Hardware Modeling [VU] (191.011)
– WS24 –
Latches and combinational loops

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

In this lecture we will discuss the two most common mistakes made when describing sequential circuit elements in VHDL. In particular, the structures that lead to undesired latches and combinational loops will be covered.

Hardware Modeling [VU] (191.011)
– WS24 –
Latches and combinational loops

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-12, 16:31 (21636bb)

In this course latches are **always** undesired

Now that you know the difference between latches and flip-flops, the question arises when you should use which. However, let us make one thing clear straight away: In the context of this course, and also the subsequent lab course, latches are **always** a mistake. Under no circumstances you should create a circuit that contains latches. In the following we will explain why we do not use latches and what you can do to mitigate coding them in the first place.

Undesired Latches

- In this course latches are **always** undesired

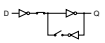
- In this course latches are **always** undesired
- Often not available as primitive building blocks

As we mentioned in a previous lecture, the synchronous design style is used in an overwhelming majority of cases. In this design style the flip-flop is usually the state-holding building block of choice, as its sampling point directly relates to the clock signal central to such designs. Due to the synchronous design style being so prominently used, many technologies target it and do not provide latches.

Undesired Latches

- In this course latches are **always** undesired
- Often not available as primitive building blocks

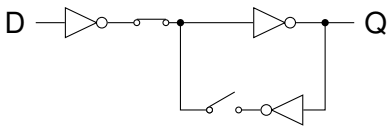
In this course latches are **always** undesired
Often not available as primitive building blocks
Instead built from available combinational elements



If the description of a circuit that targets such a technology still models a latch, it will be built from available combinational logic. In its simplest form such an implementation can look like the one depicted on the slide. It consists of three inverters and two switches, where always only one switch is closed.

Undesired Latches

- In this course latches are **always** undesired
- Often not available as primitive building blocks
  - Instead built from available combinational elements

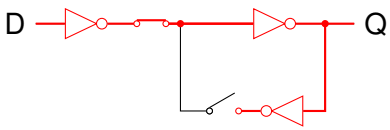


In this course latches are **always** undesired
Often not available as primitive building blocks
Instead built from available combinational elements
D Q

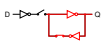
If the first switch is closed, the latch is enabled and forwards the input to its output. This is to what we referred to as the latch being transparent in the lecture about sequential elements. Furthermore, it also applies the input value to the third inverter gate in a feedback path.

Undesired Latches

- In this course latches are **always** undesired
- Often not available as primitive building blocks
  - Instead built from available combinational elements



In this course latches are **always** undesired
Often not available as primitive building blocks
Instead built from available combinational elements

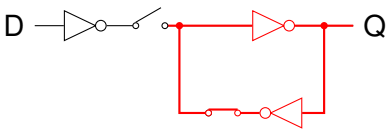


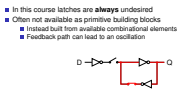
If the input switch is open and the feedback loop closed, the latch outputs the previously captured input value. This corresponds to its enable signal being inactive.

Note that the particular implementation of the switches depends on the respective technology. We will not go into detail on this topic in this course.

Undesired Latches

- In this course latches are **always** undesired
- Often not available as primitive building blocks
  - Instead built from available combinational elements

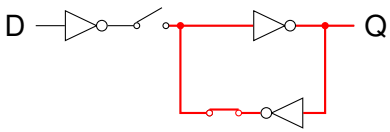




A problem with building a latch like that is that the overall delay of the feedback path, still highlighted in red, can become quite high. If the input was not stable after a transition for sufficiently long, the feedback loop might start to oscillate. This must of course never happen as subsequent parts of the circuit will either also start to oscillate or observe undesired values and transitions.

Undesired Latches

- In this course latches are **always** undesired
- Often not available as primitive building blocks
  - Instead built from available combinational elements
  - Feedback path can lead to an oscillation



How do latches *happen*?

As we discussed on the previous slide, latches can be quite problematic, and you should therefore not use them in your designs. However, sometimes they happen by mistake - especially if you are new to describing hardware. We will now look at a stereotypical example of an undesired latch.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?

The example we consider is a simple combinational comparator. It takes two unsigned inputs, "A" and "B", and computes an output "X" of type `std_ulogic`. Whenever "A" is smaller than "B" the module shall output one, otherwise we do not care. The main process of the shown code implements this. You might want to pause the video for a few seconds to analyze the code and trying to find possible problems. The problem with the shown code is that "X" is not always assigned a value when the process is triggered. Considering that this describes a combinational circuit this is certainly problematic, as such a circuit should map any possible combination of inputs to an output. So, what hardware does the shown code describe?

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?

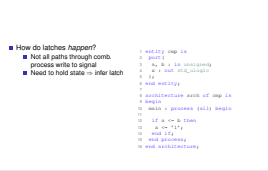
```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```

If we recall the semantics of signal assignments the resulting circuit is actually quite intuitive. The signal "X" will hold its value between assignments. Thus, if there is a path through the process that does not write to "X", we describe a state-holding behavior.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal

```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```

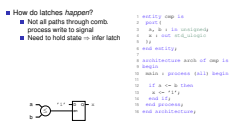


In the absence of a clock this state-holding will be implemented by a latch. We also say, that a latch is inferred from the description.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal
  - Need to hold state ⇒ infer latch

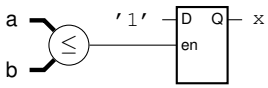
```
1 entity cmp is
2  port(
3   a, b : in unsigned;
4   x : out std_ulogic
5  );
6 end entity;
7
8 architecture arch of cmp is
9 begin
10  main : process (all) begin
11
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```



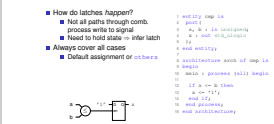
To illustrate this the image on the slide shows the circuit that the code actually describes. How can we mitigate this undesired latch? The answer is quite simple: If a combinational process writes to a signal, we must ensure that it does so on all possible paths through its body.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal
  - Need to hold state ⇒ infer latch



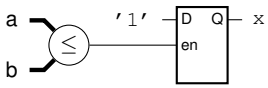

```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```



This is often not the case when signal assignments are part of conditional code and not all branches lead to an assignment. To ensure that no latch is inferred, we can add a default assignment at the beginning of the process, or make sure all cases are covered. The "others" keyword comes in handy for that purpose.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal
  - Need to hold state ⇒ infer latch
- Always cover all cases
  - Default assignment or `others`

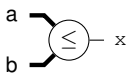



```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```

For our example, we can simply set "X" to 0 before the conditional code. If the condition is met, the previous assignment will be overwritten.

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal
  - Need to hold state ⇒ infer latch
- Always cover all cases
  - Default assignment or `others`

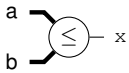



```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11   x <= '0';
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```

Finally, we want to mention that the synthesis tool we are using in this course is capable of detecting latches and will warn you whenever they find one. You must never ignore warnings like that!

Pitfalls - Latch Errors (cont'd)

- How do latches *happen*?
  - Not all paths through comb. process write to signal
  - Need to hold state ⇒ infer latch
- Always cover all cases
  - Default assignment or `others`
- Latches are detected and reported during synthesis
  - **Never** ignore these warnings!




```
1  entity cmp is
2   port(
3    a, b : in unsigned;
4    x : out std_ulogic
5   );
6  end entity;
7
8  architecture arch of cmp is
9  begin
10  main : process (all) begin
11   x <= '0';
12   if a <= b then
13    x <= '1';
14   end if;
15  end process;
16 end architecture;
```

Not only latches can result in comb. feedback loops

On the previous slides we have discussed the problems involved in creating undesired latches. A major problem is the possibility of the feedback loop involved in latches starting to oscillate. However, this is not the only scenario in which such a loop can be implied by the description of a circuit.

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops

Consider the code shown on the slide. It shows the architecture of a purely combinational eight bit counter with wrap-around on overflow. The shown process will trigger whenever the counter variable is incremented, leading to another increment. Briefly pause the video and think about this code. How does the circuit it implements look like? Is this circuit problematic?

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops

```
1 architecture beh of counter is
2  signal cnt : unsigned(7 downto 0);
3 begin
4  comb : process(all) begin
5   cnt <= cnt + 1;
6  end process;
7 end architecture;
```

Not only latches can result in comb. feedback loops

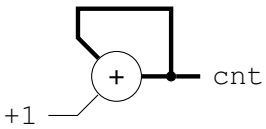
```
architecture beh of counter is
 signal cnt : unsigned(7 downto 0);
begin
 comb : process(all) begin
  cnt <= cnt + 1;
 end process;
end architecture;
```

Next to the code you can see an abstract view of the corresponding circuit. The output of the combinational counter circuit is directly fed back to one of the operand inputs.

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops

```
1  architecture beh of counter is
2   signal cnt : unsigned(7 downto 0);
3  begin
4   comb : process(all) begin
5    cnt <= cnt + 1;
6   end process;
7  end architecture;
```



Not only latches can result in comb. feedback loops

```
architecture beh of counter is
 signal cnt : unsigned(7 downto 0);
begin
 comb : process(all) begin
  cnt <= cnt + 1;
 end process;
end architecture;
```

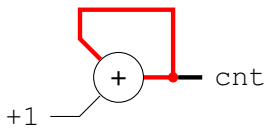
+1 cnt

This is again a combinational feedback loop and prone to the same problems as the one in the latch we discussed previously. The feedback loop is highlighted in red.

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops

```
1  architecture beh of counter is
2   signal cnt : unsigned(7 downto 0);
3  begin
4   comb : process(all) begin
5    cnt <= cnt + 1;
6   end process;
7  end architecture;
```



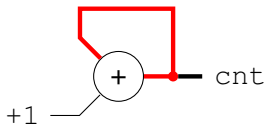
- Not only latches can result in comb. feedback loops
- A comb. process must never read **and** write to same signal

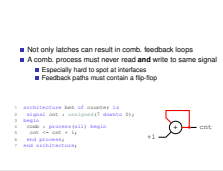
In order to prevent your designs from describing such loops, combinational processes must never read and write to the same signal. Whenever this is the case, the assignment will trigger the process which might result in another assignment, thus creating a loop.

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops
- A comb. process must never read **and** write to same signal

```
1  architecture beh of counter is
2   signal cnt : unsigned(7 downto 0);
3  begin
4   comb : process(all) begin
5    cnt <= cnt + 1;
6   end process;
7  end architecture;
```



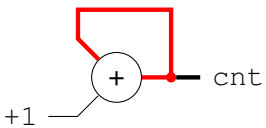


Special care must be taken at the interface between modules as it can be quite hard for a designer to spot such loops between multiple modules. However, since feedback loops can be required, for example in control applications, it is often necessary to create designs containing them. When doing so, the clue is to not create purely combinational loops, but rather to include flip-flops in the feedback path. This way, signal transitions can only propagate on a per-clock-period basis, allowing the tool to ensure that no oscillations will happen.

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops
- A comb. process must never read **and** write to same signal
  - Especially hard to spot at interfaces
  - Feedback paths must contain a flip-flop

```
1  architecture beh of counter is
2   signal cnt : unsigned(7 downto 0);
3  begin
4   comb : process(all) begin
5    cnt <= cnt + 1;
6   end process;
7  end architecture;
```



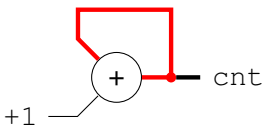
- Not only latches can result in comb. feedback loops
- A comb. process must never read **and** write to same signal
  - Especially hard to spot at interfaces
  - Feedback paths must contain a flip-flop
- Combinational loops are also reported during synthesis

Speaking of tools, just as with latches tools will also detect combinational loops and warn you about them. Again: do not ignore such warnings!

Pitfalls - Combinational Loops

- Not only latches can result in comb. feedback loops
- A comb. process must never read **and** write to same signal
  - Especially hard to spot at interfaces
  - Feedback paths must contain a flip-flop
- Combinational loops are also reported during synthesis

```
1  architecture beh of counter is
2   signal cnt : unsigned(7 downto 0);
3  begin
4   comb : process(all) begin
5    cnt <= cnt + 1;
6   end process;
7  end architecture;
```



Let us finish our discussion of the pitfalls involved in coding sequential logic by considering how we could implement the counter from the previous slide without a purely combinational feedback loop. First, let us introduce an additional signal with the suffix "next". We will use this signal to name the output of the combinational adder required for our counter. The other signal, `cnt`, will be used to describe a flip-flop.

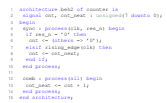
Pitfalls - Combinational Loops (cnt'd)


```
1  architecture beh2 of counter is
2   signal cnt, cnt_next : unsigned(7 downto 0);
3  begin
```

The process describing the register adheres to the structure you already saw in a previous lecture. The modelled register features an asynchronous reset and samples the value of cnt_next on any rising clock edge. This value stored by the register is then processed by combinational logic.

Pitfalls - Combinational Loops (cnt'd)

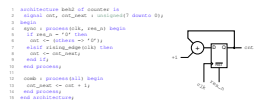

```
1  architecture beh2 of counter is
2   signal cnt, cnt_next : unsigned(7 downto 0);
3  begin
4   sync : process(clk, res_n) begin
5    if res_n = '0' then
6     cnt <= (others => '0');
7    elsif rising_edge(clk) then
8     cnt <= cnt_next;
9    end if;
10  end process;
11
```



The respective process is almost the same as the one before, that resulted in a combinational loop, except for a small detail. Instead of having the same signal in the left and right-hand side of the increment assignment, the current output of the register is incremented by one and assigned to the signal that the flip-flop will sample at the next rising clock edge. Observe how the combinational process thus now no longer reads from and writes to the same signal! This is immediately clear when we look at the circuit described by this code.

Pitfalls - Combinational Loops (cnt'd)

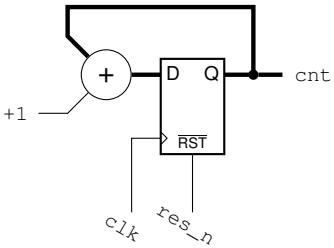

```
1  architecture beh2 of counter is
2   signal cnt, cnt_next : unsigned(7 downto 0);
3  begin
4   sync : process(clk, res_n) begin
5    if res_n = '0' then
6     cnt <= (others => '0');
7    elsif rising_edge(clk) then
8     cnt <= cnt_next;
9    end if;
10  end process;
11
12  comb : process(all) begin
13   cnt_next <= cnt + 1;
14  end process;
15 end architecture;
```



As expected, the resulting circuit, shown on the slide, consists of an adder and a flip-flop. Due to the flip-flop intercepting the path from the adder's output to its input, there is no combinational loop any longer. Note that this structure comprising a synchronous process modeling flip-flops that sample "next" signals produced by one or more combinational processes is an important design pattern used for many VHDL designs. Keep this in mind when writing your own code.

Pitfalls - Combinational Loops (cnt'd)


```
1  architecture beh2 of counter is
2   signal cnt, cnt_next : unsigned(7 downto 0);
3  begin
4   sync : process(clk, res_n) begin
5    if res_n = '0' then
6     cnt <= (others => '0');
7    elsif rising_edge(clk) then
8     cnt <= cnt_next;
9    end if;
10  end process;
11
12  comb : process(all) begin
13   cnt_next <= cnt + 1;
14  end process;
15 end architecture;
```



Latches
Often due to missing default values in comb. processes

Finally, let us finish this lecture by highlighting the key takeaways. Undesired latches often have their origin in missing default assignments in combinational processes containing conditional code, which then often results in signals being required to hold their value. To mitigate this, use default assignments or ensure that you cover all possible outcomes of conditions. Of course, doing both is also fine.

Summary

- Latches
  - Often due to missing default values in comb. processes

- Latches
  - Often due to missing default values in comb. processes
- Comb. loops
  - Never read from and write to the same signal in a comb. process

Combinational loops are the result of combinational logic reading from and writing to the same signal, leading to feedback paths that do not contain flip-flops. To mitigate this, make sure that your combinational processes never read from and write to the same signal. If such feedback behavior is required, introduce flip-flops somewhere in the feedback path. Of course, such loops can also be the result of multiple combinational processes.

Summary

- Latches
  - Often due to missing default values in comb. processes
- Comb. loops
  - Never read from and write to the same signal in a comb. process

Finally, never ignore the respective warnings produced by the tools. In this course, this always highlights fatal issues in your design that you need to deal with. The slide shows you examples of how such warnings can look like.

Summary

- Latches
  - Often due to missing default values in comb. processes
- Comb. loops
  - Never read from and write to the same signal in a comb. process
- Never ignore the tool warnings!

```
10631 VHDL Process Statement warning at top_arch.vhd(13): inferring latch(es) for signal or variable "abc", which holds its previous value in one or more paths through the process

332125 Found combinational loop of 2 nodes
```

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!