Hardware Modeling [VU] (191.011)
– WS24 –
Synchronizers and Debouncers

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

In the previous lecture you heard that external inputs can lead to the highly problematic phenomenon of metastability. In this lecture we will show you how the effects of metastability can be mitigated by using synchronizers. Furthermore, we will discuss debouncers which can be used to sanitize inputs from bouncing contacts.

Hardware Modeling [VU] (191.011)
– WS24 –
Synchronizers and Debouncers

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-12, 16:34 (21636bb)

We can get a statistical estimate of the MTBU
$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$

Before we really dive into this lecture's topics, let us recall some important things about the guest lecture on metastability. First, recall that while we cannot avoid metastability or determine when exactly it will occur, we can at least get a statistical estimate of its effects in form of the meant time between upsets, or MTBUusing the formula shown on the slide.

Recall: MTBU Estimation

- We can get a statistical estimate of the MTBU

$$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$$

- We can get a statistical estimate of the MTBU
$$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$$
- Exponential dependence of MTBU on time to resolve t_{res}

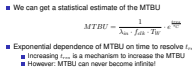
An important observation that we can make is that t_{res}, which is the time a flip-flop has to resolve its metastability, has an exponential influence on the MTBU.

Recall: MTBU Estimation

- We can get a statistical estimate of the MTBU

$$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$$

- Exponential dependence of MTBU on time to resolve t_{res}



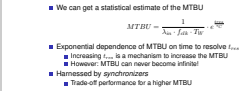
Therefore, increasing this time is an efficient and powerful mechanism to increase the mean time between upsets. However, while the formula shows us that we can make the MTBUarbitrarily large, it also suggests that it can never become infinite. This aligns with the fact that metastability can in general not be avoided. We can merely make it improbable to affect us.

Recall: MTBU Estimation

- We can get a statistical estimate of the MTBU

$$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$$

- Exponential dependence of MTBU on time to resolve t_{res}
  - Increasing t_{res} is a mechanism to increase the MTBU
  - However: MTBU can never become infinite!



In synchronous designs this is usually achieved using special circuits called *synchronizers*. These circuits essentially trade performance against a higher MTBU.

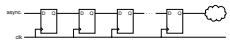
Recall: MTBU Estimation

- We can get a statistical estimate of the MTBU

$$MTBU = \frac{1}{\lambda_{in} \cdot f_{clk} \cdot T_W} \cdot e^{\frac{t_{res}}{\tau_C}}$$

- Exponential dependence of MTBU on time to resolve t_{res}
  - Increasing t_{res} is a mechanism to increase the MTBU
  - However: MTBU can never become infinite!
- Harnessed by *synchronizers*
  - Trade-off performance for a higher MTBU

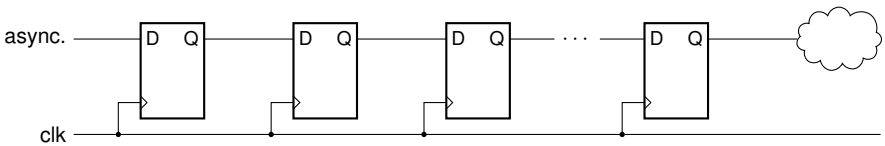
Chain of flip-flops

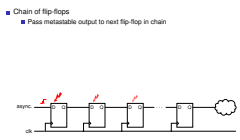


The most popular synchronizer is the one shown in the figure. It is essentially just a chain of flip-flops with no combinational logic in between the output of one flip-flop and input of the succeeding one. Each flip-flop is merely handing over its state to the next one in the chain.

Waiting Synchronizers

- Chain of flip-flops

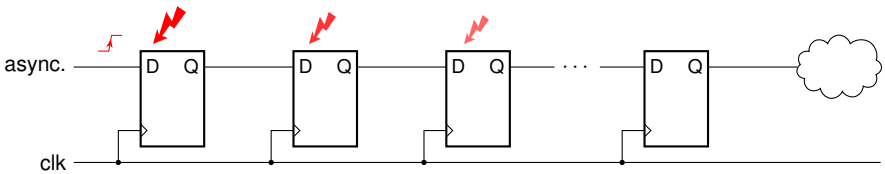


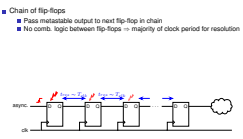


The basic idea is, that if the first flip-flop becomes metastable due to an asynchronous input transition falling within its setup-hold window, it forwards its metastable output to the next flip-flop. This handling over decreases the probability of the metastable output being forwarded for each flip-flop in the chain. But why should this actually work?

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain

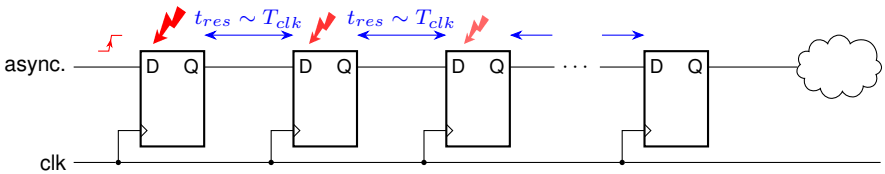


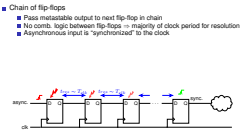


Well, if we recall the basic functionality of flip-flops, we can note that handing over a value to a successor only happens once per clock cycle. And since there is no combinational logic between the flip-flops, almost the whole clock period is available as time to resolve. Thus, if the first flip-flop in the chain becomes metastable, it already has some time to resolve before its output is captured by its neighbor. Therefore, this metastable value will at least partly be resolved. In a nutshell, for the succeeding flip-flop the process of resolving the metastable value will be easier because of the prior work its predecessor put into this exact task.

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution

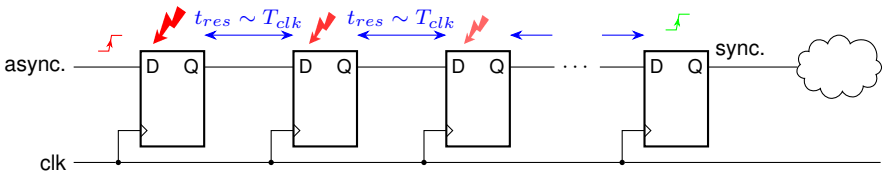




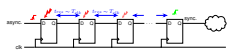
Finally, if an appropriate synchronizer is used, the input will have been synchronized to the circuit's clock with an overwhelmingly high probability. But what do we mean by "appropriate"?

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock



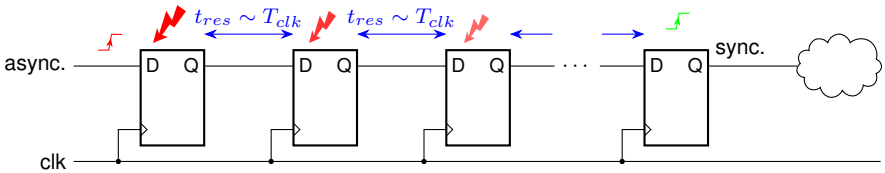
- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones



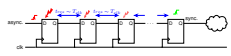
Basically, we want the MTBUto become astronomically high such that it is completely improbable that a flip-flop in our actual circuit is ever upset. Following our previous observations about the MTBU, in order to achieve such high values we need a long time to resolve. The nice thing about this circuit now is, that it turns out that we can sum up all individual resolution times for the overall MTBU.

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones



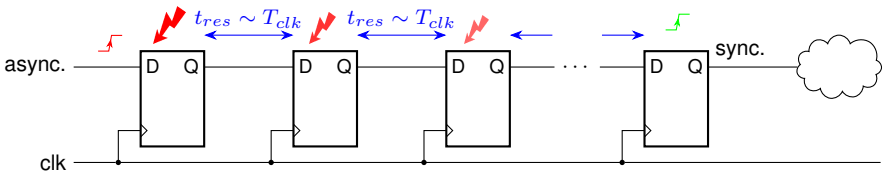
- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones
  - ⇒ Exponential increase in MTBU **per** flip-flop



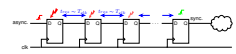
This essentially means that we can achieve an exponential increase in the MTBU**per** flip-flop in the chain.

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones
  - ⇒ Exponential increase in MTBU **per** flip-flop



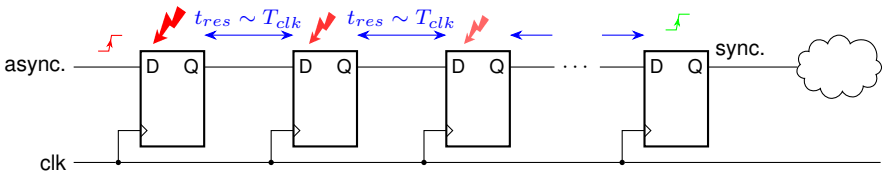
- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones
  - ⇒ Exponential increase in MTBU **per** flip-flop
- In practice often two flip-flops, three to be on the safe side
  - Trade-off between latency and MTBU



Due to this relation, in practice often two flip-flops are sufficient to obtain MTBUvalues of more than a hundred years, depending on the application and flip-flop parameters. Synchronizers comprising three flip-flops can be considered to be very safe in most cases. However, the reason why do not simply always take three or even more flip-flops is that each element of the chain also means that input values need longer to pass through the synchronizer, thus increasing latency.

Waiting Synchronizers

- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops ⇒ majority of clock period for resolution
  - Asynchronous input is "synchronized" to the clock
- Overall resolution time is the sum of the individual ones
  - ⇒ Exponential increase in MTBU **per** flip-flop
- In practice often two flip-flops, three to be on the safe side
  - Trade-off between latency and MTBU



Let us now discuss how we can implement such a synchronizer circuit in VHDL, which at this point of the lecture should be a fairly trivial task. The slide already shows you a suitable entity declaration.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

The first thing we recognize are two generics. One defines the stages, which is the amount of flip-flops minus 1. The other one is for defining a reset value of the synchronizer chain's flip-flops. This is necessary because signals can either be low or high active.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

Next, we of course have the required ports. For a chain of flip-flops this naturally includes a clock and a reset signal. Furthermore, we require an input for the asynchronous signal and an output for the synchronized one.

VHDL Implementation


```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity synchronizer is
5    generic (
6      STAGES  : natural;
7      RES_VAL : std_ulogic
8    );
9    port (
10     clk   : in std_ulogic;
11     res_n : in std_ulogic;
12     async : in std_ulogic;
13     sync  : out std_ulogic
14   );
15 end entity;
```

Let us now get to the architecture.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of synchronizer is
  signal ffs: std_ulogic_vector(0 to STAGES);
begin
  process (clk, res_n) begin
    if res_n = '0' then
      ffs <= (others => RES_VAL);
    elsif rising_edge(clk) then
      ffs(0) <= async;
      for i in 1 to STAGES-1 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```

We first declare a vector signal for the flip-flops. Each of the vectors elements will correspond to the state and thus output of one flip-flop.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of synchronizer is
  signal ffs: std_ulogic_vector(0 to STAGES);
begin
  process (clk, res_n) begin
    if res_n = '0' then
      ffs <= (others => RES_VAL);
    elsif rising_edge(clk) then
      ffs(0) <= async;
      for i in 1 to STAGES-1 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```

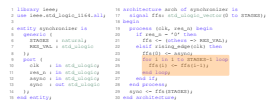
Next, inside a typical process for describing flip-flops, we first assign the asynchronous input to the first vector element. This models the first flip-flop sampling this input at each active clock edge.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of synchronizer is
  signal ffs: std_ulogic_vector(0 to STAGES);
begin
  process (clk, res_n) begin
    if res_n = '0' then
      ffs <= (others => RES_VAL);
    elsif rising_edge(clk) then
      ffs(0) <= async;
      for i in 1 to STAGES-1 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```



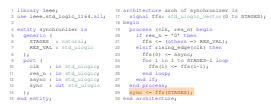
Each of the other flip-flops in the chain will sample the output of its preceding flip-flop, which we model via a loop.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of synchronizer is
  signal ffs: std_ulogic_vector(0 to STAGES);
begin
  process (clk, res_n) begin
    if res_n = '0' then
      ffs <= (others => RES_VAL);
    elsif rising_edge(clk) then
      ffs(0) <= async;
      for i in 1 to STAGES-1 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```



Finally, the output of the last flip-flop is provided as the output of the overall synchronizer.

VHDL Implementation


```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity synchronizer is
  generic (
    STAGES  : natural;
    RES_VAL : std_ulogic
  );
  port (
    clk   : in std_ulogic;
    res_n : in std_ulogic;
    async : in std_ulogic;
    sync  : out std_ulogic
  );
end entity;
```

```vhdl
architecture arch of synchronizer is
  signal ffs: std_ulogic_vector(0 to STAGES);
begin
  process (clk, res_n) begin
    if res_n = '0' then
      ffs <= (others => RES_VAL);
    elsif rising_edge(clk) then
      ffs(0) <= async;
      for i in 1 to STAGES-1 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```

MTBU can be made *arbitrarily* large by appropriate synchronizer

Finally, before we continue with the next topic, we want to point out a few important aspects which you should take away from this lecture. The first one is that the MTBUcan be made arbitrarily large by using an appropriate synchronizer.

Important Aspects

- MTBU can be made *arbitrarily* large by appropriate synchronizer

MTBU can be made *arbitrarily* large by appropriate synchronizer
A synchronizer does **not** prevent metastability!

However, we need to stress that a synchronizer does not prevent metastability. It merely makes it less probable to affect a circuit.

Important Aspects

- MTBU can be made *arbitrarily* large by appropriate synchronizer
  - A synchronizer does **not** prevent metastability!

MTBU can be made *arbitrarily* large by appropriate synchronizer
A synchronizer does **not** prevent metastability!
A single flip-flop alone is not a synchronizer

Next, we want to point out that a single flip-flop alone is no kind of synchronizer. For synchronization, we really need at least a second flip-flop as this is creating the additional time to resolve.

Important Aspects

- MTBU can be made *arbitrarily* large by appropriate synchronizer
  - A synchronizer does **not** prevent metastability!
- A single flip-flop alone is not a synchronizer

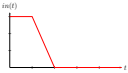
- MTBU can be made *arbitrarily* large by appropriate synchronizer
  - A synchronizer does **not** prevent metastability!
- A single flip-flop alone is not a synchronizer
- The MTBU is a statistical quantity
  - No guarantee for upset-freedom at any time

And finally, the MTBUis a statistical quantity. Thus, even if you design a synchronizer with an MTBUof more than the age of the universe, we might experience two subsequent upsets within a second after starting our design. Of course, this becomes astronomically unlikely, but it still remains possible.

Important Aspects

- MTBU can be made *arbitrarily* large by appropriate synchronizer
  - A synchronizer does **not** prevent metastability!
- A single flip-flop alone is not a synchronizer
- The MTBU is a statistical quantity
  - No guarantee for upset-freedom at any time

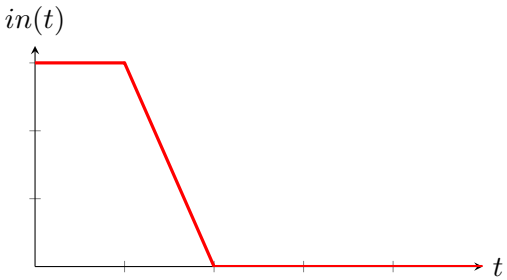
Asynchronous inputs are not the only problem at interfaces

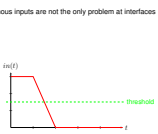


As you might already have heard before, being uncorrelated to an internal clock is only an issue external inputs have. Another issue has rather to do with the shape of a transition on the input rather than its time of arrival. To motivate what we mean, consider the waveform on the slide. Basically, we plotted the voltage at the input against the time. Note how the transition from high to low does not happen immediately but rather takes some time to the speed with which a signal propagates being finite.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces

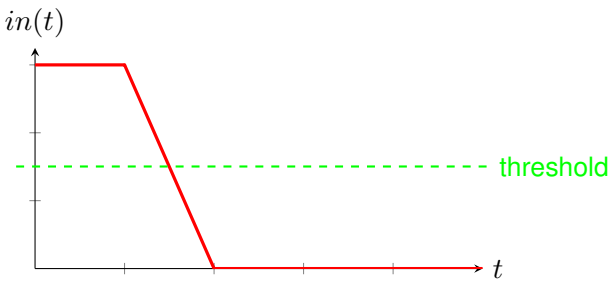


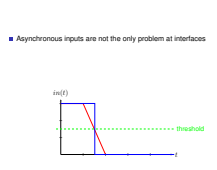


If we recall that digital circuits discretize analog voltages by comparing it against a threshold voltage, we do not really care about this slope in the majority of cases in praxis.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces

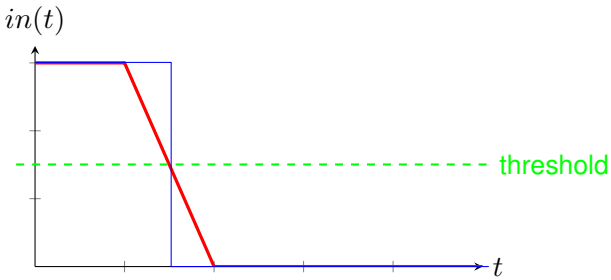


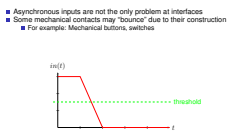


As the blue waveform in the illustration on the slide shows you, the input transition is properly recognized by a receiving circuit and in praxis often even made steeper.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces

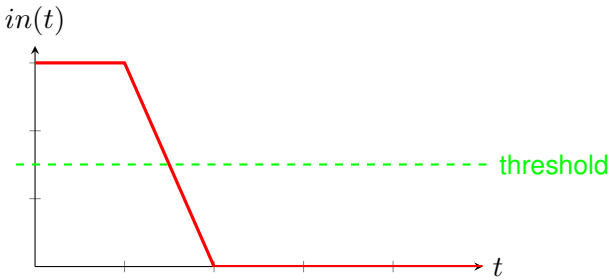


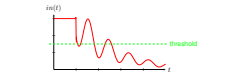


However, input transitions like the red one are not always what we actually get at inputs, as many mechanical devices like buttons and switches come with a further problem. Often these mechanical sensors often feature spring contacts to sustain some pressure on contacts to make it conduct well and reliably. When switching though, these springs tend to vibrate, meaning that, for example, a switch is opened and closed a couple of times before eventually reaching its final position. This happens so fast that we as human observers cannot recognize it in our daily lives. However, for a computer operated at megahertz frequencies these input changes are clearly visible. We refer to this phenomenon as bouncing.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces
- Some mechanical contacts may “bounce” due to their construction
  - For example: Mechanical buttons, switches

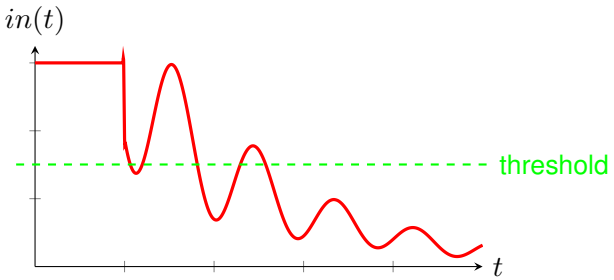


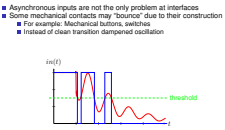


As a result of the input contact bouncing, the input voltage will be a dampened oscillation rather than a simple slop as before. You can find this illustrated on the slide.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces
- Some mechanical contacts may “bounce” due to their construction
  - For example: Mechanical buttons, switches
  - Instead of clean transition dampened oscillation

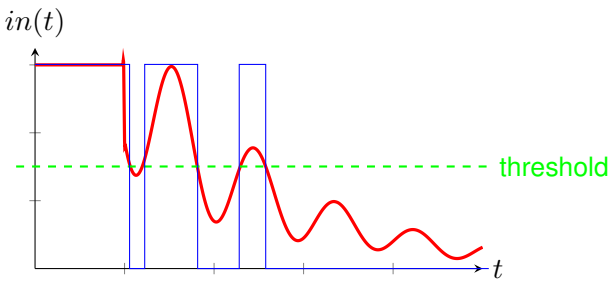


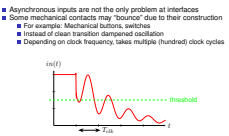


As before, a digital circuit element will essentially discretize this input voltage based on a threshold. However, whereas the slop did result in a single clean transition of the right type, the oscillation might actually lead to multiple transitions, both of the wanted and the unwanted type.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces
- Some mechanical contacts may “bounce” due to their construction
  - For example: Mechanical buttons, switches
  - Instead of clean transition dampened oscillation

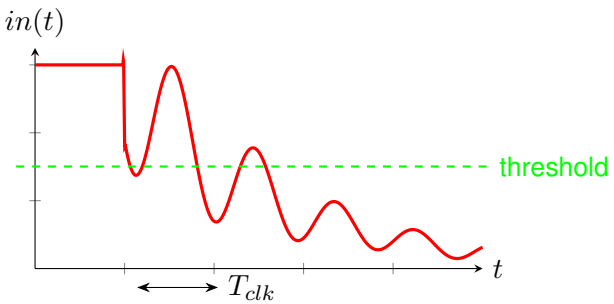


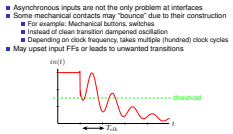


This problem is further aggravated by the oscillation often being of significantly lower period than the clock signal, thus resulting in the input requiring up to hundreds of clock cycles to become stable.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces
- Some mechanical contacts may “bounce” due to their construction
  - For example: Mechanical buttons, switches
  - Instead of clean transition dampened oscillation
  - Depending on clock frequency, takes multiple (hundred) clock cycles

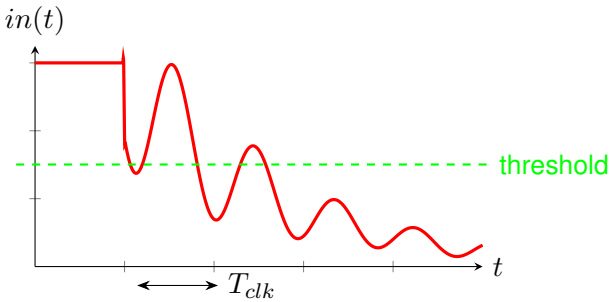




Obviously, this issue is highly problematic as it can result in flip-flops being upset or to unwanted transitions that bring the circuit in erroneous states.

Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces
- Some mechanical contacts may “bounce” due to their construction
  - For example: Mechanical buttons, switches
  - Instead of clean transition dampened oscillation
  - Depending on clock frequency, takes multiple (hundred) clock cycles
- May upset input FFs or leads to unwanted transitions



Simply filter-out sequence of input transitions that is “too fast”

Fortunately, such fast and undesired bursts of transition can easily be recognized as no one really push a button in a millisecond-pace. Therefore, any type of low-pass filtering can be applied to mitigate bouncing.

Counter Measures

- Simply filter-out sequence of input transitions that is “too fast”

Simply filter-out sequence of input transitions that is "too fast"
Analog (low pass) filtering

Sometimes this is done using analog circuitry like an RC filter, which leads to a digital circuit not even observing the bouncing in the first place. However, while this is always possible, it is often inconvenient to include analog circuit components in a circuit.

Counter Measures

- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering

- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering
  - Digital filtering to check if output stabilizes
    - Use timer to wait (FSM)
    - Alternatives exist

Therefore, the standard solution to debouncing is usually filtering out undesired transitions in the digital domain. One particular method, which we will discuss on the next slide, is to simply use a timer to wait for the bouncing input to stabilize. This can be implemented using a very simple finite state machine. However, be aware that alternatives to this approach exist, although they are all based around the same general approach of simply waiting the input becomes sufficiently stable.

Counter Measures

- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering
  - Digital filtering to check if output stabilizes
    - Use timer to wait (FSM)
    - Alternatives exist

- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering
  - Digital filtering to check if output stabilizes
    - Use timer to wait (FSM)
    - Alternatives exist
  - Software-based debouncing

We also want to point out that it is possible to debounce inputs in software in case a system is capable of executing code with direct access to inputs. For example, on microcontrollers bouncing buttons or switches are often handled by maintaining a counter in software and using it to wait after a transition for the input to stabilize.

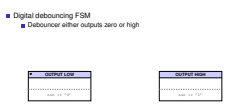
Counter Measures

- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering
  - Digital filtering to check if output stabilizes
    - Use timer to wait (FSM)
    - Alternatives exist
  - Software-based debouncing

Finally, let us discuss one particular digital debouncer, which we will model using a state machine Let us now model such an FSM that waits for input transitions to be stable. We assume that the input is active-high.

Debouncer Implementation

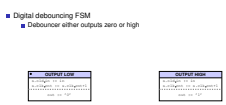
- Digital debouncing FSM



We start by creating two states named `OUTPUT_LOW` and `OUTPUT_HIGH`. The idea is that the FSM will be in the `OUTPUT_LOW` state while the input is low or currently transitioning to high and bouncing and likewise for the `OUTPUT_HIGH`. Since the input is assumed to be active high the initial state is the low one.

Debouncer Implementation

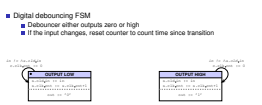
- Digital debouncing FSM
  - Debouncer either outputs zero or high



In these states we sample the current input value into a register such that we can detect transitions. Next, we add a counter to the state register of the FSM which is used to detect if the input was stable for a sufficiently long time. We will use increment this counter each clock cycle per default, using it to keep track of the time since the last detected transition.

Debouncer Implementation

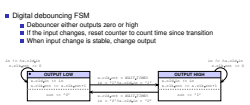
- Digital debouncing FSM
  - Debouncer either outputs zero or high



If an input transition is now detected, the FSM will reset its counter, thus starting to count the amount of clock cycles since this transition.

Debouncer Implementation

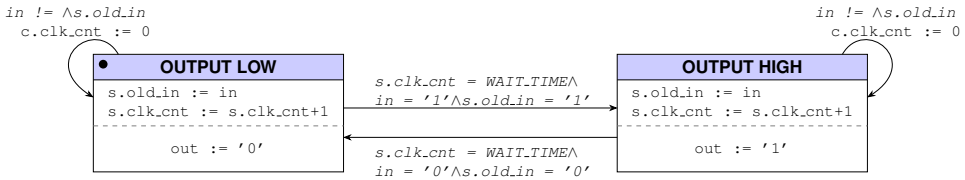
- Digital debouncing FSM
  - Debouncer either outputs zero or high
  - If the input changes, reset counter to count time since transition



Based on this resetting of the counter, and if the constant `WAIT_TIME` is sufficiently high, we know that the input will be stable when the counter reaches this value and the input is not the one corresponding to the current state. Therefore, the debouncing FSM can transition to the other state and change its output. We leave a VHDL implementation of this model as an exercise and finally want to point out again that this is only one possible digital debouncer circuit.

Debouncer Implementation

- Digital debouncing FSM
  - Debouncer either outputs zero or high
  - If the input changes, reset counter to count time since transition
  - When input change is stable, change output



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!