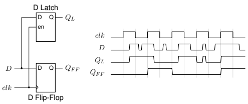
It is crucial that you really understand the difference between latches and flip-flops. Hence, consider this simple demo circuit consisting of a D latch and a D flip-flop. Both storage elements are connected to the same input signal D. The signal used to clock the flip-flop (clk) is used as the enable signal for the D latch.



D Latches vs. D Flip-Flop

HWMod
WS24

Seq. Elem.
- Introduction
  - Latches vs. FFs
  - Reset Signal
  - Active Signal Levels
- D Latches
- D Flip-Flop
- LFSR

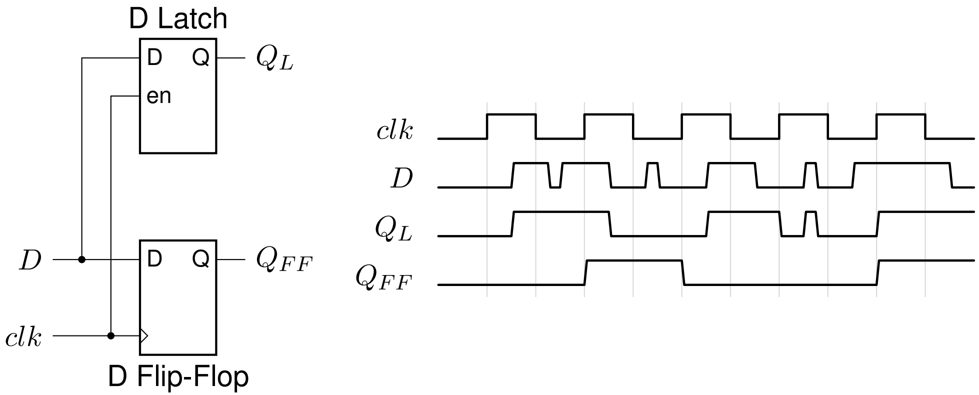


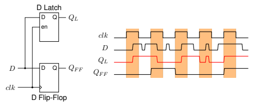
The timing diagram on the right side of the slide shows how the two storage elements react to the same input signals.

D Latches vs. D Flip-Flop

HWMod
WS24

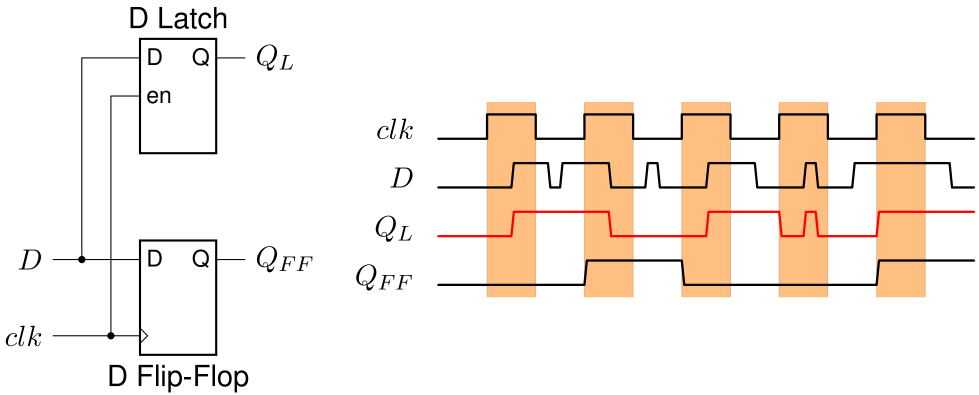
Seq. Elem.
Introduction
Latches vs. FFs
Reset Signal
Active Signal Levels
D Latches
D Flip-Flop
LFSR

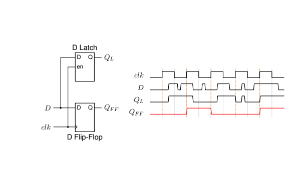




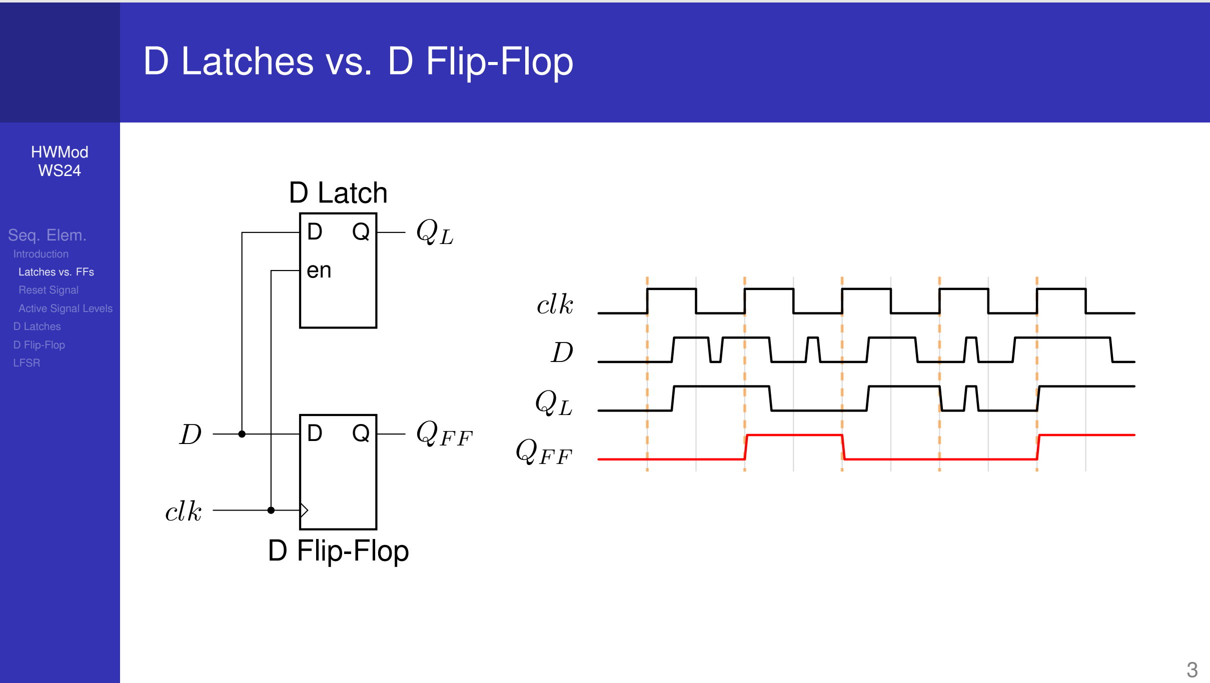
Let’s first consider the D latch. As we know from the previous slide it is transparent when the enable-signal is active. In our case this is during the high period of the clock, which is highlighted in the timing diagram. Notice that during these periods the output signal Q_L directly follows the input signal D. During the low-phase, Q_L is kept at the last value it had during the high period – changes of D during this phase have no effect.

D Latches vs. D Flip-Flop





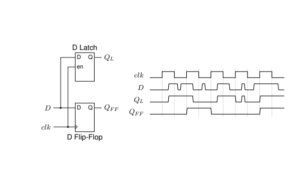
For the D flip-flop only the rising clock edges are relevant. Only at these points in time D is captured and transferred to Q_{FF}. For the rest of the time the value of D is completely irrelevant.



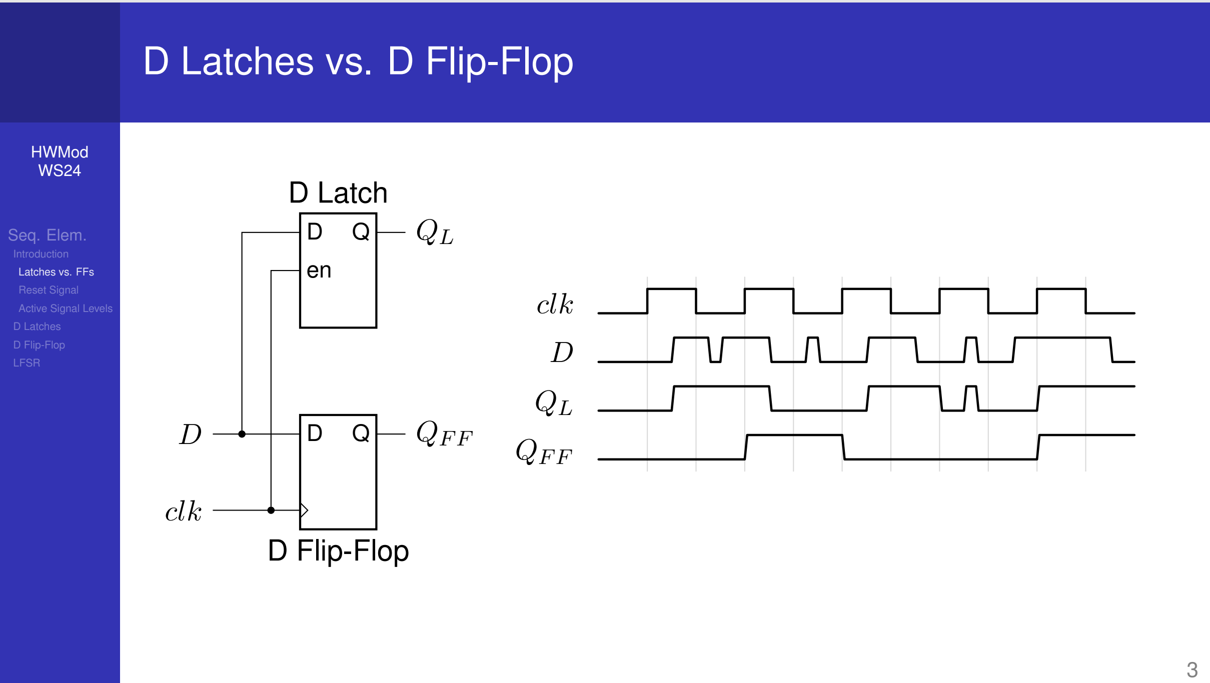
D Latches vs. D Flip-Flop

HWMod
WS24

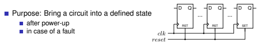
Seq. Elem.
- Introduction
- Latches vs. FFs
- Reset Signal
- Active Signal Levels
- D Latches
- D Flip-Flop
- LFSR



Please, do not continue watching this lecture if this paramount difference is not clear to you at this point! If necessary, pause the video to really understand the shown timing diagram.

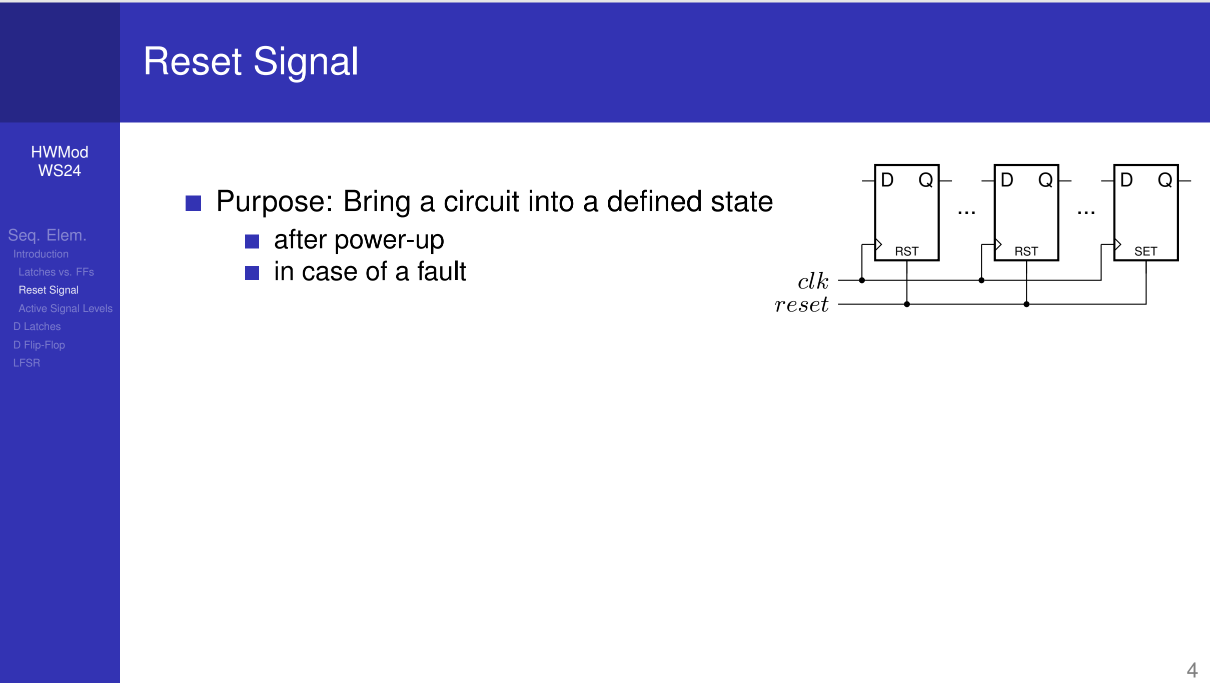


D Latches vs. D Flip-Flop



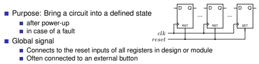
- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault

Most of the time in digital circuit design we want the storage elements to be initialized with a specific value upon power-up or when the circuit entered a faulty state from which we cannot recover by other means. This is the purpose of the reset signal.



Reset Signal

- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault

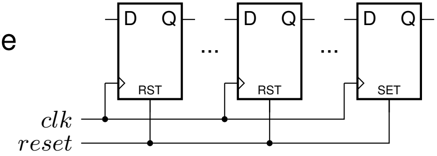


- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button

Like the clock, the reset is also a global signal that is usually connected to all storage elements in a design or module. On many chips the reset is a physical external pin that can, for example, be connected to a push button.

Reset Signal

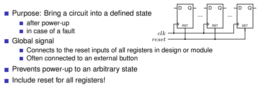
- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button



Sequential Circuit Elements in VHDL

Introduction

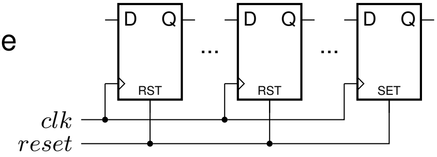
Reset Signal



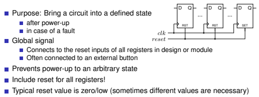
- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!

The reset signal ensures that all storage elements are set to a predefined value, allowing the system to start from a known, stable state. A latch or flip-flop without a reset can power-up to an arbitrary value. Usually this is undesired, which is why it is good practice to reset all sequential elements, even if you consider it unnecessary. This also minimizes the potential for divergence between the behavior of the simulation and hardware.

Reset Signal



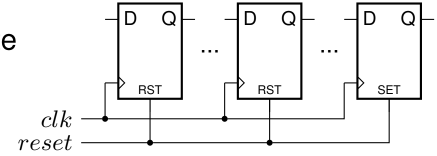
- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!



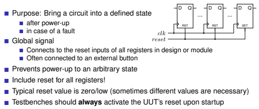
- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!
- Typical reset value is zero/low (sometimes different values are necessary)

The typical reset value for storage elements is zero. However, this is not a strict rule, as sometimes it is also necessary to initialize storage elements to one. As shown by the right-most flip-flop in the drawing, we will mark flip-flops and latches whose reset value is one by labelling their reset inputs with SET instead of RST.

Reset Signal

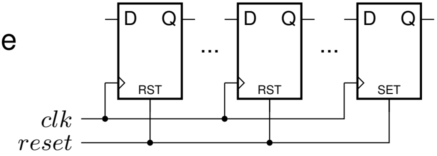


- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!
- Typical reset value is zero/low (sometimes different values are necessary)

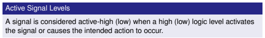

- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!
- Typical reset value is zero/low (sometimes different values are necessary)
- Testbenches should **always** activate the UUT's reset upon startup

Note that testbenches must always start by activating the reset, bringing the UUT into a defined state before applying test stimuli. While the reset is active all input signals to the UUT should also be set to defined values such that when the reset signal is released the design only sees valid input values.

Reset Signal

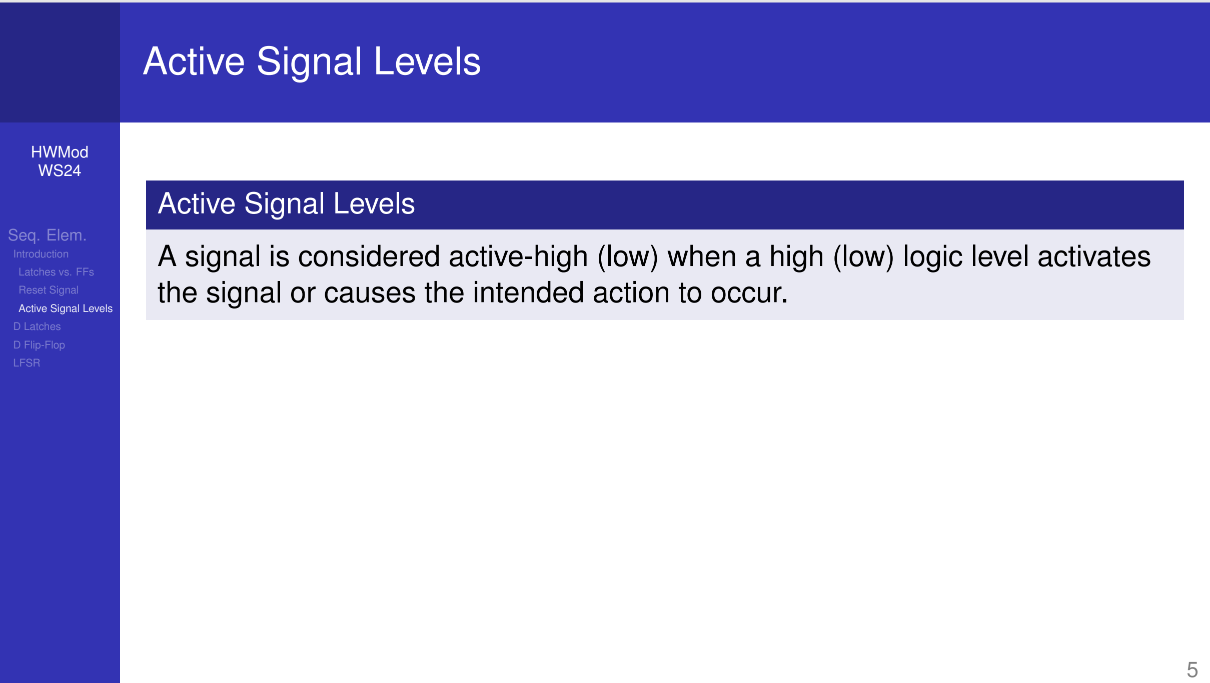


- Purpose: Bring a circuit into a defined state
  - after power-up
  - in case of a fault
- Global signal
  - Connects to the reset inputs of all registers in design or module
  - Often connected to an external button
- Prevents power-up to an arbitrary state
- Include reset for all registers!
- Typical reset value is zero/low (sometimes different values are necessary)
- Testbenches should **always** activate the UUT's reset upon startup



Active Signal Levels
A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.

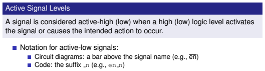
Before we look at some VHDL code examples we have to introduce the important concept of active signal levels, referring to the specific logic level at which a signal is considered ”active” or ”asserted”. In binary digital systems, signals can either be active-high or active-low. An active-high signal is active when it is at a logical one, typically represented by a high voltage level. Conversely, an active-low signal is considered active when its logical value is zero.



Active Signal Levels

Active Signal Levels

A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.


Active Signal Levels

A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.

- Notation for active-low signals:
  - Circuit diagrams: a bar above the signal name (e.g., $\overline{\text{en}}$)
  - Code: the suffix `_n` (e.g., `en_n`)

To clearly distinguish active-low from active-high signals, we mark signals in circuit diagrams with a horizontal bar above the signal name and identifiers in source code with the suffix _n.

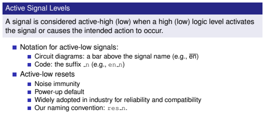
Active Signal Levels

Seq. Elem.
Introduction
Latches vs. FFs
Reset Signal
Active Signal Levels
D Latches
D Flip-Flop
LFSR

Active Signal Levels

A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.

- Notation for active-low signals:
  - Circuit diagrams: a bar above the signal name (e.g., $\overline{\text{en}}$)
  - Code: the suffix `_n` (e.g., `en_n`)


Active Signal Levels
A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.
- Notation for active-low signals:
  - Circuit diagrams: a bar above the signal name (e.g., $\overline{\text{en}}$)
  - Code: the suffix `_n` (e.g., `en_n`)
- Active-low resets
  - Noise immunity
  - Power-up default
  - Widely adopted in industry for reliability and compatibility
  - Our naming convention: `res_n`.

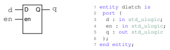
The first time you will come into touch with an active-low signal is the reset signal. In digital designs the reset is often implemented in an active-low way, as this comes with several benefits. This design choice ensures that the system starts in a predictable state during power-up, as logic lines tend to default to low voltage, often called ground. Additionally, active-low resets are more robust in noisy environments, because electrical interference is less likely to falsely trigger a low signal than a high one. These characteristics, combined with the historical adoption of active-low conventions in TTL and CMOS technologies, make it a reliable practice and industry-standard. We will therefore exclusively use active-low resets, referring to this signal as `res_n`. With that being said, we have everything we need to implement our first D latch in VHDL. We start with the latch as it is arguably the simpler state holding element.

Active Signal Levels

Active Signal Levels

A signal is considered active-high (low) when a high (low) logic level activates the signal or causes the intended action to occur.

- Notation for active-low signals:
  - Circuit diagrams: a bar above the signal name (e.g., $\overline{\text{en}}$)
  - Code: the suffix `_n` (e.g., `en_n`)
- Active-low resets
  - Noise immunity
  - Power-up default
  - Widely adopted in industry for reliability and compatibility
  - Our naming convention: `res_n`.

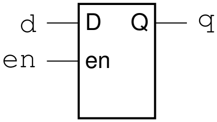


The entity declaration of a D latch is quite straight-forward. We need two inputs `d` and `en` as well as a single output `q` .

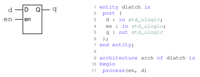
D Latches

HWMod
WS24

Seq. Elem.
Introduction
D Latches
Reset
D Flip-Flop
LFSR



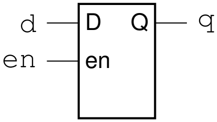
```vhdl
entity dlatch is
 port (
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```



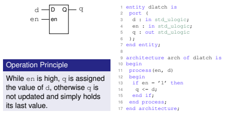
```vhdl
entity dlatch is
 port (
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch is
begin
 process(en, d)
```

In the architecture we use a process that is sensitive to both inputs.

D Latches

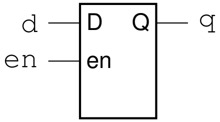


```vhdl
entity dlatch is
 port (
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch is
begin
 process(en, d)
```



In the process body we need a single if-condition, that checks whether the enable signal is asserted and assigns the signal `d` to `q` if that's the case. Should this condition be false, the signal value of `q` is not updated, meaning that it will retain its previous value. Notice that this exactly captures the behavioral specification of the D latch from the introduction.

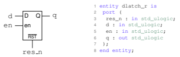
D Latches



Operation Principle

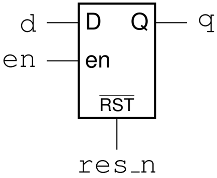
While `en` is high, `q` is assigned the value of `d`, otherwise `q` is not updated and simply holds its last value.

```vhdl
entity dlatch is
 port (
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch is
begin
 process(en, d)
 begin
  if en = '1' then
   q <= d;
  end if;
 end process;
end architecture;
```

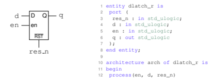


To implement a reset mechanism for our D latch, we first extend its interface by a reset input. Naturally, we use an active-low reset.

D Latch - Reset



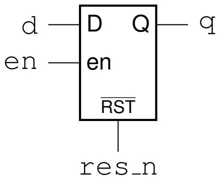
```vhdl
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```



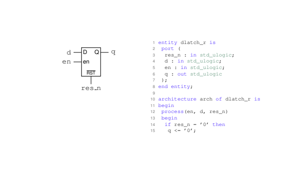
```vhdl
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch_r is
begin
 process(en, d, res_n)
```

In our latch process we first have to extend the sensitivity list to also include the reset input.

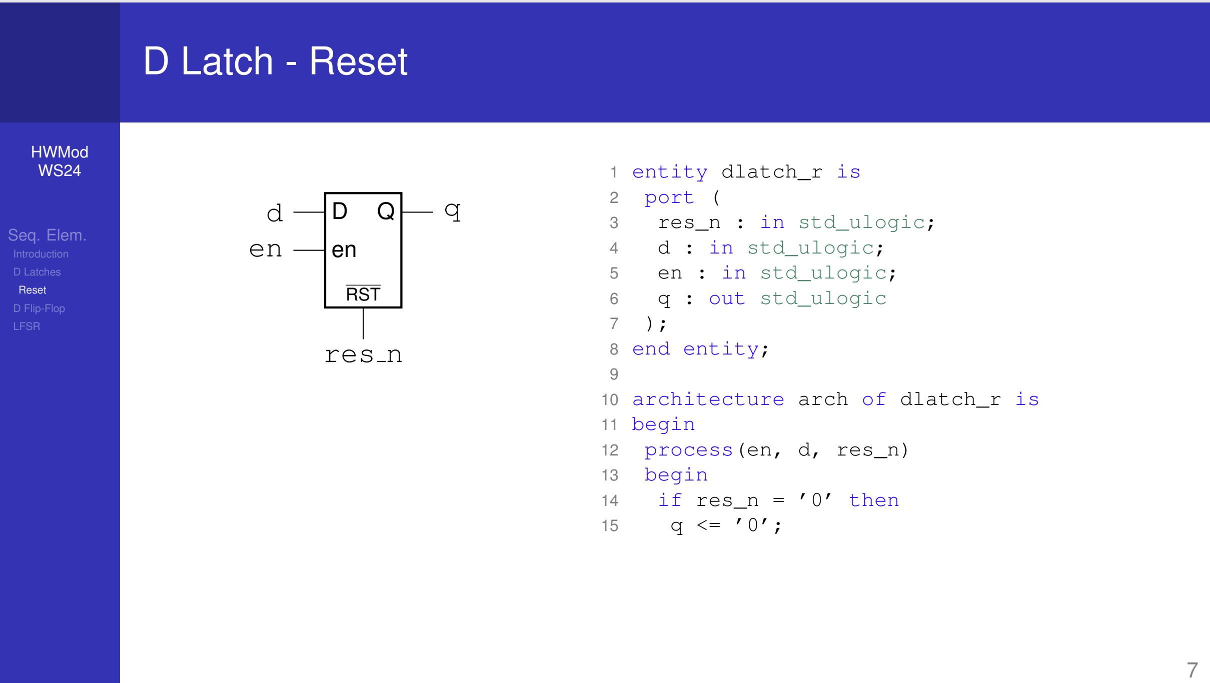
D Latch - Reset



```vhdl
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch_r is
begin
 process(en, d, res_n)
```



Then we introduce the reset condition. Since the reset signal is active-low, we need to test it for the value zero. If this condition is true, the output q is simply set to zero.



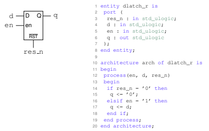
D Latch - Reset

HWMod
WS24

Seq. Elem.
Introduction
D Latches
Reset
D Flip-Flop
LFSR

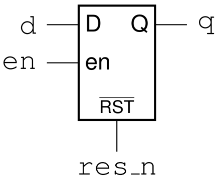
d
en
D
Q
q
en
RST
res_n

```
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch_r is
begin
 process(en, d, res_n)
 begin
  if res_n = '0' then
   q <= '0';
```

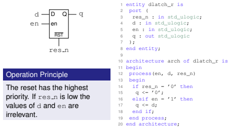


The code that tests the enable signal goes in the else-if-branch. This means that the enable signal is only evaluated, if the reset is not active. Hence, the reset always overrides the internal state of the latch.

D Latch - Reset



```
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch_r is
begin
 process(en, d, res_n)
 begin
  if res_n = '0' then
   q <= '0';
  elsif en = '1' then
   q <= d;
  end if;
 end process;
end architecture;
```

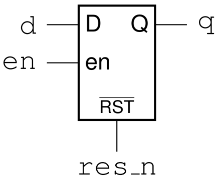

Operation Principle

The reset has the highest priority. If res_n is low the values of d and en are irrelevant.

D Latch - Reset

HWMod
WS24

Seq. Elem.
Introduction
D Latches
Reset
D Flip-Flop
LFSR

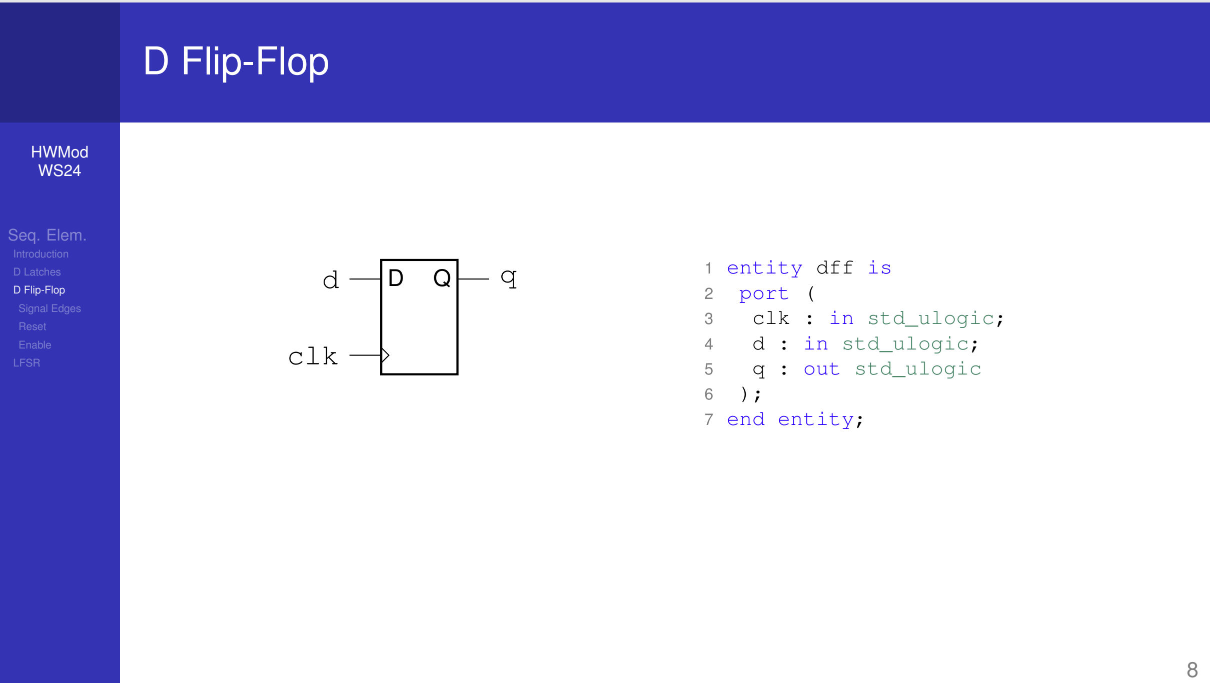


Operation Principle

The reset has the highest priority. If `res_n` is low the values of `d` and `en` are irrelevant.

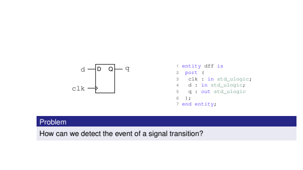
```vhdl
entity dlatch_r is
 port (
  res_n : in std_ulogic;
  d : in std_ulogic;
  en : in std_ulogic;
  q : out std_ulogic
 );
end entity;

architecture arch of dlatch_r is
begin
 process(en, d, res_n)
 begin
  if res_n = ’0’ then
   q <= ’0’;
  elsif en = ’1’ then
   q <= d;
  end if;
 end process;
end architecture;
```

Let us now turn our attention to the D flip-flop. Like with the D latch, we need the input `d` as well as the output `q` . However, instead of the enable signal the D flip-flop needs a clock input to trigger its operation.

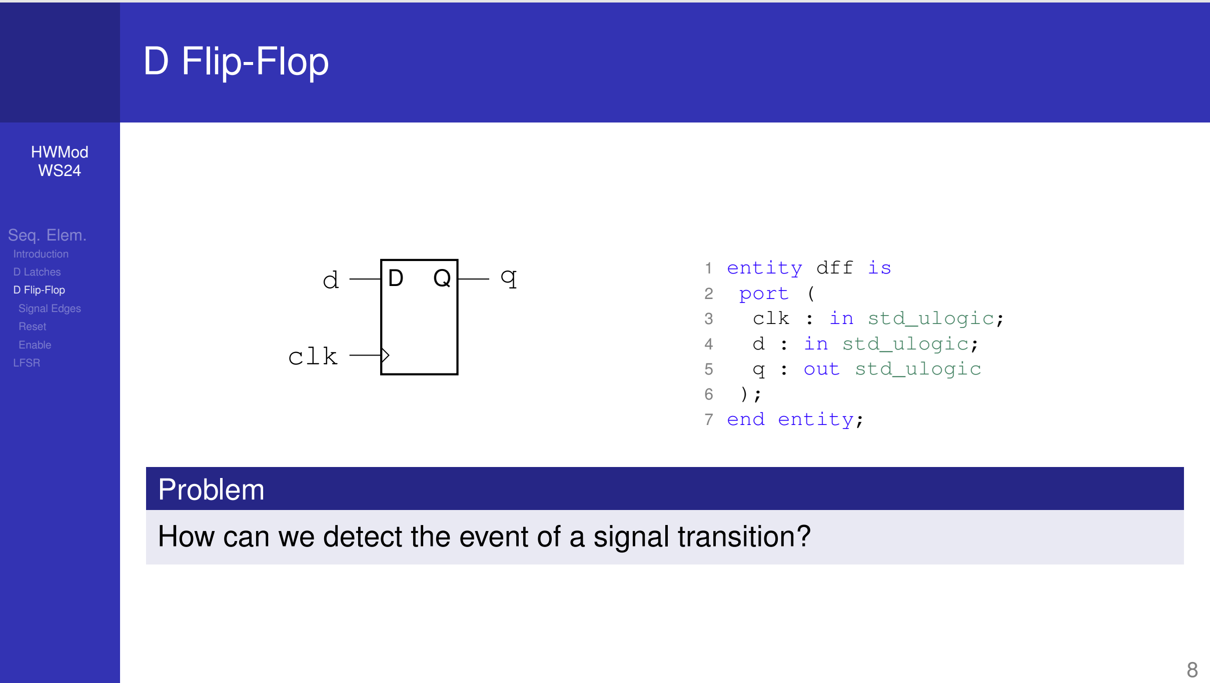


D Flip-Flop

```vhdl
entity dff is
 port (
  clk : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```



Furthermore, we need to react to signal events, that is transitions, rather than the state of a signal. This requires the introduction of a new VHDL language feature.



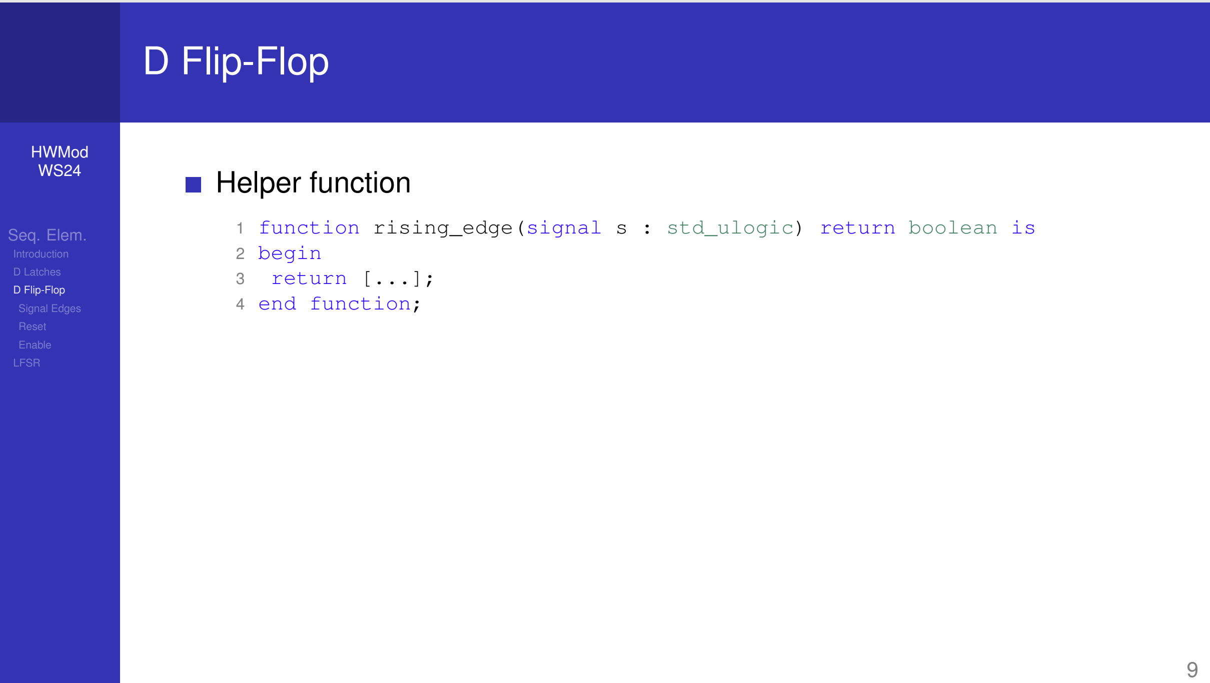
D Flip-Flop

```vhdl
entity dff is
 port (
  clk : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```

Problem

How can we detect the event of a signal transition?

Let’s say we have a helper function `rising_edge` that takes a signal of type `std_ulogic` and returns a Boolean. The function shall only return true when a rising edge has just occurred on signal S and false otherwise. On the following slides we will investigate how we can implement such a function in VHDL.



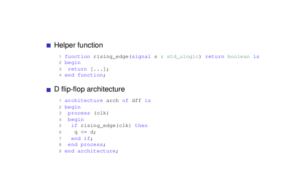
D Flip-Flop

HWMod
WS24

Seq. Elem.
Introduction
D Latches
D Flip-Flop
Signal Edges
Reset
Enable
LFSR

- Helper function

```vhdl
function rising_edge(signal s : std_ulogic) return boolean is
begin
 return [...];
end function;
```



Helper function

```
function rising_edge(signal s : std_ulogic) return boolean is
begin
 return [...];
end function;
```

D flip-flop architecture

```
architecture arch of dff is
begin
 process (clk)
 begin
  if rising_edge(clk) then
   q <= d;
  end if;
 end process;
end architecture;
```

Using this helper function we can then implement the architecture of the D flip-flop in a quite straight-forward way. As with the D latch we first create a process. However, now, we only need the clock signal in the sensitivity list as the process only needs to be evaluated when an event happens on the clock signal. Then we use an if-condition and our helper function to check if a rising clock edge occurred. If this is the case we assign `d` to `q`. Now the only question that remains is how the body of the `rising_edge` function can be implemented. In other words: What expression must be inserted instead of the place-holder in the function body?

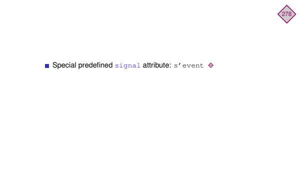
D Flip-Flop

- Helper function

```
function rising_edge(signal s : std_ulogic) return boolean is
begin
 return [...];
end function;
```

- D flip-flop architecture

```
architecture arch of dff is
begin
 process (clk)
 begin
  if rising_edge(clk) then
   q <= d;
  end if;
 end process;
end architecture;
```



Special predefined `signal` attribute: `s’event`

Luckily the VHDL standard defines the ”event” attribute for signals which we can use for our purpose.

Signal Edges – `event` Attribute

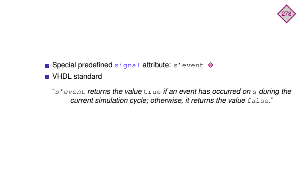
278

- Special predefined `signal` attribute: `s’event` 278

Sequential Circuit Elements in VHDL

D Flip-Flop

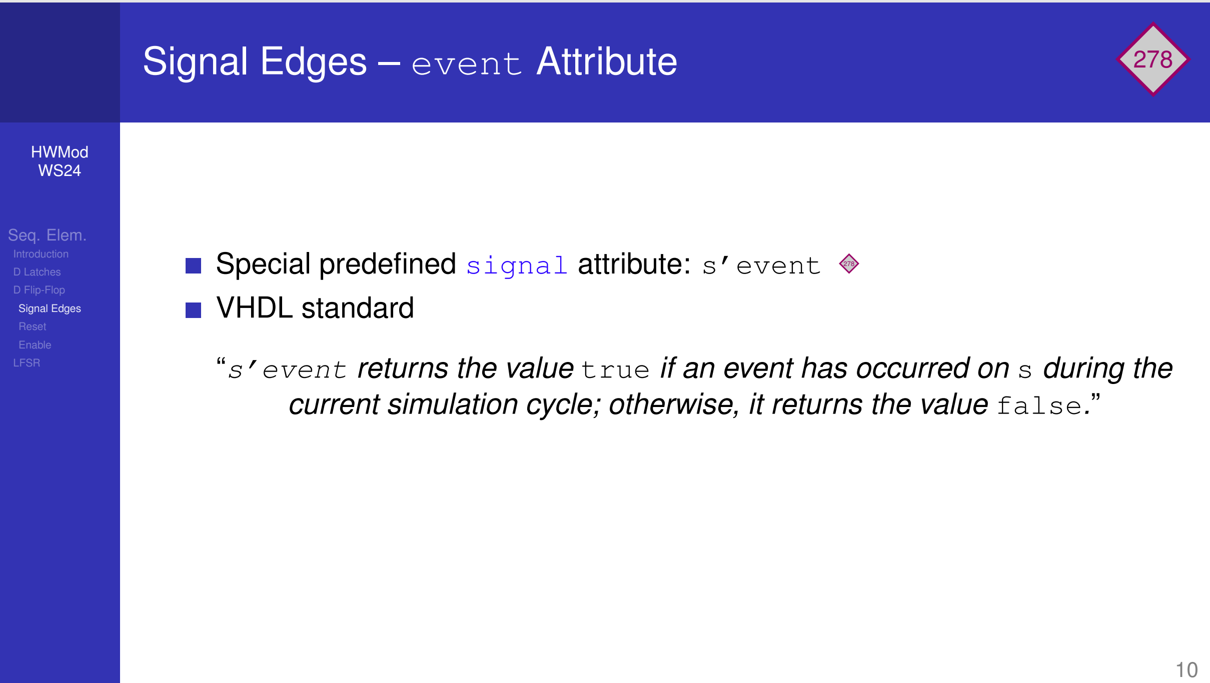
Signal Edges – `event` Attribute



- Special predefined `signal` attribute: `s'event`
- VHDL standard

  "*`s'event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*"

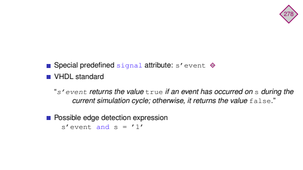
As stated by the short excerpt from the VHDL standard shown on the slide, the event attribute returns the Boolean value true whenever there occurred an event during the current simulation cycle. If no event occurred, the attribute will return false.



Signal Edges – `event` Attribute

- Special predefined `signal` attribute: `s'event`
- VHDL standard

  "*`s'event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*"



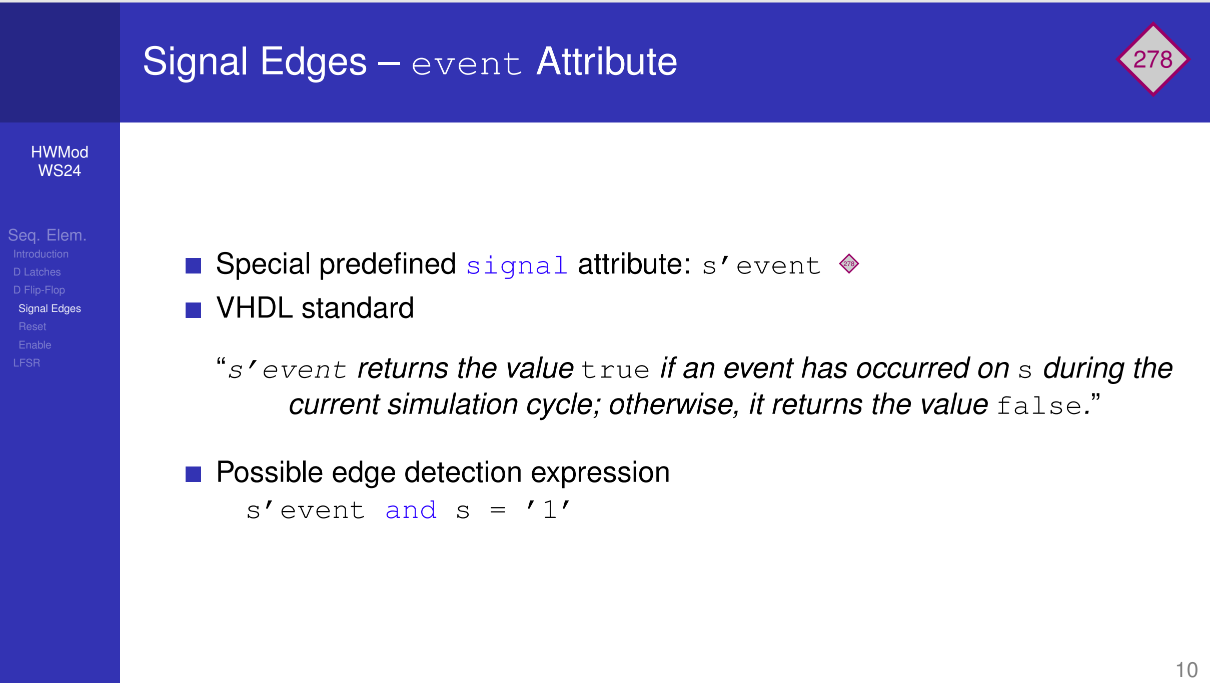
278

- Special predefined `signal` attribute: `s’event`
- VHDL standard

“*`s’event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*”

- Possible edge detection expression
  `s’event and s = ’1’`

We can harness this attribute to express a simple condition for detecting rising edges, as shown on the slide. This Boolean expression will evaluate to true whenever the signal `s` transitioned to one in the current cycle. You might want to pause the video at this point and think about whether this expression leads to undesired behavior.



Signal Edges – `event` Attribute

278

HWMod
WS24

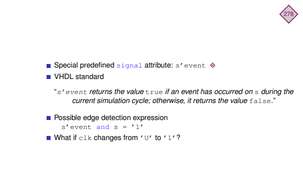
Seq. Elem.
Introduction
D Latches
D Flip-Flop
Signal Edges
Reset
Enable
LFSR

- Special predefined `signal` attribute: `s’event`
- VHDL standard

“*`s’event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*”

- Possible edge detection expression

```
s’event and s = ’1’
```

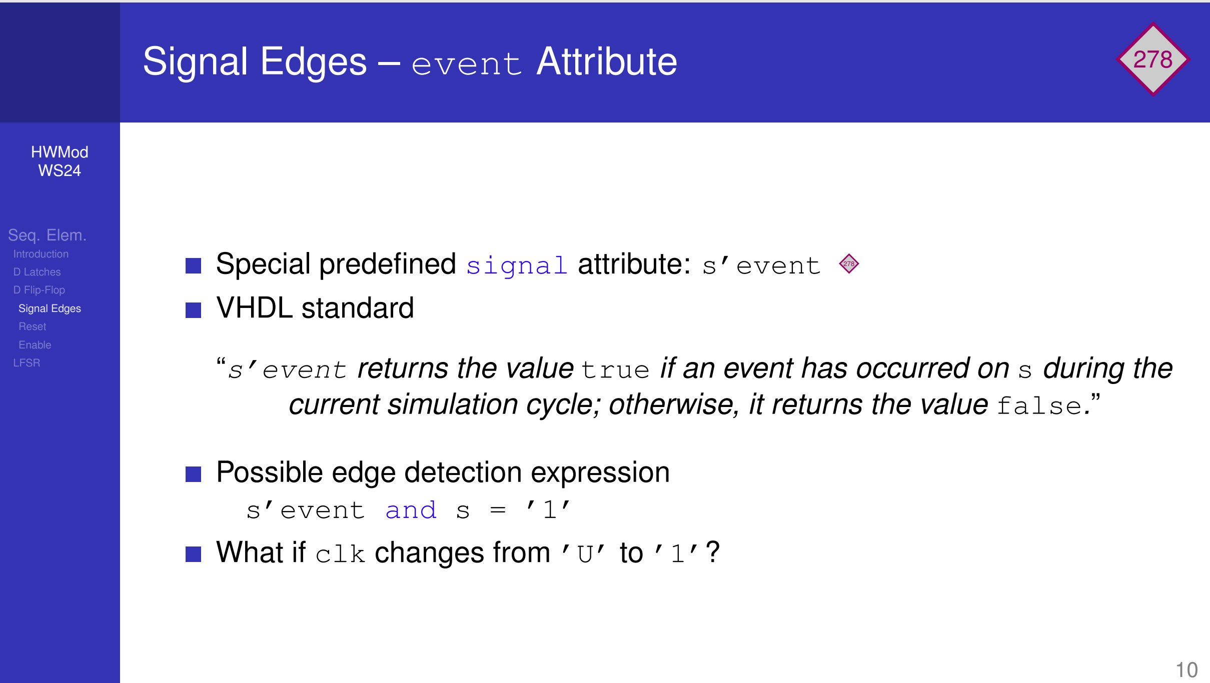

278

- Special predefined `signal` attribute: `s'event`
- VHDL standard

  "*`s'event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*"

- Possible edge detection expression
  `s'event and s = '1'`
- What if `clk` changes from `'U'` to `'1'`?

As you probably detected yourself, a potential issue of this expression is that it detects arbitrary changes to the value one. For example, it will also evaluate to true when the clock signal changes from uninitialized to 1. Naturally, this does not correspond to a rising clock edge. How can we deal with this issue?



Signal Edges – `event` Attribute

278

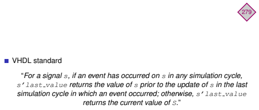
- Special predefined `signal` attribute: `s'event`
- VHDL standard

  "*`s'event` returns the value `true` if an event has occurred on `s` during the current simulation cycle; otherwise, it returns the value `false`.*"

- Possible edge detection expression

  ```
  s'event and s = '1'
  ```

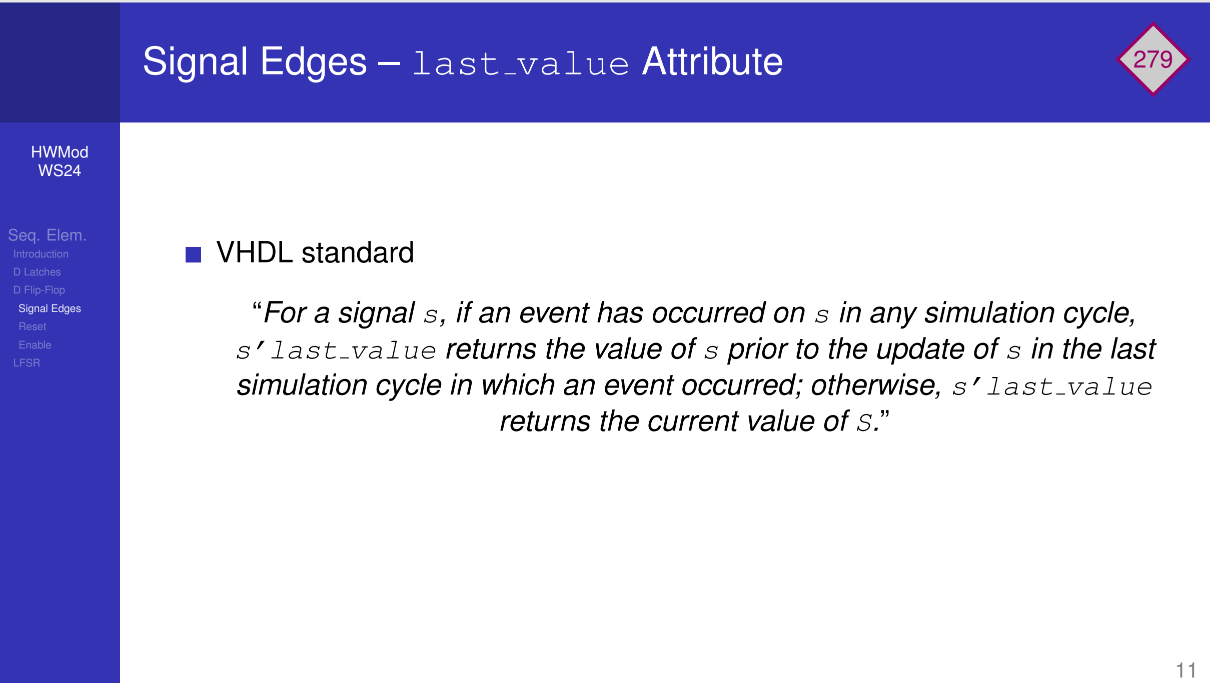
- What if `clk` changes from `'U'` to `'1'`?



■ VHDL standard

"*For a signal* `s`*, if an event has occurred on* `s` *in any simulation cycle,* `s'last_value` *returns the value of* `s` *prior to the update of* `s` *in the last simulation cycle in which an event occurred; otherwise,* `s'last_value` *returns the current value of* `S`*.*"

Again, the VHDL standard comes to the rescue with an attribute. In case an event already occured on a signal `s`, the `last_value` attribute returns the value of the signal before the most recent event.



Signal Edges – `last_value` Attribute

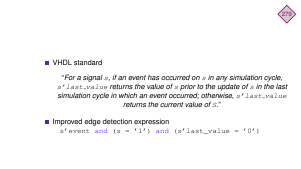
279

HWMod
WS24

Seq. Elem.
Introduction
D Latches
D Flip-Flop
Signal Edges
Reset
Enable
LFSR

- VHDL standard

> "*For a signal* `s`*, if an event has occurred on* `s` *in any simulation cycle,* `s'last_value` *returns the value of* `s` *prior to the update of* `s` *in the last simulation cycle in which an event occurred; otherwise,* `s'last_value` *returns the current value of* `S`*.*"



- VHDL standard

  “*For a signal s, if an event has occurred on s in any simulation cycle, s’last_value returns the value of s prior to the update of s in the last simulation cycle in which an event occurred; otherwise, s’last_value returns the current value of S.*”

- Improved edge detection expression

  s’event and (s = ’1’) and (s’last_value = ’0’)

We can use this attribute to refine our edge detection expression such that it only evaluates to true whenever an event occurred on the respective signal and this event led to a transition from the value zero to one. So, are we done? Is this expression valid for our purpose?

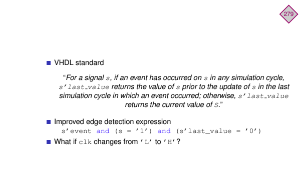
Signal Edges – `last_value` Attribute

- VHDL standard

  “*For a signal `s`, if an event has occurred on `s` in any simulation cycle, `s’last_value` returns the value of `s` prior to the update of `s` in the last simulation cycle in which an event occurred; otherwise, `s’last_value` returns the current value of `S`.*”

- Improved edge detection expression

```
s’event and (s = ’1’) and (s’last_value = ’0’)
```

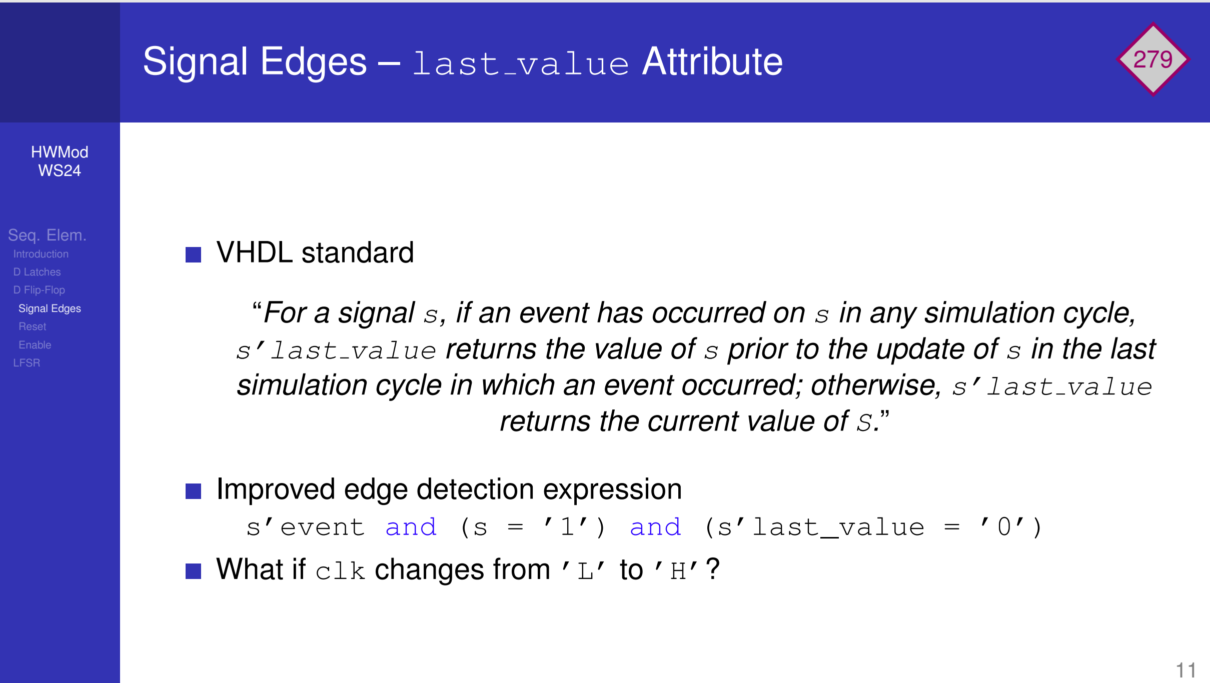


- VHDL standard

  "For a signal `s`, if an event has occurred on `s` in any simulation cycle, `s'last_value` returns the value of `s` prior to the update of `s` in the last simulation cycle in which an event occurred; otherwise, `s'last_value` returns the current value of `s`."

- Improved edge detection expression
  `s'event and (s = '1') and (s'last_value = '0')`
- What if `clk` changes from `'L'` to `'H'`?

Unfortunately not. The improved edge detection expression still comes with potential issues. Recall that our signals are in most cases actually not binary, but rather can take nine distinct values in the case of `std_ulogic`. Within these nine values, the transition from zero to one is not the only valid rising edge. For example, a transition from weak low to weak high should also be detected.



Signal Edges – `last_value` Attribute

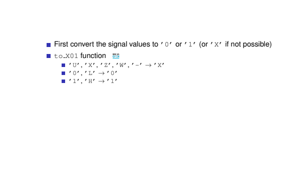
- VHDL standard

  "For a signal `s`, if an event has occurred on `s` in any simulation cycle, `s'last_value` returns the value of `s` prior to the update of `s` in the last simulation cycle in which an event occurred; otherwise, `s'last_value` returns the current value of `s`."

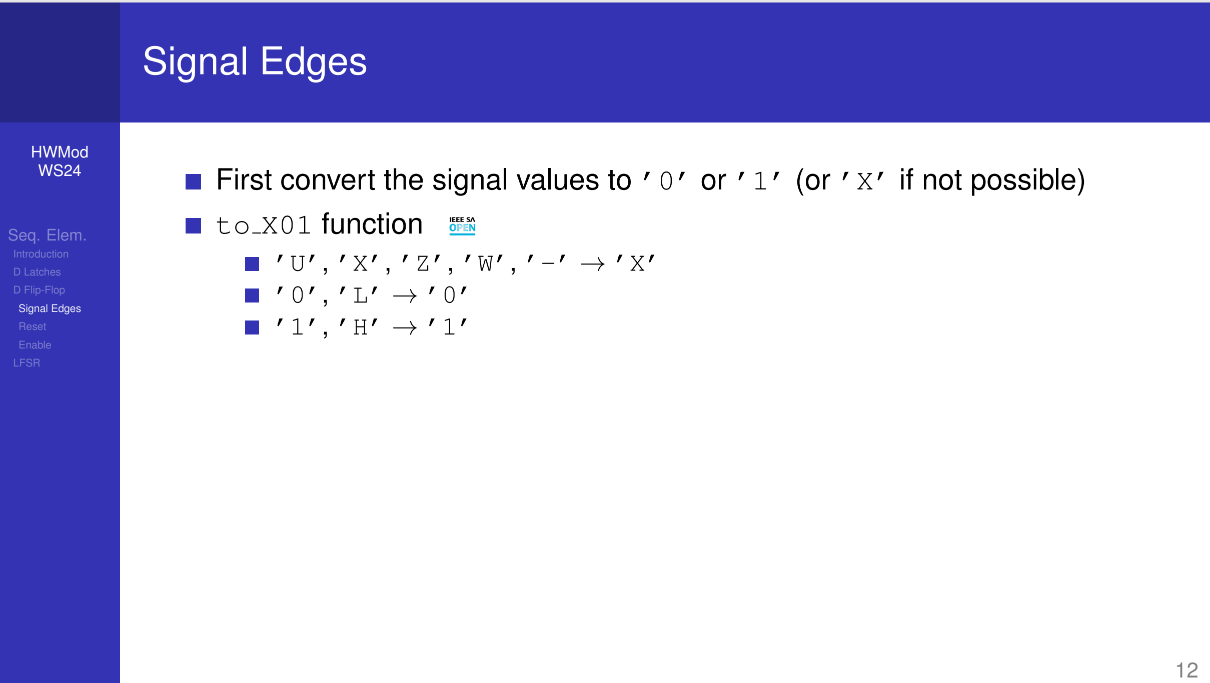
- Improved edge detection expression

  ```
  s'event and (s = '1') and (s'last_value = '0')
  ```

- What if `clk` changes from `'L'` to `'H'`?

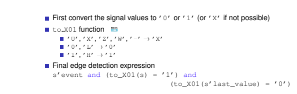

- First convert the signal values to '0' or '1' (or 'X' if not possible)
- to_X01 function
  - 'U','X','Z','W','-' → 'X'
  - '0','L' → '0'
  - '1','H' → '1'

A simple way to handle this problem is by reducing the set of values we consider in our `rising_edge` function by mapping the nine values to their binary equivalents if possible, and otherwise to ”X”. For that purpose the `std_logic_1164` package contains the `to_X01` function, performing the mapping shown on the slide.

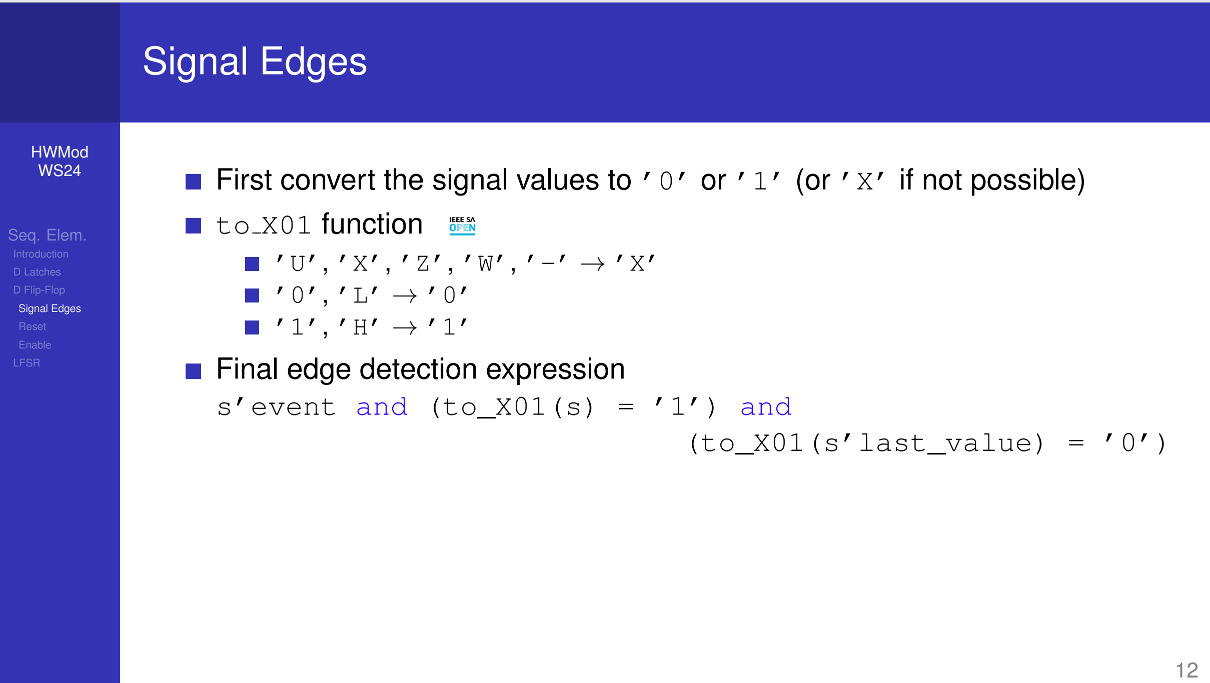


Signal Edges

- First convert the signal values to '0' or '1' (or 'X' if not possible)
- `to_X01` function IEEE SA OPEN
  - 'U','X','Z','W','-' → 'X'
  - '0','L' → '0'
  - '1','H' → '1'


- First convert the signal values to '0' or '1' (or 'X' if not possible)
- to_X01 function
  - 'U','X','Z','W','-' → 'X'
  - '0','L' → '0'
  - '1','H' → '1'
- Final edge detection expression
  s'event and (to_X01(s) = '1') and (to_X01(s'last_value) = '0')

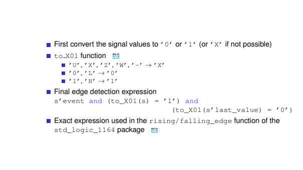
Using this function and previously introduced attributes we can now finally write down an expression to detect rising edges of a signal. This expression is simply the one from the previous slide with the current and previous signal values being mapped to '0','1','X'.



Signal Edges

- First convert the signal values to '0' or '1' (or 'X' if not possible)
- to_X01 function IEEE SA OPEN
  - 'U','X','Z','W','-' → 'X'
  - '0','L' → '0'
  - '1','H' → '1'
- Final edge detection expression

```
s'event and (to_X01(s) = '1') and
                   (to_X01(s'last_value) = '0')
```

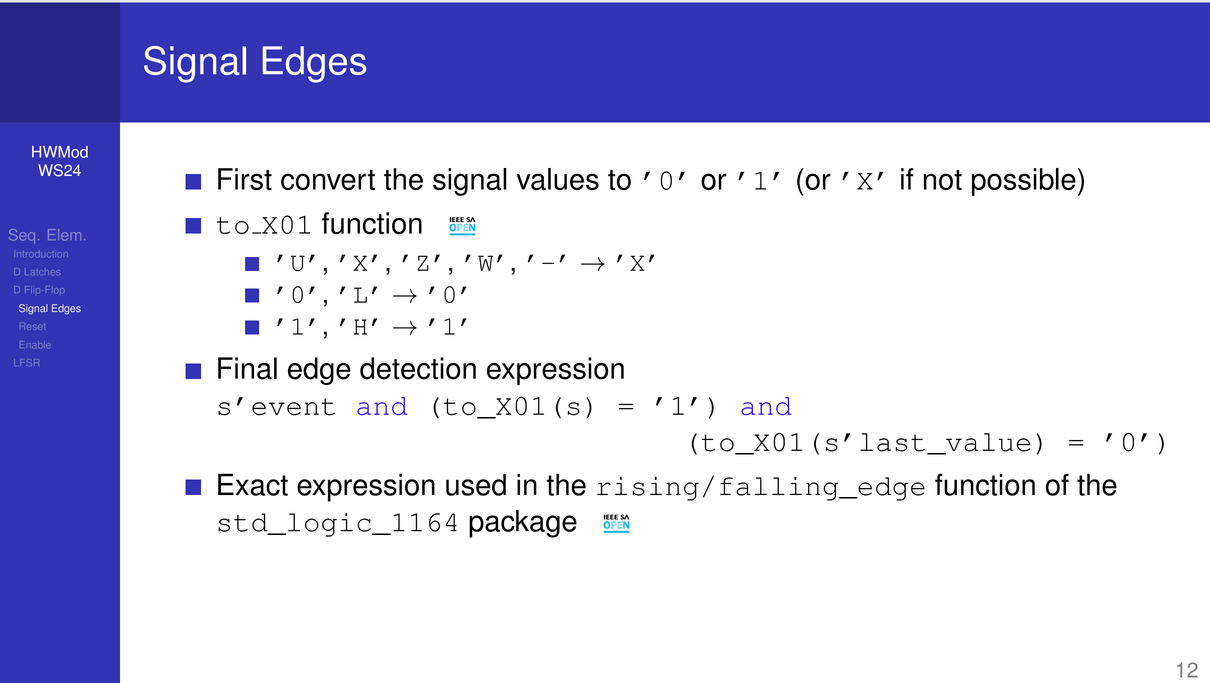


- First convert the signal values to `'0'` or `'1'` (or `'X'` if not possible)
- `to_X01` function
  - `'U','X','Z','W','-'` → `'X'`
  - `'0','L'` → `'0'`
  - `'1','H'` → `'1'`
- Final edge detection expression

```
s'event and (to_X01(s) = '1') and
                        (to_X01(s'last_value) = '0')
```

- Exact expression used in the `rising/falling_edge` function of the `std_logic_1164` package

This expression is the exact same one as used in the IEEE implementation and is not subject to the issues we mentioned before. For the sake of completeness, note that the IEEE also defines a similar function for detecting falling edges.



Signal Edges

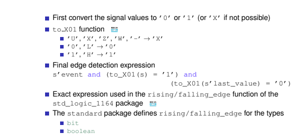
HWMod
WS24

Seq. Elem.
Introduction
D Latches
D Flip-Flop
Signal Edges
Reset
Enable
LFSR

- First convert the signal values to `'0'` or `'1'` (or `'X'` if not possible)
- `to_X01` function
  - `'U','X','Z','W','-'` → `'X'`
  - `'0','L'` → `'0'`
  - `'1','H'` → `'1'`
- Final edge detection expression

```
s'event and (to_X01(s) = '1') and
                        (to_X01(s'last_value) = '0')
```

- Exact expression used in the `rising/falling_edge` function of the `std_logic_1164` package



- First convert the signal values to '0' or '1' (or 'X' if not possible)
- to_X01 function
  - 'U','X','Z','W','-' → 'X'
  - '0','L' → '0'
  - '1','H' → '1'
- Final edge detection expression
  s'event and (to_X01(s) = '1') and (to_X01(s'last_value) = '0')
- Exact expression used in the rising/falling_edge function of the std_logic_1164 package
- The standard package defines rising/falling_edge for the types
  - bit
  - boolean

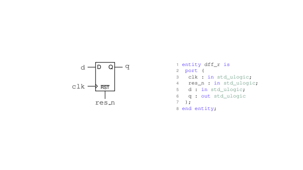
Finally, we want to mention that the rising and falling edge functions are not only defined for signals of the type `std_ulogic`, but also for the `bit` and `boolean` types in the standard package.

Signal Edges

- First convert the signal values to `'0'` or `'1'` (or `'X'` if not possible)
- `to_X01` function
  - `'U','X','Z','W','-'` → `'X'`
  - `'0','L'` → `'0'`
  - `'1','H'` → `'1'`
- Final edge detection expression

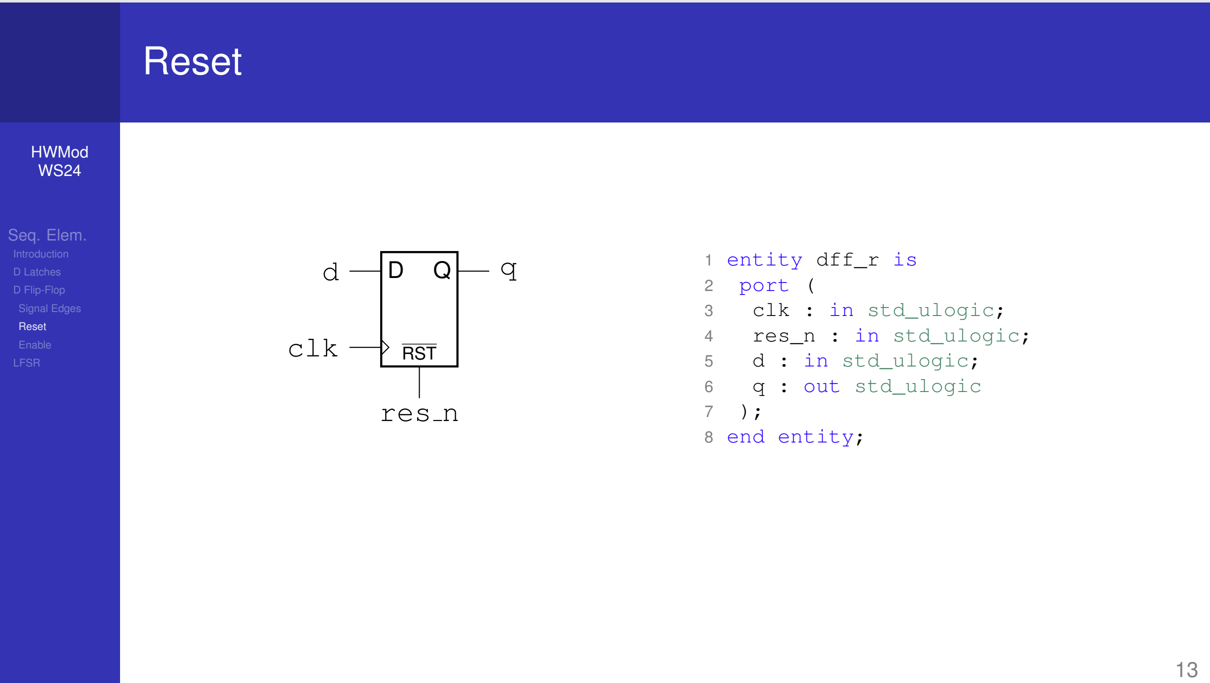
```
s'event and (to_X01(s) = '1') and
                              (to_X01(s'last_value) = '0')
```

- Exact expression used in the `rising/falling_edge` function of the `std_logic_1164` package
- The `standard` package defines `rising/falling_edge` for the types
  - `bit`
  - `boolean`



```vhdl
entity dff_r is
 port (
  clk : in std_ulogic;
  res_n : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```

Just as for the latch, it must be possible to reset a flip-flop. Again, we use a dedicated reset input for that. The slide shows the respective symbol and entity declaration. However, whereas the semantics of the reset are rather obvious for a latch, they are less clear for a flip-flop.

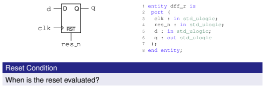


Reset

HWMod
WS24

Seq. Elem.
Introduction
D Latches
D Flip-Flop
Signal Edges
Reset
Enable
LFSR

```vhdl
entity dff_r is
 port (
  clk : in std_ulogic;
  res_n : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```



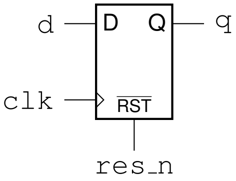
```
entity dff_r is
 port (
  clk : in std_ulogic;
  res_n : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```

Reset Condition

When is the reset evaluated?

In particular: Is the reset evaluated only at the active clock edges of the flip-flop, or is it level-sensitive and resets the flip-flop independent of the clock?

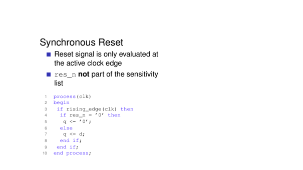
Reset



```
entity dff_r is
 port (
  clk : in std_ulogic;
  res_n : in std_ulogic;
  d : in std_ulogic;
  q : out std_ulogic
 );
end entity;
```

Reset Condition

When is the reset evaluated?


Synchronous Reset
- Reset signal is only evaluated at the active clock edge
- res_n **not** part of the sensitivity list

```
process(clk)
begin
 if rising_edge(clk) then
  if res_n = '0' then
   q <= '0';
  else
   q <= d;
  end if;
 end if;
end process;
```

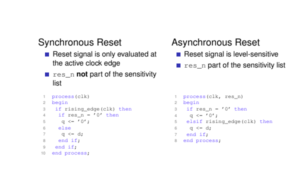
The answer is that both interpretations of the reset are possible. Depending on the desired behavior, a flip-flop can in general be equipped with either reset type. A reset that is only evaluated at the active clock edge is called a synchronous reset for obvious reasons. When describing this in VHDL it is important to omit the reset from the respective sensitivity list, as we only want clock edges to trigger the process. The code snippet on the slide shows this. Also note that the reset condition check is nested within the condition for the rising clock edge.

Reset (Cont'd)

Synchronous Reset

- Reset signal is only evaluated at the active clock edge
- `res_n` **not** part of the sensitivity list

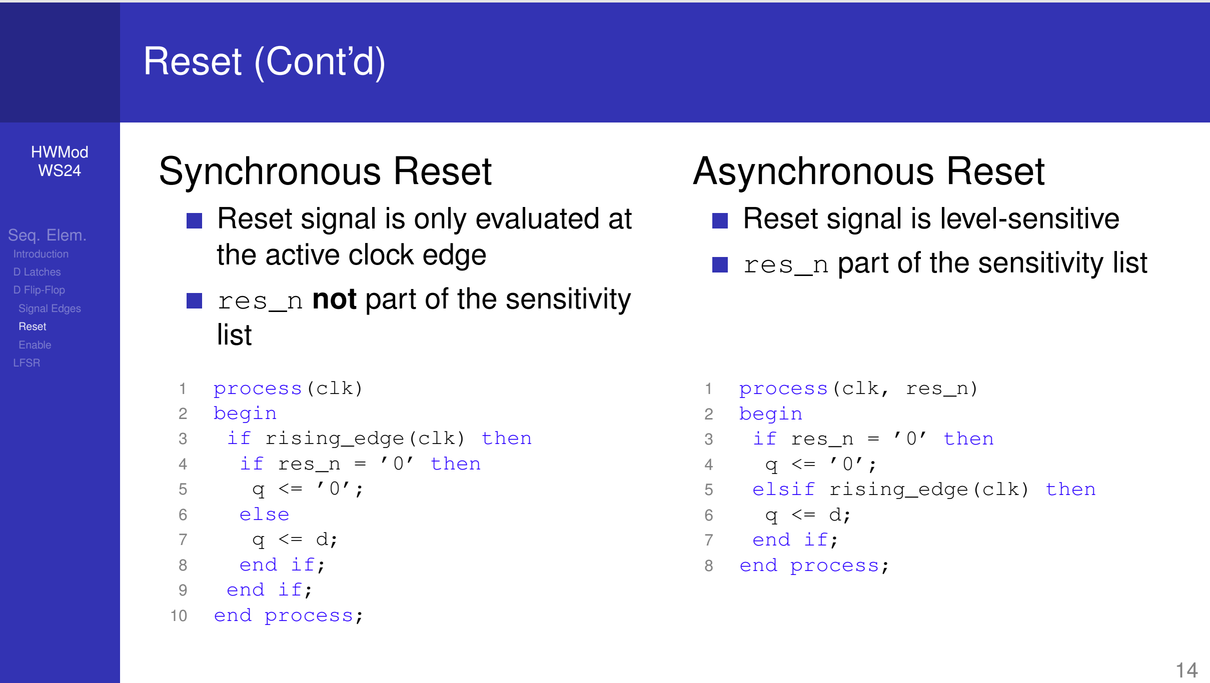
```
process(clk)
begin
 if rising_edge(clk) then
  if res_n = '0' then
   q <= '0';
  else
   q <= d;
  end if;
 end if;
end process;
```



Synchronous Reset
- Reset signal is only evaluated at the active clock edge
- res_n **not** part of the sensitivity list

Asynchronous Reset
- Reset signal is level-sensitive
- res_n part of the sensitivity list

The other flavor of reset is the asynchronous one, which will reset the flip-flop whenever the reset is active, independently of the clock. In the respective VHDL code this is reflected by the reset being included in the sensitivity list. Furthermore, the condition for the reset precedes, and thus dominates, the one for the rising clock edge.



Reset (Cont'd)

Synchronous Reset

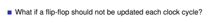
- Reset signal is only evaluated at the active clock edge
- res_n **not** part of the sensitivity list

```
process(clk)
begin
 if rising_edge(clk) then
  if res_n = '0' then
   q <= '0';
  else
   q <= d;
  end if;
 end if;
end process;
```

Asynchronous Reset

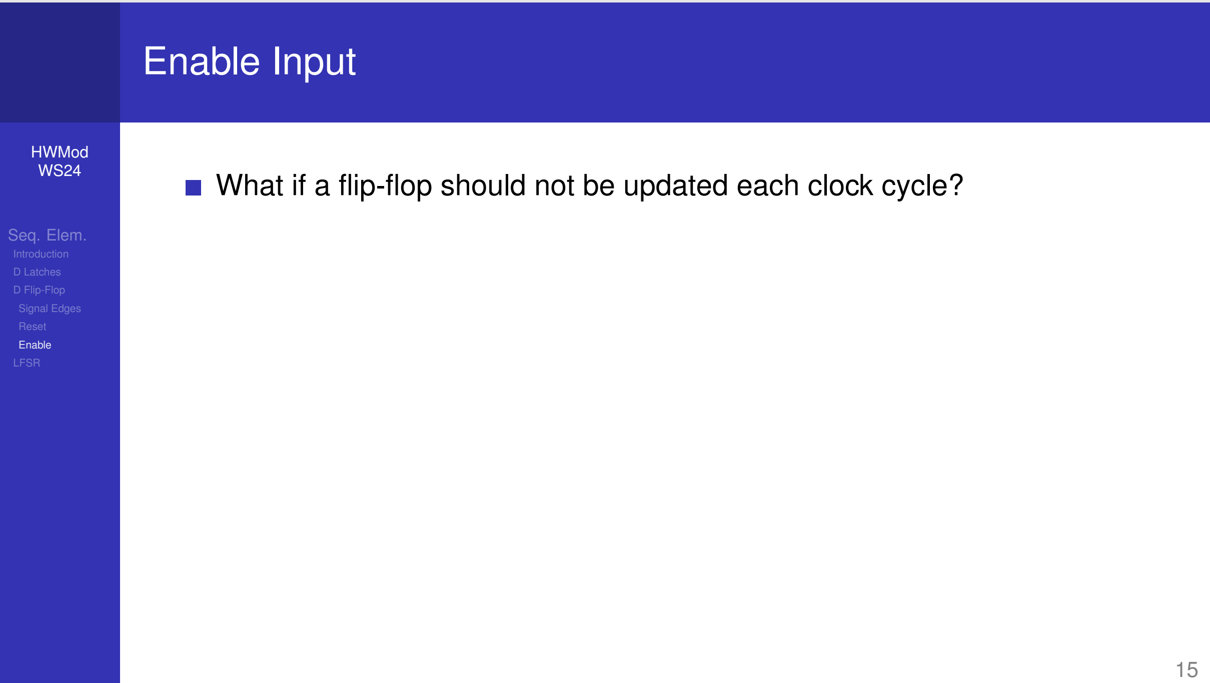
- Reset signal is level-sensitive
- res_n part of the sensitivity list

```
process(clk, res_n)
begin
 if res_n = '0' then
  q <= '0';
 elsif rising_edge(clk) then
  q <= d;
 end if;
end process;
```



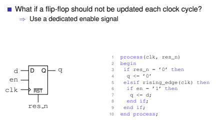
What if a flip-flop should not be updated each clock cycle?

What if we want a flip-flop that does not update its value in each clock cycle? For example, a design might be in an overall state where certain flip-flops or registers must hold their values, although their inputs might change.



Enable Input

- What if a flip-flop should not be updated each clock cycle?



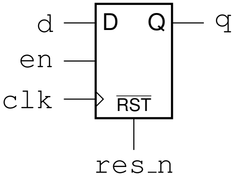
What if a flip-flop should not be updated each clock cycle?
⇒ Use a dedicated enable signal

```vhdl
process(clk, res_n)
begin
 if res_n = '0' then
  q <= '0'
 elsif rising_edge(clk) then
  if en = '1' then
   q <= d;
  end if;
 end if;
end process;
```

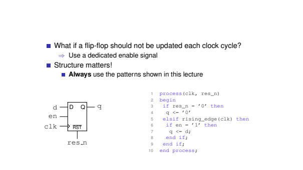
We can achieve this by using a dedicated enable input in addition to the clock, reset and data inputs. A respective flip-flop symbol and VHDL code modeling an asynchronous reset and a synchronous enable are shown on the slide. Note how the condition for the enable signal is contained within the body of the `elsif` branch. Thus, the enable can only take effect at rising clock edges.

Enable Input

- What if a flip-flop should not be updated each clock cycle?
  - ⇒ Use a dedicated enable signal



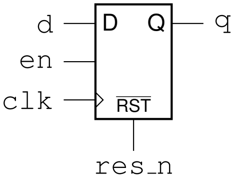
```vhdl
process(clk, res_n)
begin
 if res_n = '0' then
  q <= '0'
 elsif rising_edge(clk) then
  if en = '1' then
   q <= d;
  end if;
 end if;
end process;
```



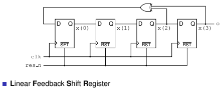
Before we close this lecture with a simple design example, we really want to stress an important point. It is vital that you strictly stick to the code patterns for latches and flip-flops presented in this lecture. Do not experiment with custom flip-flop descriptions! It might be the case that such structures work in simulation but fail to synthesize or, even worse, result in a circuit that does not match the intended design. Synthesis tools look for the code patterns presented here and will – if you stick to them – generate correct circuits! Please always keep that in mind when writing VHDL code!

Enable Input

- What if a flip-flop should not be updated each clock cycle?
  - ⇒ Use a dedicated enable signal
- Structure matters!
  - **Always** use the patterns shown in this lecture

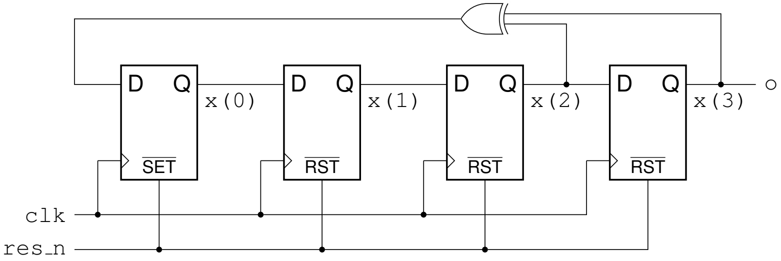


```
process(clk, res_n)
begin
 if res_n = '0' then
  q <= '0'
 elsif rising_edge(clk) then
  if en = '1' then
   q <= d;
  end if;
 end if;
end process;
```

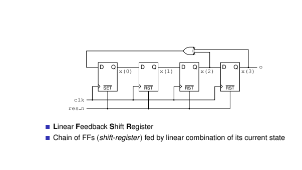


Finally, let us look at a more elaborate example design using the synchronous design style and flip-flops. The slide shows the schematic of a circuit called a linear-feedback-shift-register, abbreviated as ”LFSR”.

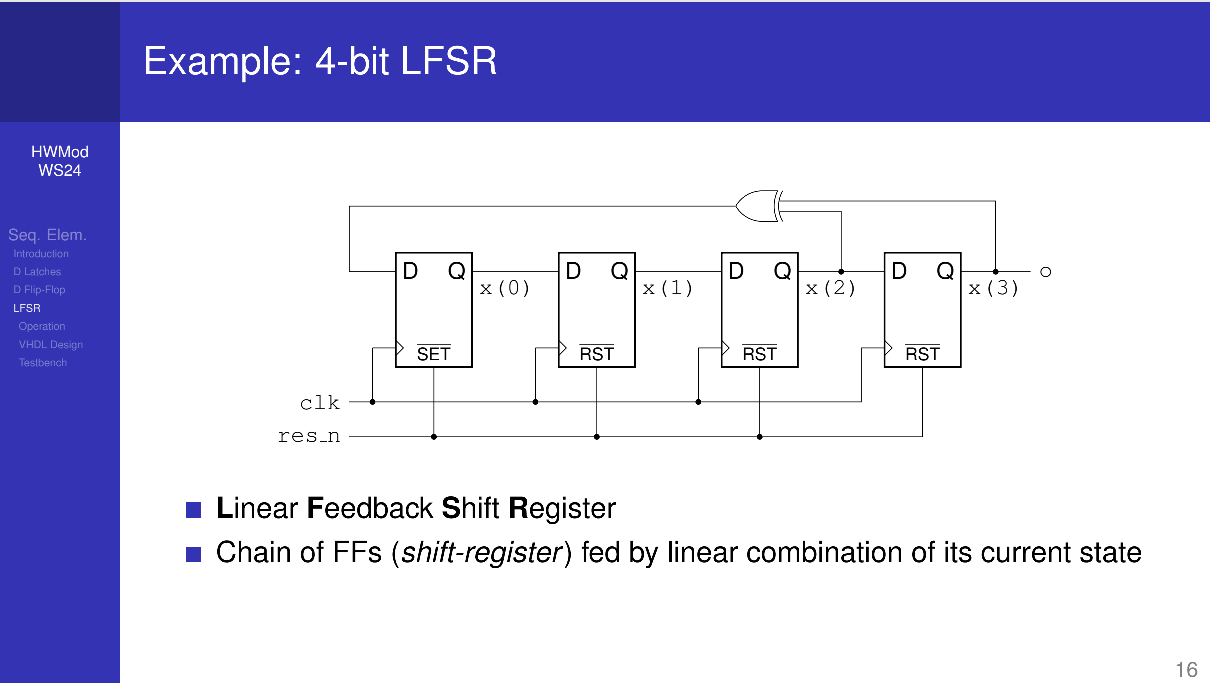
Example: 4-bit LFSR



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



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 is a chain of flip-flops where each flip-flop samples the output of its predecessor, thus essentially shifting the currently stored bits by one flip-flop per clock period.

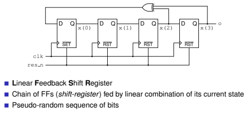


Example: 4-bit LFSR

HWMod
WS24

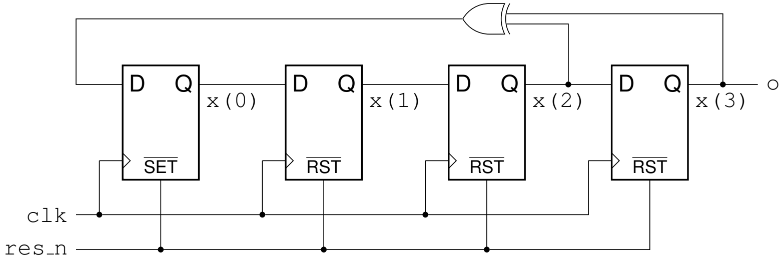
Seq. Elem.
Introduction
D Latches
D Flip-Flop
LFSR
Operation
VHDL Design
Testbench

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

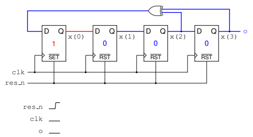


Such circuits are often used as pseudo-random number generators, which means that they can generate a sequence of bits that looks like it is random while it actually is not. We will now briefly demonstrate its operation without getting into details about the involved mathematics.

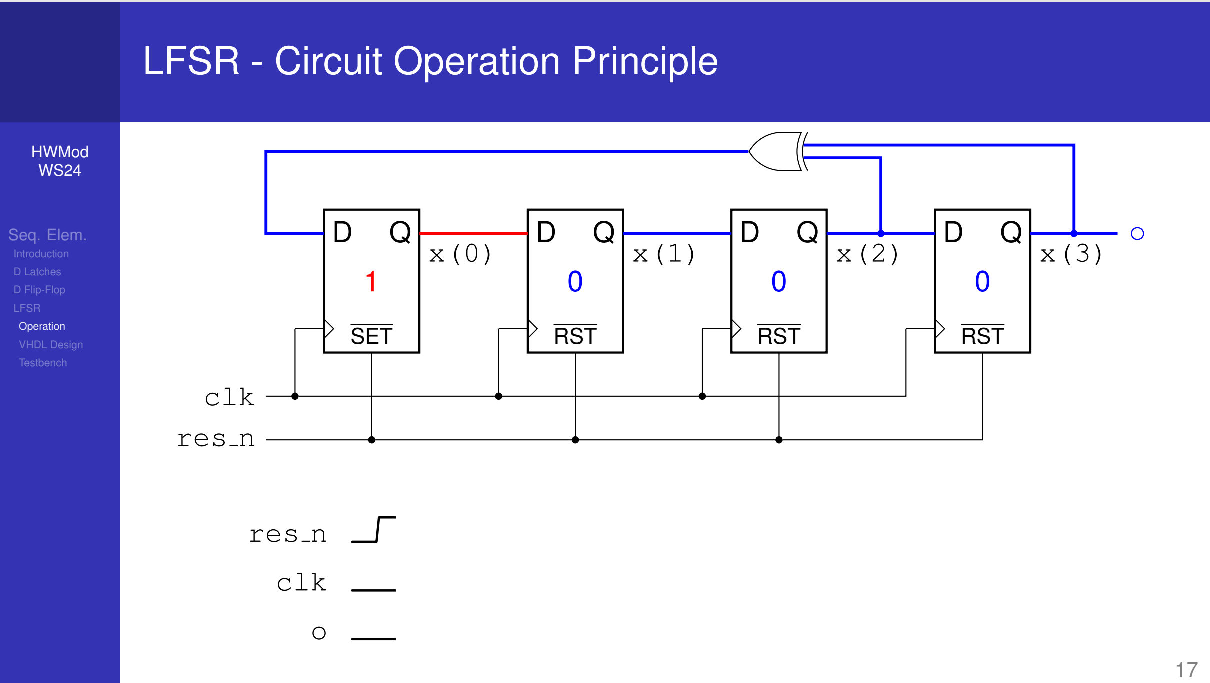
Example: 4-bit LFSR



- **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



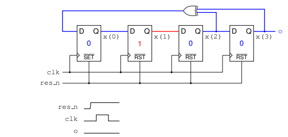
In this illustration of the ”LFSR” the data wires of the circuit are highlighted in blue when propagating a logical zero, and in red for logical ones. We also show the logical value currently held by each flip-flop and a timing diagram of the inputs and outputs. As you already heard, an initial reset is paramount for every state-holding element. Therefore, this is done first leading to almost all flip-flops being set to zero. The first flip-flop, featuring an active-low set signal rather than a reset, is set to 1. This is vital, as all flip-flops being zero would result in the output being constantly zero as well. That’s not very random.



LFSR - Circuit Operation Principle

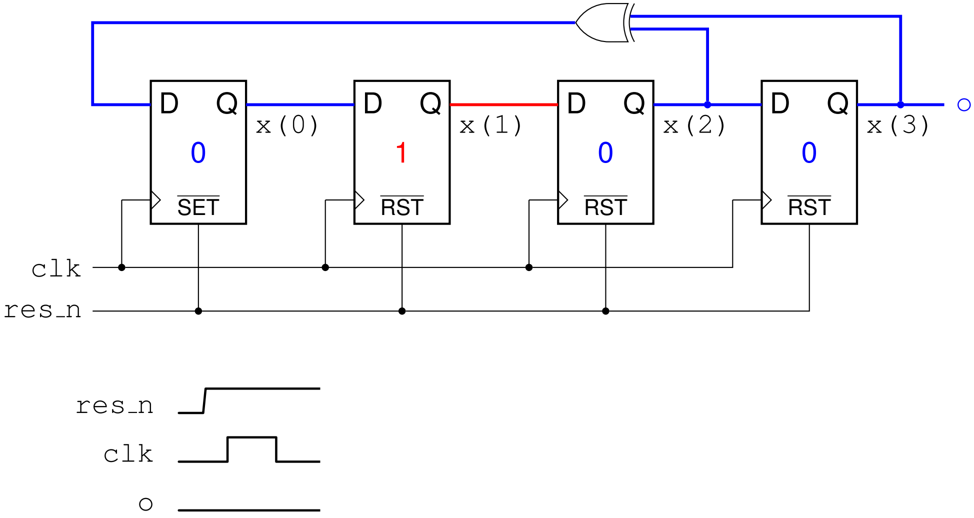
HWMod
WS24

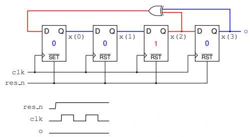
Seq. Elem.
Introduction
D Latches
D Flip-Flop
LFSR
Operation
VHDL Design
Testbench



At the first rising clock edge, all flip-flops sample their inputs, leading to the stored one being shifted to the right and the left-most flip-flop to store zero now.

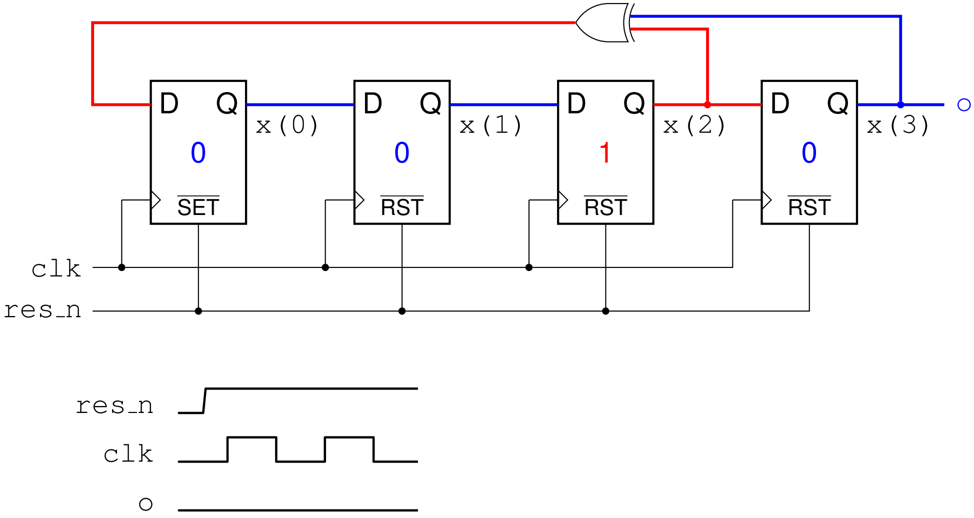
LFSR - Circuit Operation Principle

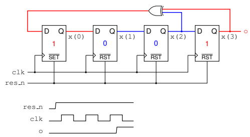




At the next clock edge, the current register bits are shifted again, making the feedback path high.

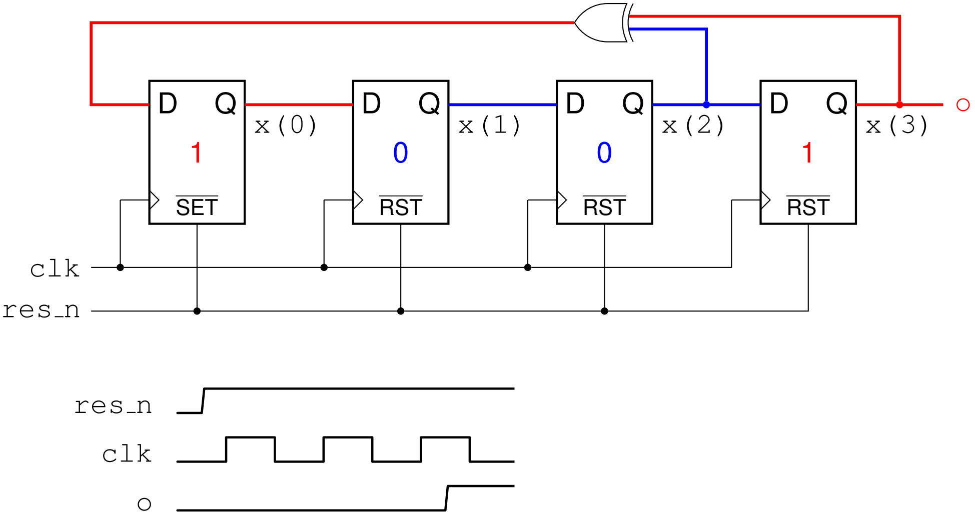
LFSR - Circuit Operation Principle

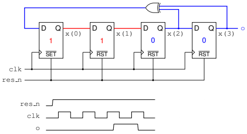




The first flip-flop now sampled the one from the feedback path, while the already stored one shifted into the last flip-flop, asserting the output signal.

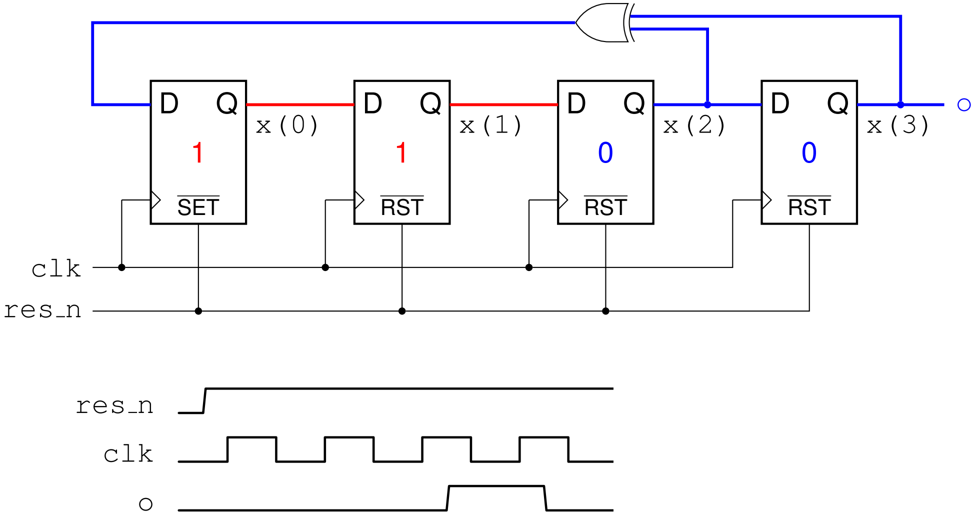
LFSR - Circuit Operation Principle

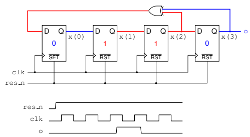




The shifting of bits continues, making the output low again.

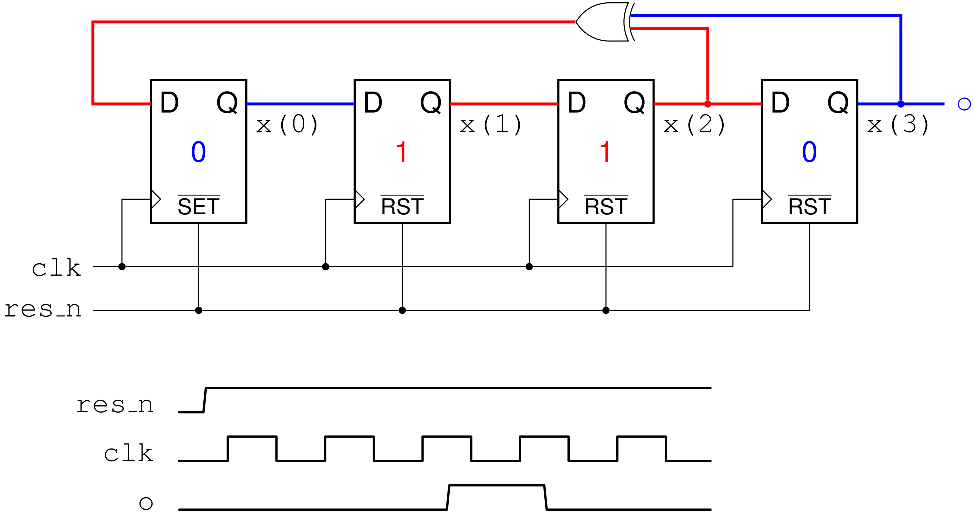
LFSR - Circuit Operation Principle

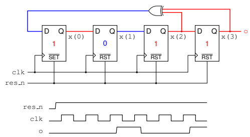




The ”LFSR” continues in this fashion of sampling the feedback and shifting its internal state, producing a sequence of zeros and ones at its output.

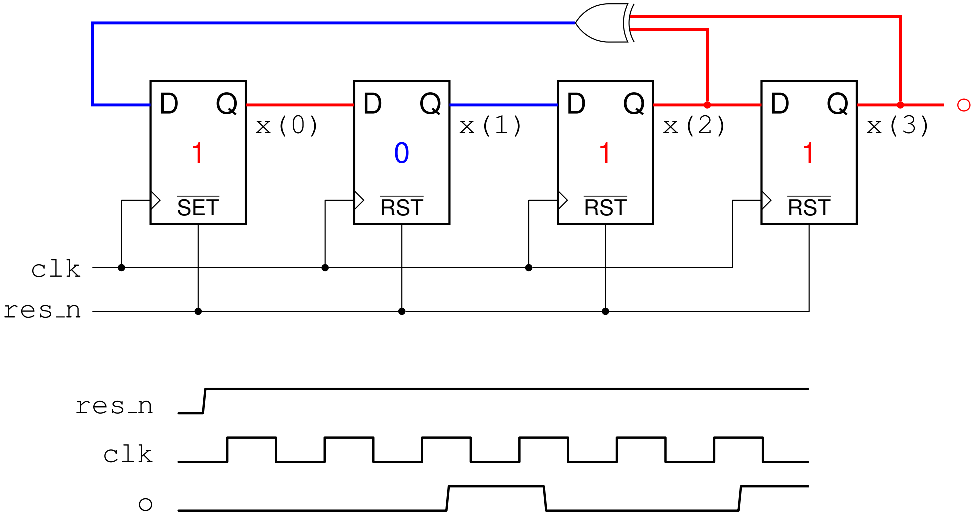
LFSR - Circuit Operation Principle

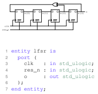




The shifting now leads to the output being set for two consecutive clock cycles. However, by now you have likely gotten the hang of it, and we can continue by discussing how such a circuit can be described in VHDL.

LFSR - Circuit Operation Principle





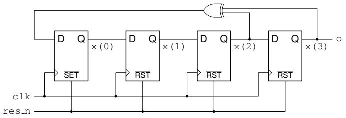
```
entity lfsr is
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

The slide shows an entity declaration for the ”LFSR” circuit. Naturally, it contains a clock, reset and output port. Let us now look at the accompanying architecture.

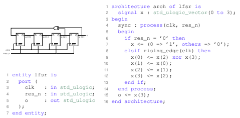
LFSR - VHDL Design

HWMod
WS24

Seq. Elem.
Introduction
D Latches
D Flip-Flop
LFSR
Operation
VHDL Design
Testbench

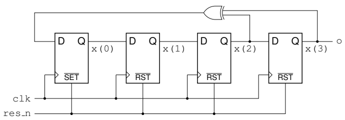


```
entity lfsr is
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    o     : out std_ulogic
  );
end entity;
```



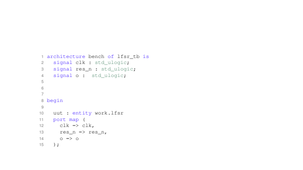
The architecture declares a single vector signal for the register "x". To model this register, we create a process that asynchronously resets this signal and updates it at rising clock edges. During the reset, all bits of "x", except for the one at index zero, are reset to zero via an aggregate expression. The first bit is set to one to model the asynchronous set of the left-most flip-flop. At rising clock edges, the register is updated such that it implements the introduced "LFSR". In particular, this means that we connect the distinct bits of the register to form a chain of flip-flops, where each flip-flop samples the output of its predecessor at a rising clock edge. The left-most flip-flop samples the XOR of the bits two and three. 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

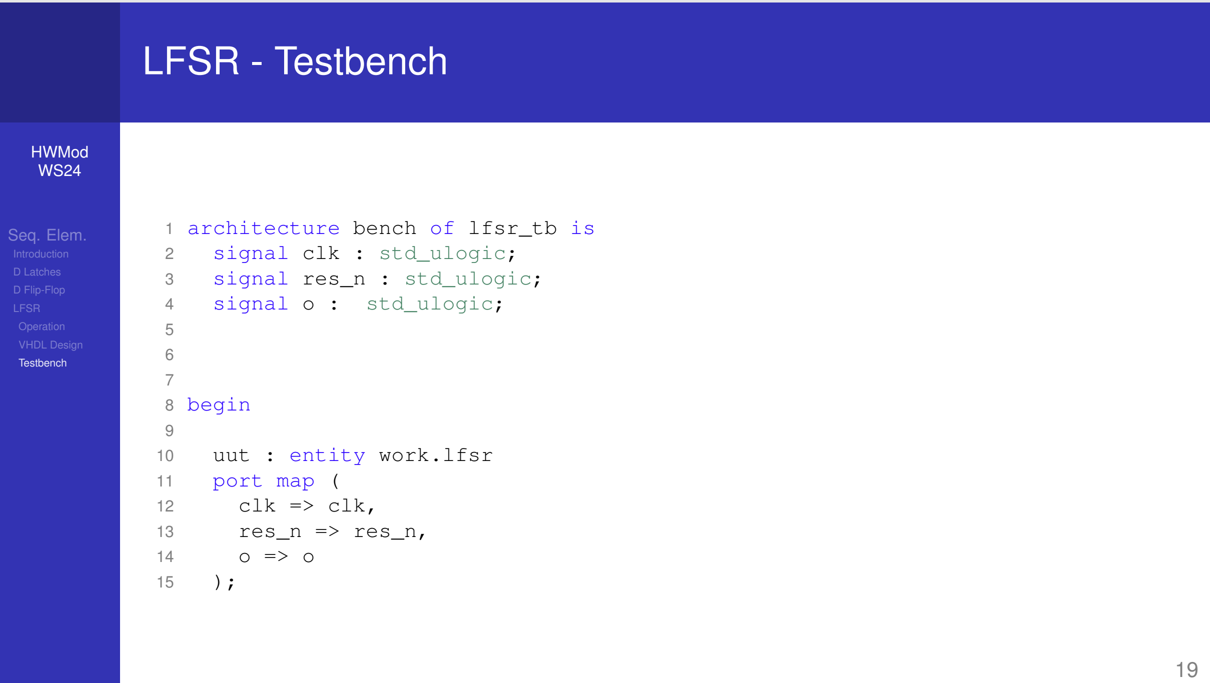


```vhdl
entity lfsr is
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    o     : out std_ulogic
  );
end entity;
```

```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;
```

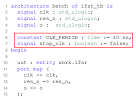


To conclude this example, let us discuss how a testbench can be written for the ”LFSR”. In particular, a synchronous design like this one obviously requires a clock during simulation. But where does this clock signal come from? Well, as for all inputs of the unit-under-test, the testbench has to generate and apply it. We will now show you how this can be done.

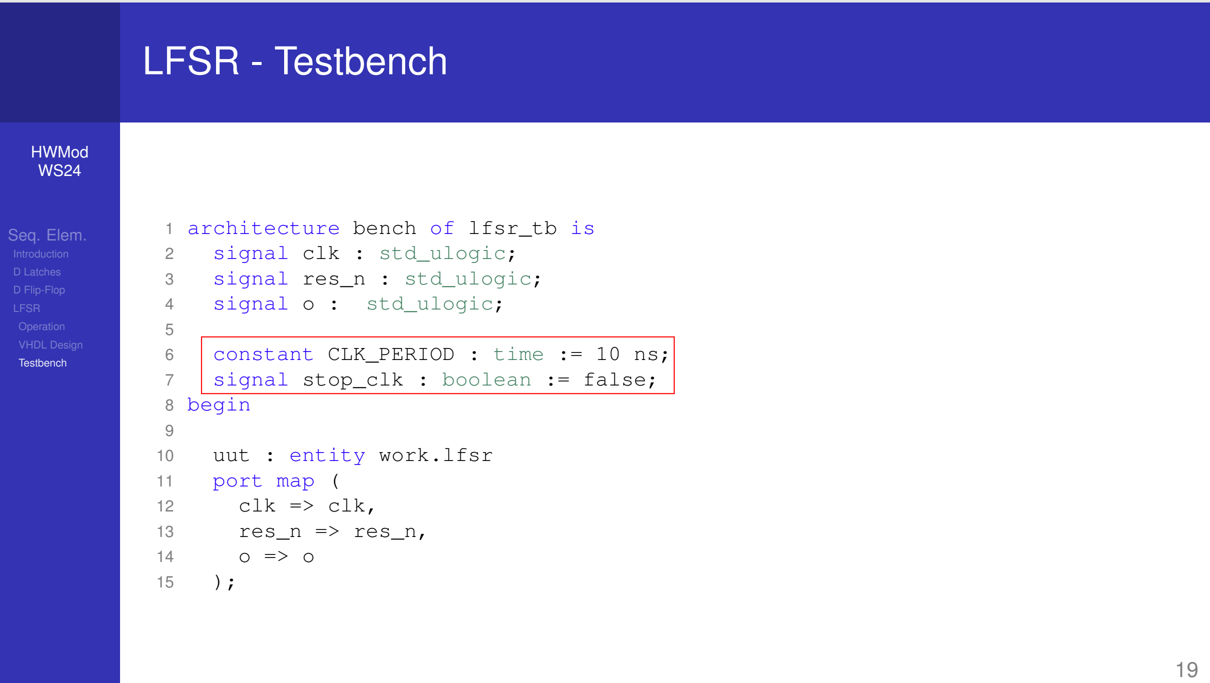


LFSR - Testbench

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



begin

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

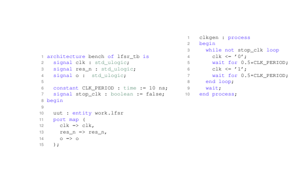


First, we declare an additional constant and signal. The constant holds the desired clock period, in this case we choose ten nanoseconds. However, in general, a design is specified to work within a range or with a particular clock frequency which should then also be generated by the testbench. The purpose of the stop_clk signal will become clear in a moment.

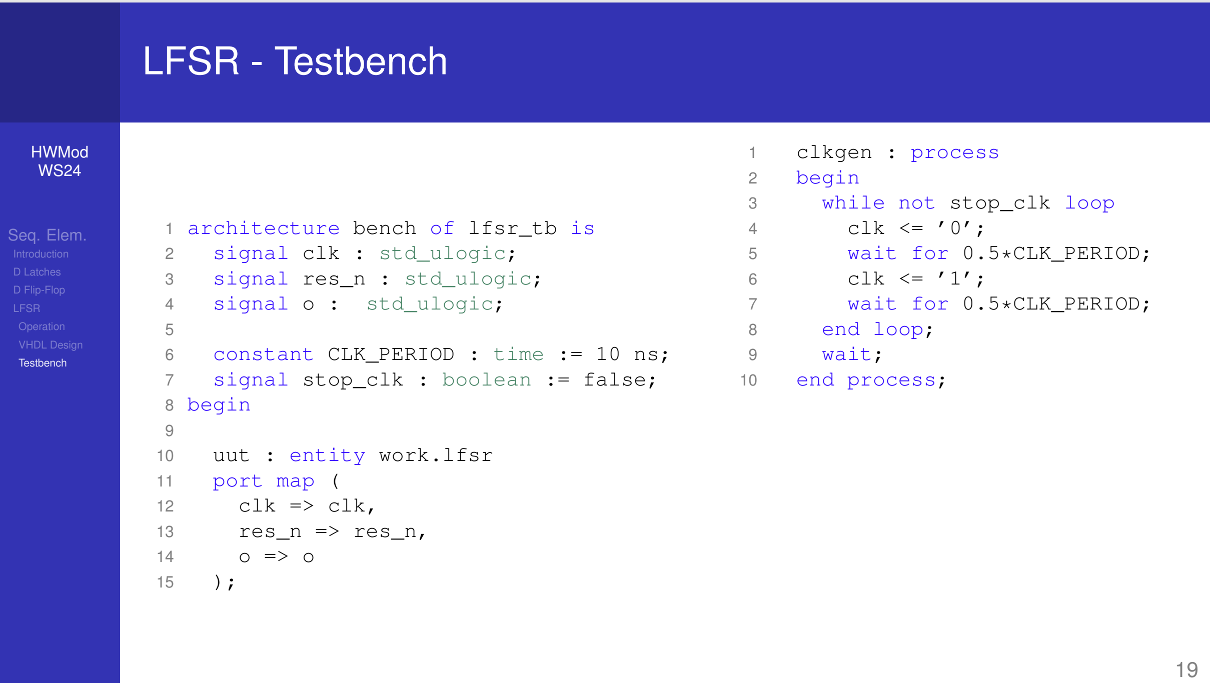


LFSR - Testbench

```
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
  );
```



However, first we create a dedicated clock generation process. This non-synthesizable code periodically toggles the clock signal. The desired clock period is achieved by waiting for half the respective constant after each assignment. If we want the simulation to terminate automatically, we must ensure that all signals become stable eventually. 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. After that, the clock remains stable, and the clock generation process will wait indefinitely. This stop signal will be set by the stimulus process once it is done testing the unit-under-test. We will now look at this process.



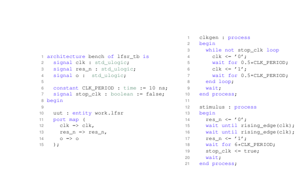
LFSR - Testbench

HWMod
WS24

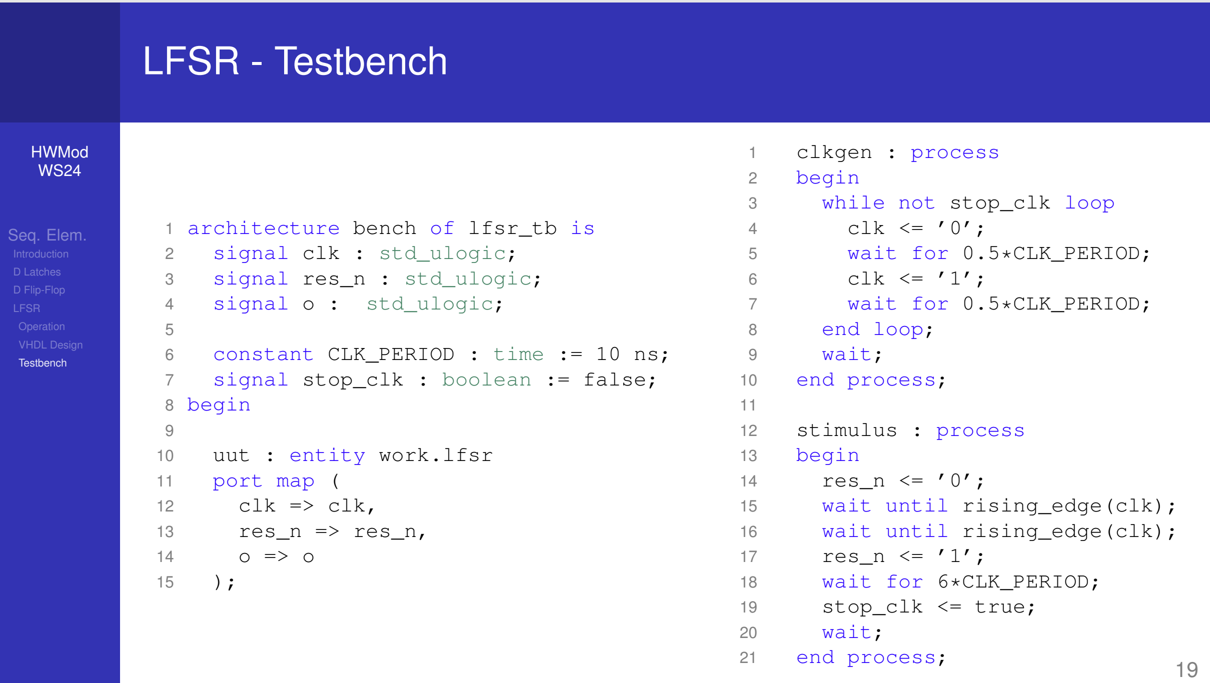
Seq. Elem.
Introduction
D Latches
D Flip-Flop
LFSR
Operation
VHDL Design
Testbench

```
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;
```



The first task of the stimulus process is to reset the ”UUT”, which is also done by the shown code. 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 set the active-low reset to low for two clock cycles. 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.



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;
```

Lecture Complete!

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!