Hardware Modeling [VU] (191.011)
– WS25 –
Synchronizers and Debouncers

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:05 (f8a58e9)

Recall: MTBU Estimation

- We can get a statistical estimate of the MTBU

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

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}

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!

Recall: MTBU Estimation

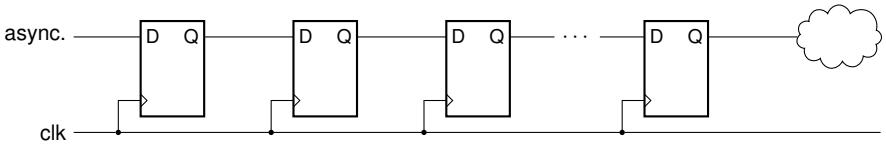
- We can get a statistical estimate of the MTBU

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

- 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

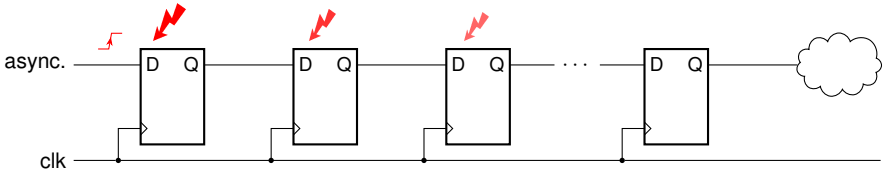
Waiting Synchronizers

- Chain of flip-flops



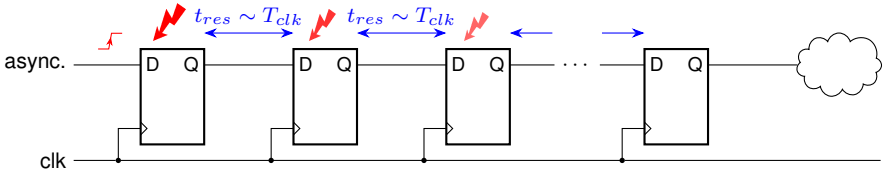
Waiting Synchronizers

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



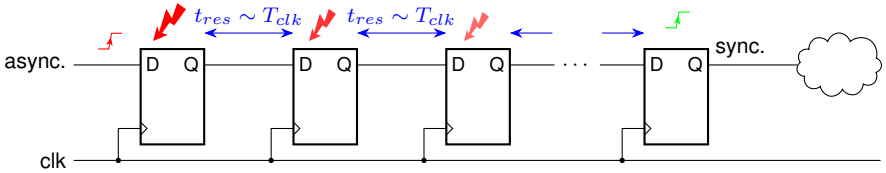
Waiting Synchronizers

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



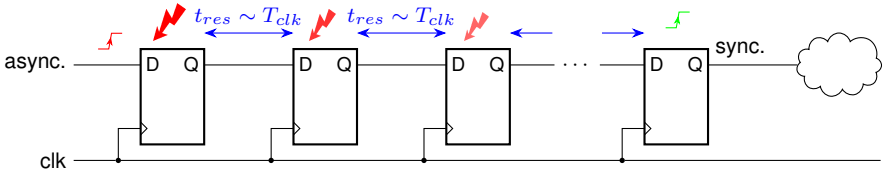
Waiting Synchronizers

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



Waiting Synchronizers

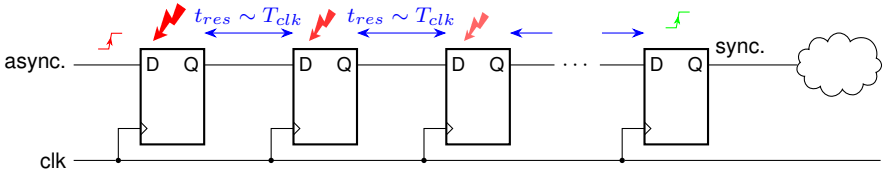
- Chain of flip-flops
  - Pass metastable output to next flip-flop in chain
  - No comb. logic between flip-flops $\Rightarrow$ majority of clock period for resolution
  - Asynchronous input is “synchronized” to the clock
- Overall resolution time is the sum of the individual ones



Waiting Synchronizers

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

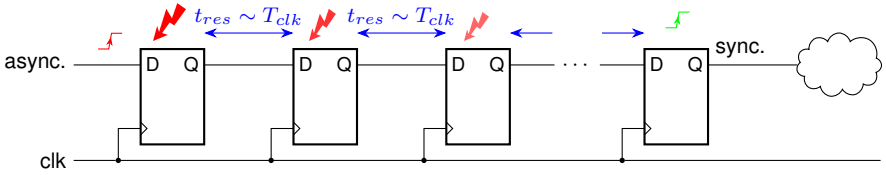
  $\Rightarrow$ Exponential increase in MTBU **per** flip-flop



Waiting Synchronizers

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

  $\Rightarrow$ 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



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;
```

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;
```

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 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 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end 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 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end 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 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end 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 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end 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 loop
        ffs(i) <= ffs(i-1);
      end loop;
    end if;
  end process;
  sync <= ffs(STAGES);
end architecture;
```

Important Aspects

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

Important Aspects

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

Important Aspects

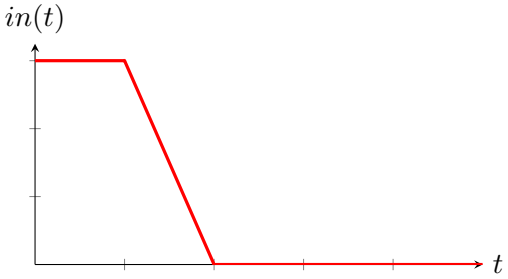
- MTBU can be made *arbitrarily* large by appropriate synchronizer
  - A synchronizer does **not** prevent metastability!
- A single flip-flop alone is not a synchronizer

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

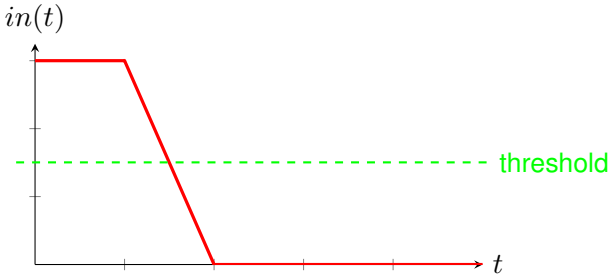
Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces



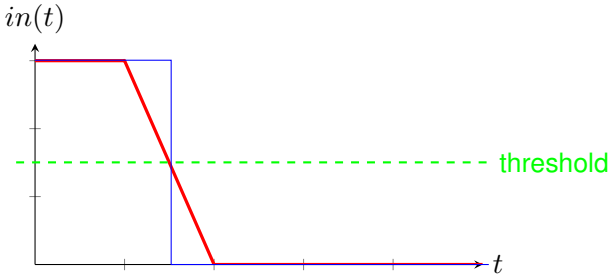
Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces



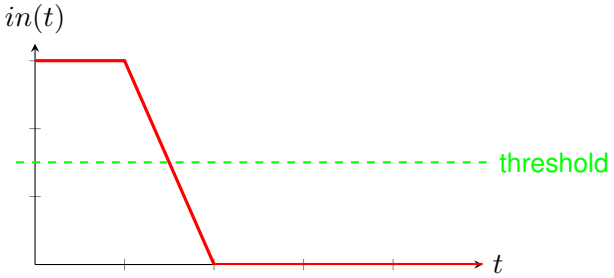
Bouncing Inputs

- Asynchronous inputs are not the only problem at interfaces



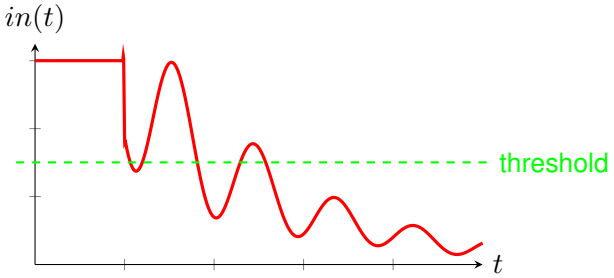
Bouncing Inputs

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



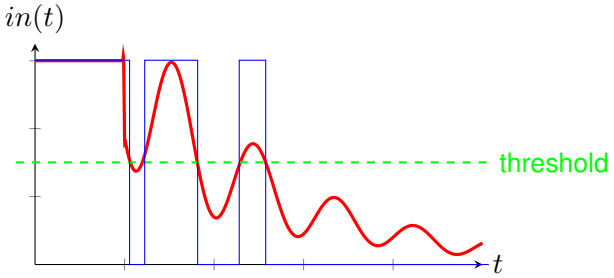
Bouncing Inputs

- 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



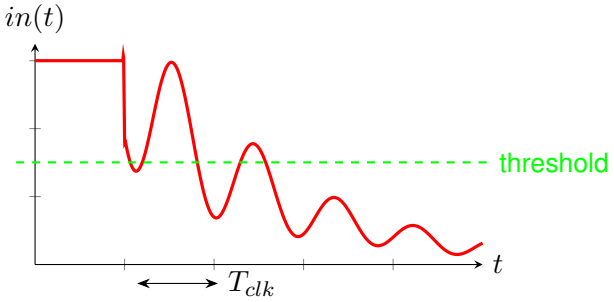
Bouncing Inputs

- 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



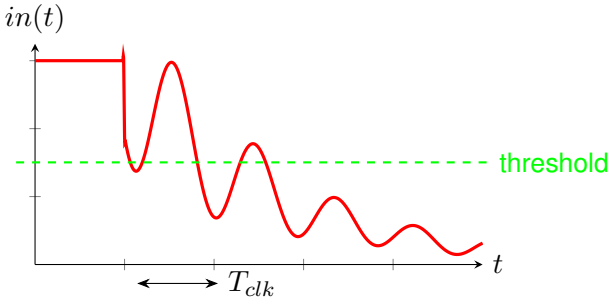
Bouncing Inputs

- 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



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



Counter Measures

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

Counter Measures

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

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

Counter Measures

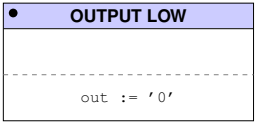
- Simply filter-out sequence of input transitions that is “too fast”
  - Analog (low pass) filtering
  - Digital filtering to check if output stabilizes
    - Use timer to wait (FSM)
    - Alternatives exist
  - Software-based debouncing

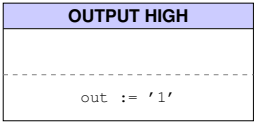
Debouncer Implementation

- Digital debouncing FSM

Debouncer Implementation

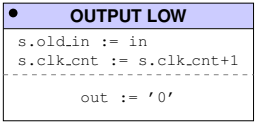
- Digital debouncing FSM
  - Debouncer either outputs zero or high

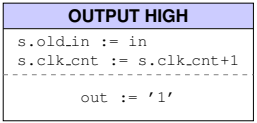




Debouncer Implementation

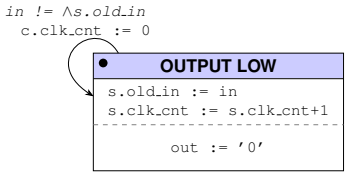
- Digital debouncing FSM
  - Debouncer either outputs zero or high

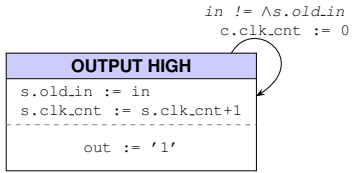




Debouncer Implementation

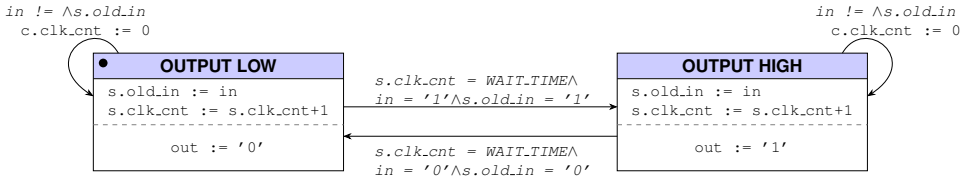
- Digital debouncing FSM
  - Debouncer either outputs zero or high
  - If the input changes, reset counter to count time since transition





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



Lecture Complete!