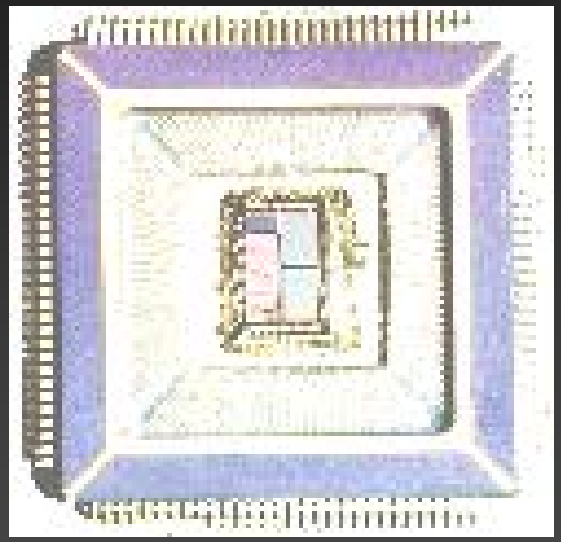


Hardware Modeling

VHDL Synthesis

Vienna University of Technology
Department of Computer Engineering
ECS Group

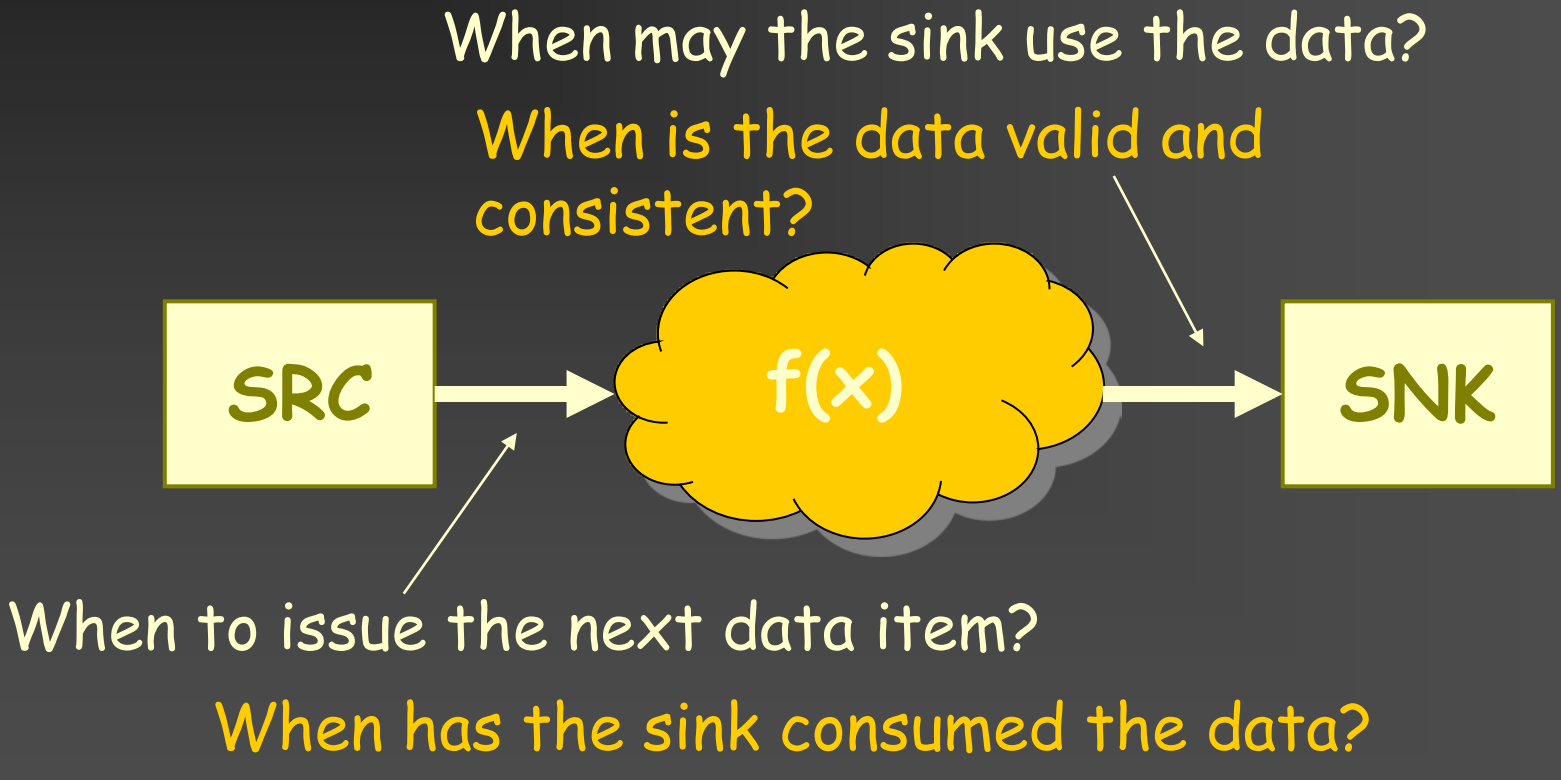
Contents

- Synchronous design style
- Reset and external inputs
- Two process method
- State machines
- Platform specific components

Synchronous Design Style

- Single central clock signal
- All events triggered by this clock
- Powerful toolchains available
- Nearly all commercial circuits implemented in synchronous style

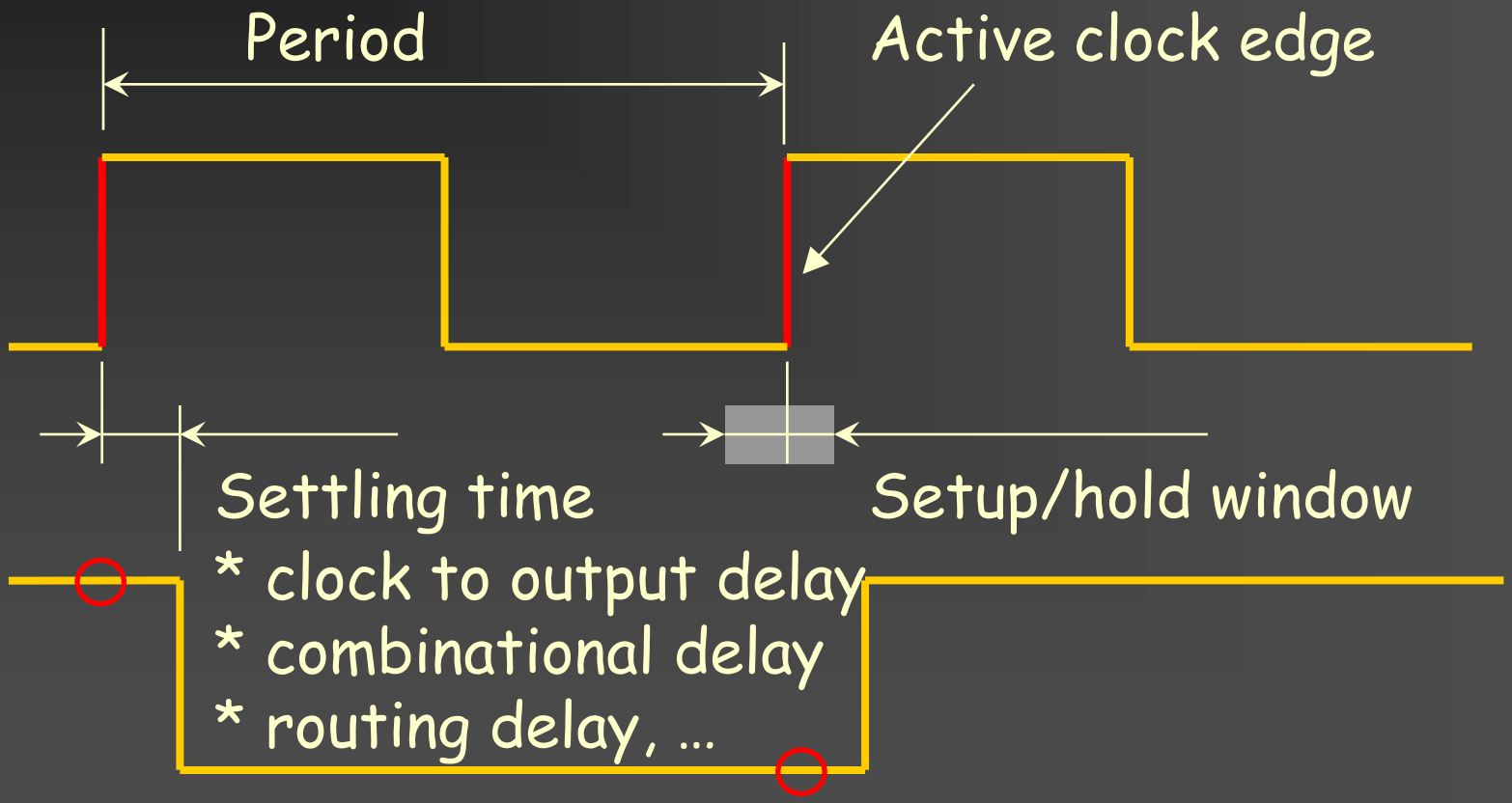
The Problem


When may the sink use the data?
When is the data valid and consistent?
SRC
f(x)
SNK
When to issue the next data item?
When has the sink consumed the data?

Setup- and Hold-Time

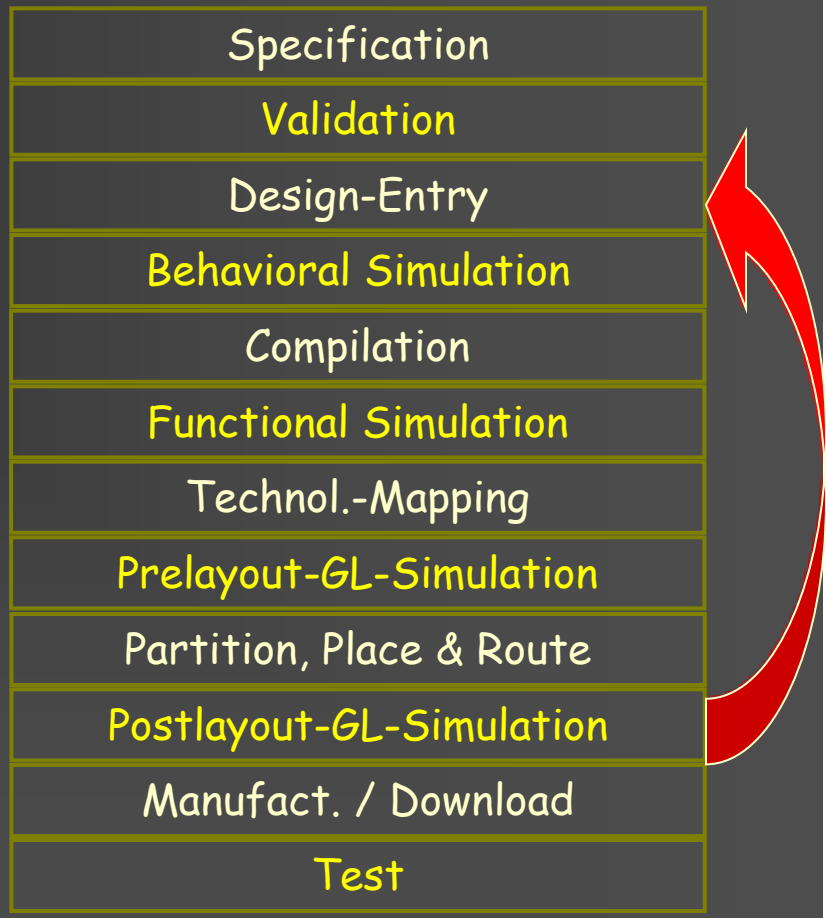
- How long must the data be stable before the active clock edge? – Setup time (t_{su})
- How long must the data be valid after the active clock edge? – Hold time (t_h)
- How long does the data need to reach the sink? - Settling time

Synchronous Timing

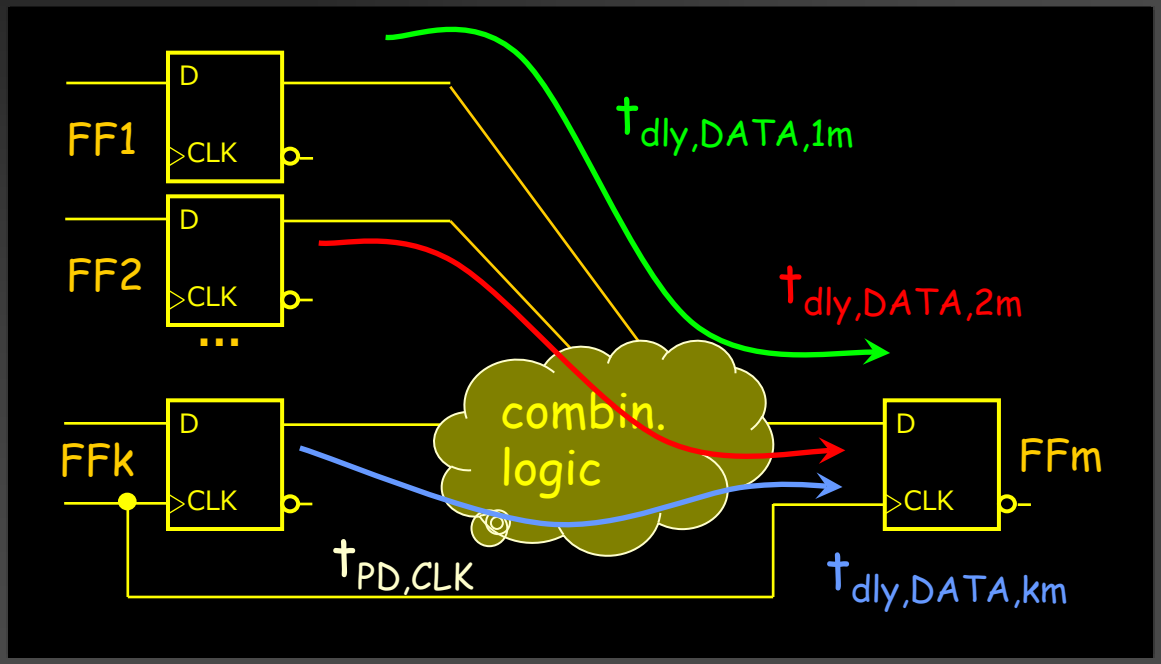


Timing Analysis

- Possible only at the END of the design-flow (Large iteration loop!)
- Enormous complexity
- Only feasible with ideal clock net



Timing Analysis



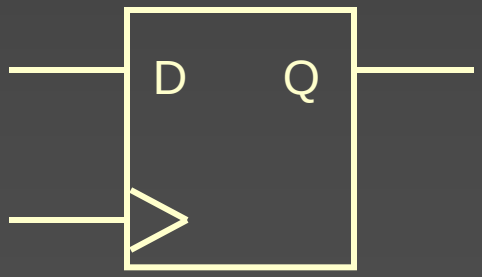
Synchronous Design - Problems

- Slowest path determines clock speed
- All activity within a short period of time (EMI, power consumption, …)
- Complex timing analysis needed
- Single violation may lead to a system failure (no graceful degradation)

Example: Coding a D-FlipFlop

```
entity d_ff is
  port
  (
    sys_clk : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end entity d_ff;
```

```
architecture beh of d_ff is
begin
  process(sys_clk)
    if rising_edge(sys_clk) then
      q <= d;
    end if;
  end process;
end architecture beh;
```



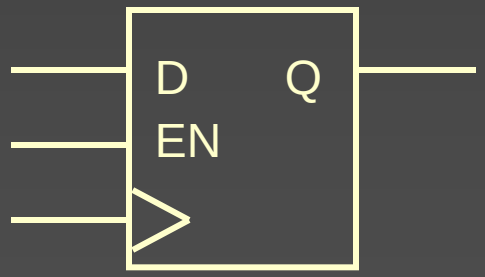
Clock Enable

- How to prevent an output change at an active clock edge?
- Gated clock
  - Bad design style (at least for FPGAs)
- Clock enable signal
  - Not part of the clock net
  - Direct implementation vs. multiplexer

Example: Adding clock enable

```
entity d_ff is
  port
  (
    sys_clk : in std_logic;
    sys_clk_en : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end entity d_ff;
```

```
architecture beh of d_ff is
begin
  process(sys_clk)
    if rising_edge(sys_clk) then
      if sys_clk_en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end architecture beh;
```



Contents

- Synchronous design style
- Reset and external inputs
- Two process method
- State machines
- Platform specific components

External Inputs

- May change their value at arbitrary times (asynchronous)
- Synchronizer necessary
  - Number of stages depend on required dependability
  - For uncritical systems: Two-flop synchronizer state of the art

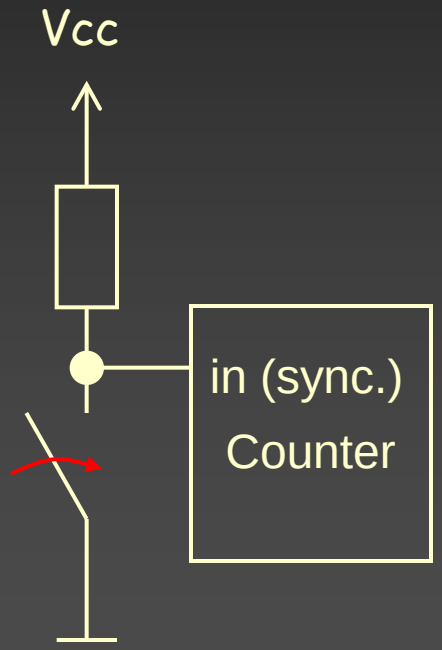
Example: Coding a Synchronizer

```
entity sync is
  generic
  (
    SYNC_STAGES : integer := 2
  );
  port
  (
    sys_clk : in std_logic;
    sys_res_n : in std_logic;
    input : in std_logic;
    output : out std_logic
  );
end entity sync;
```

```
architecture beh of sync is
  signal sync :
    std_logic_vector(1 to SYNC_STAGES);
begin
  process(sys_clk, sys_res_n)
    if sys_res_n = '0' then
      sync <= (others => '0');
    elsif rising_edge(sys_clk) then
      sync(1) <= input;
      for i in 2 to SYNC_STAGES loop
        sync(i) <= sync(i - 1);
      end loop;
    end if;
  end process;
  output <= sync(SYNC_STAGES);
end architecture beh;
```

Switches and Buttons

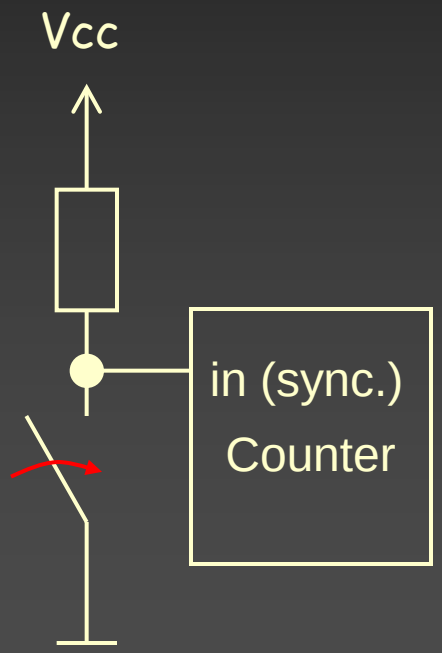
- Consider the following circuit:



- How many events does the counter see, if the button is pressed once?

Switches and Buttons

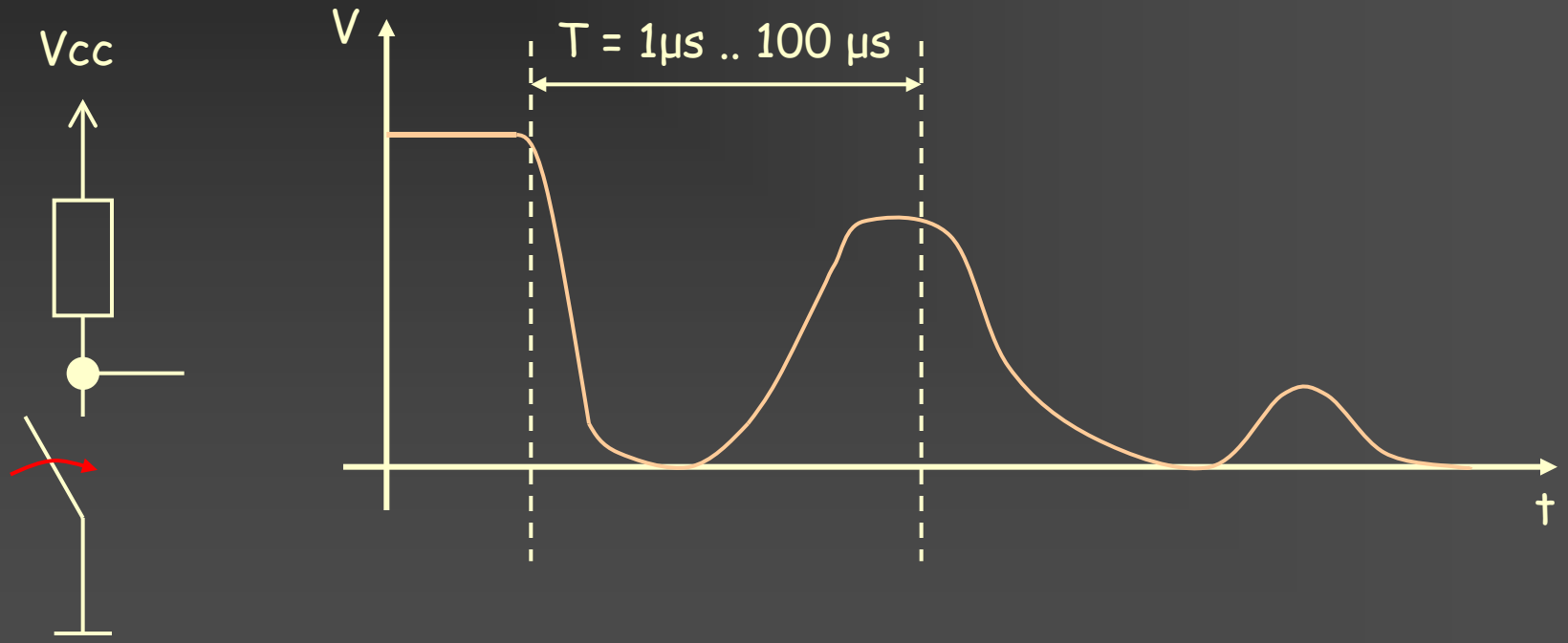
- Consider the following circuit:



- How many events does the counter see, if the button is pressed once?
- More or equal than one!
- Cause: bouncing

Bouncing

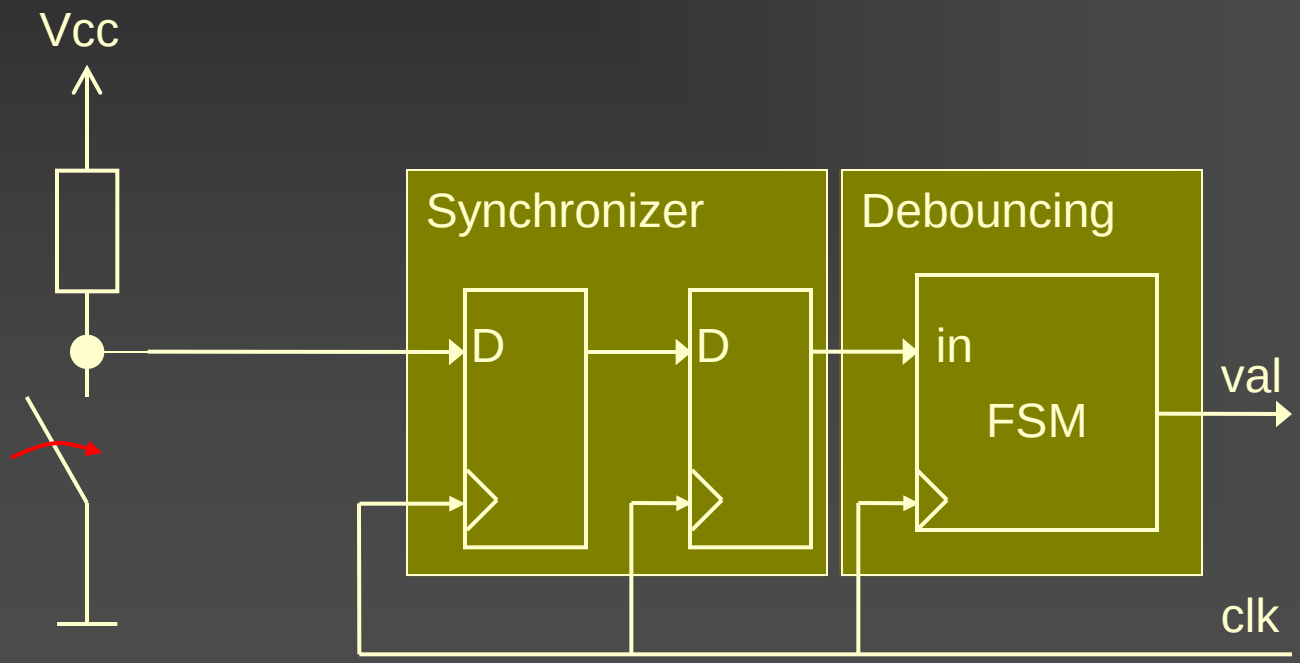
- Imperfect switching!



Up to 10 cycles !!

Debouncing

- Buttons/Switches must be debounced



Reset

- Two different kinds of reset
  - Synchronous Reset
  - Asynchronous Reset
- Reset Polarity
  - Active high
  - Active low

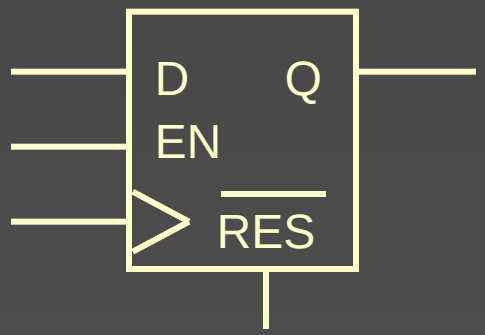
Reset

- Reset = external input
  ➔Synchronization necessary
- If a button is used
  ➔Debouncing necessary
- Synchronization & Debouncing normally implemented in the top level module
  - Use two-flop synchronizer without reset

D-FlipFlop with Async. Reset

```
entity d_ff is
  port
  (
    sys_clk : in std_logic;
    sys_res_n : in std_logic;
    sys_clk_en : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end entity d_ff;
```

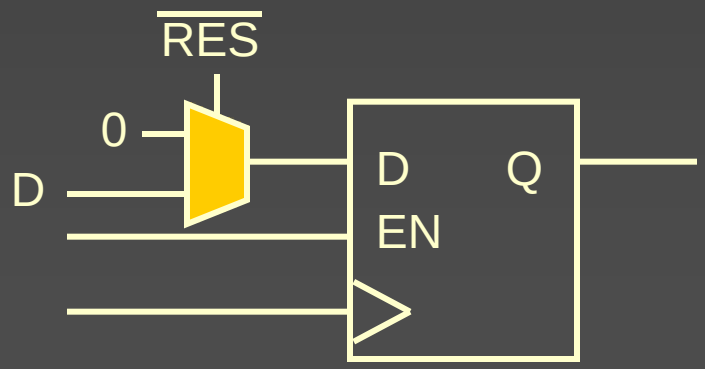
```
architecture beh of d_ff is
begin
  process(sys_clk, sys_res_n)
    if sys_res_n = '0' then
      q <= '0';
    elsif rising_edge(sys_clk) then
      if sys_clk_en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end architecture beh;
```



D-FlipFlop with Sync. Reset

```
entity d_ff is
  port
  (
    sys_clk : in std_logic;
    sys_res_n : in std_logic;
    sys_clk_en : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end entity d_ff;
```

```
architecture beh of d_ff is
begin
  process(sys_clk)
    if rising_edge(sys_clk) then
      if sys_clk_en = '1' then
        if sys_res_n = '0' then
          q <= '0';
        else
          q <= d;
        end if;
      end if;
    end if;
  end process;
end architecture beh;
```



Contents

- Synchronous design style
- Reset and external inputs
- Two process method
- State machines
- Platform specific components

Two Process Method

- Separate combinational and sequential logic
  - Asynchronous process: combinational logic
  - Synchronous process: registers
- Combinational logic calculates next value
- Synchronous process stores results

Synchronous Process

```
process(sys_clk, sys_res_n)
begin
  if sys_res_n = '0' then
    -- Set reset values
  elsif rising_edge(sys_clk) then
    -- Store next values
  end if;
end process;
```

- Stores next values into the registers
- Reset handling (sync. or async.)
- Sensitivity List: only clock and reset signal

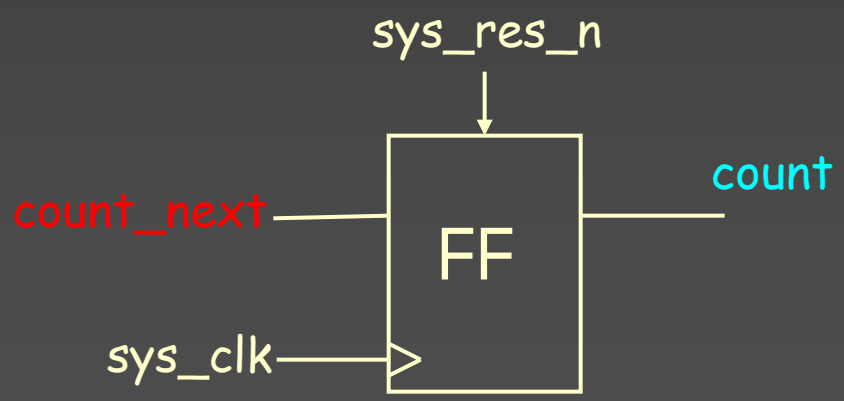
Asynchronous Process

- NO edge triggered elements!
- Sensitivity list: Contains all read signals
- Use default values to prevent latches
- Use only „stable“ signals (register outputs)
- Use short signal paths only

Example: Counter

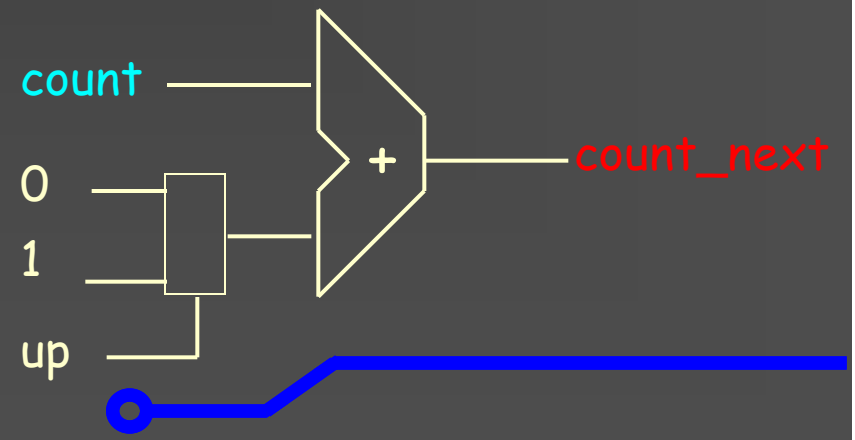
Synchronous process

```
process(sys_clk, sys_res_n)
begin
  if sys_res_n = '0' then
    count <= (others =>'0');
  elsif rising_edge(sys_clk) then
    count <= count_next;
  end if;
end process;
```



Asynchronous process

```
process(up, count)
begin
  count_next <= count;
  if (up = '1') then
    count_next <=
      std_logic_vector(
        unsigned(count) + 1);
  end if;
end process;
```



Contents

- Synchronous design style
- Reset and external inputs
- Two process method
- State machines
- Platform specific components

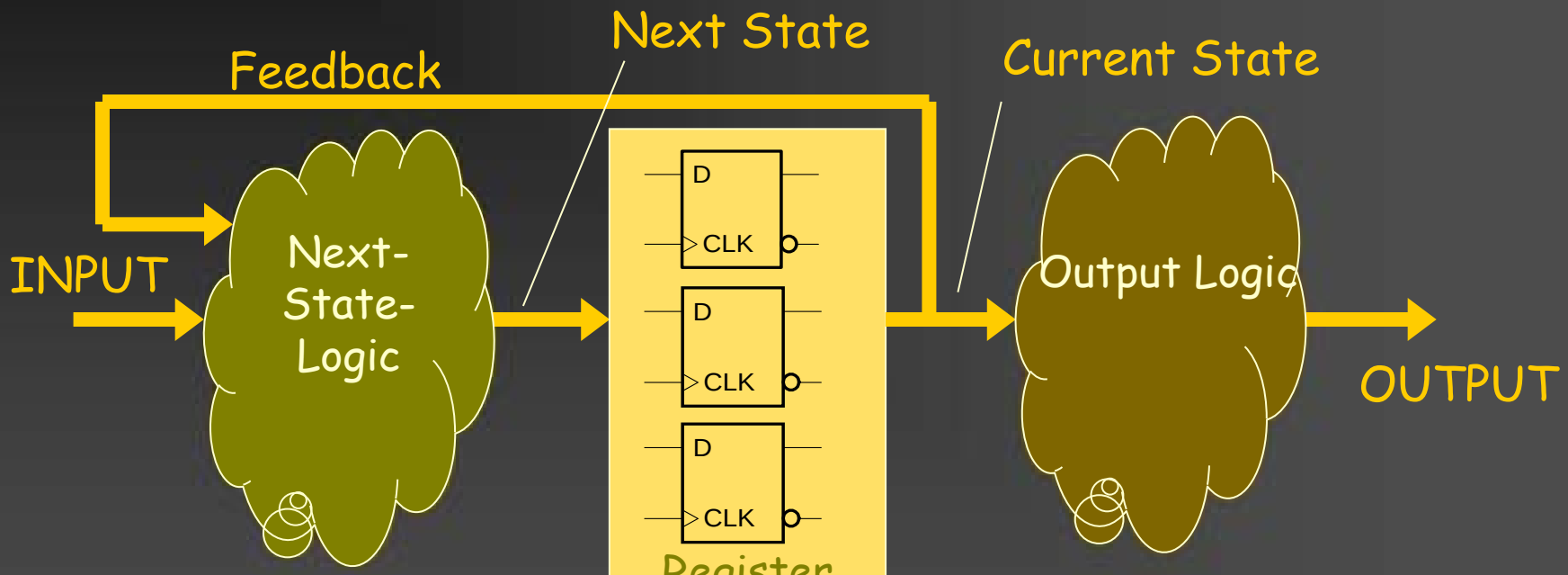
Finite State Machines (FSM)

- Theory
  - Mealy
  - Moore
- Design (state chart)
- VHDL coding
  - Three process method

FSM Principles

- Sequence of states
- Only synchronous state changes allowed
- State changes based on current state and input signals
- Output signals depend only on the current state (Moore state machine).
- Asynchronous path from input to output logic possible (Mealy state machines)

Moore-State Machine

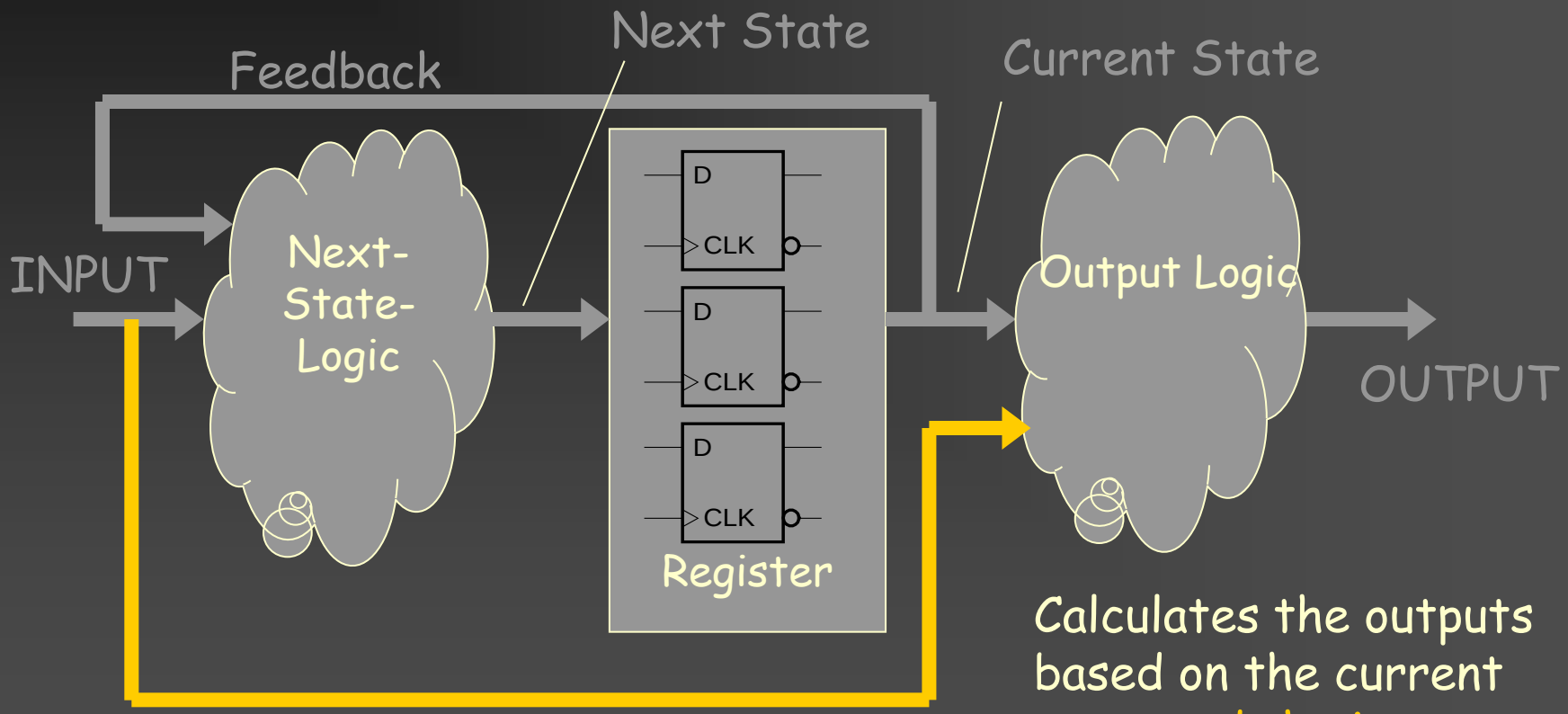


Calculates the next state based on the current one and the input signals

Synchronous state switch

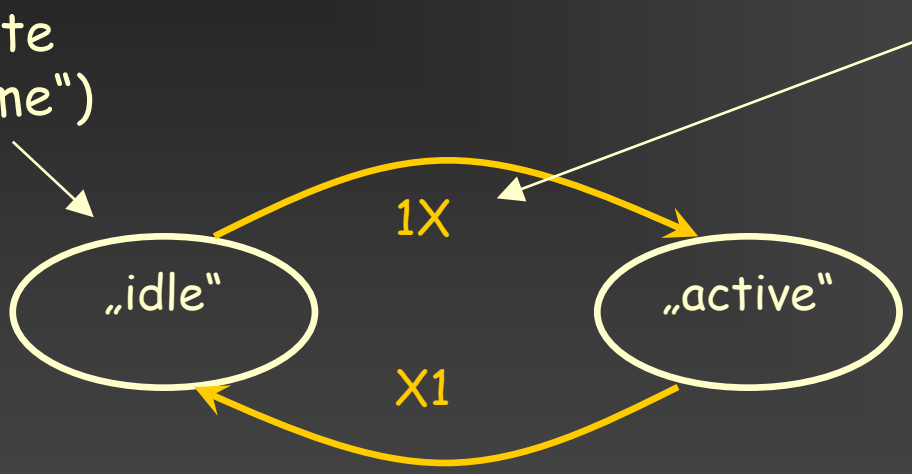
Calculates the outputs based on the current state

Mealy-State Machine



Calculates the outputs based on the current state and the input signals

State Charts for Moore FSMs



Notation for the state change conditions (list of input signals)

(trigger, sleep)

State	Outputs
idle	0
active	1

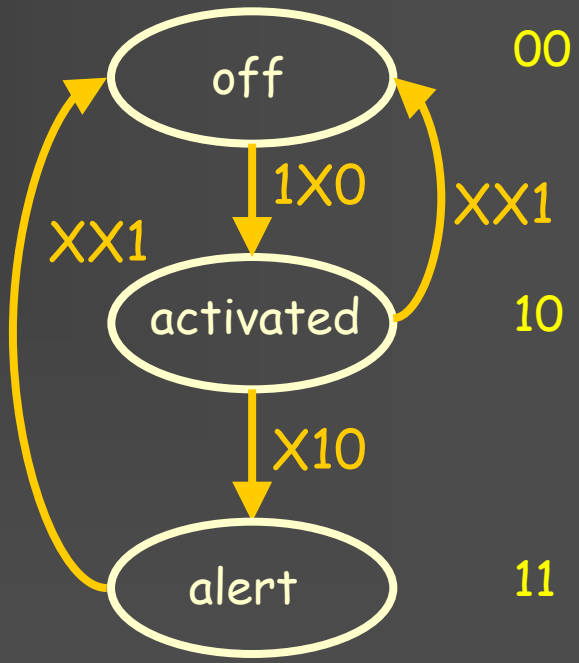
State to output mapping (list of output signals)

Example: Burglar Alarm

Inputs: activate button, door contact, code panel

Outputs: activation led, alarm siren

State	Output	Condition	Next state
off	00	1X0	activated
activated	10	XX1	off
		X10	alert
alert	11	XX1	off



FSM VHDL Coding

- Three process method
  - Synchronous process
  - Asynchronous next state process
  - Asynchronous output process
- Use case statement for state differentiation
- Use enumeration for state names

FSM VHDL Template (1)

```
type STATE_TYPE is (AAA, BBB, ....);
signal state, state_next : STATE_TYPE;
```

```
process(sys_clk, sys_res_n)
begin
  if sys_res_n = '0' then
    -- Set reset state
    state <= AAA;
  elsif rising_edge(sys_clk) then
    -- Store next state
    state <= state_next;
  end if;
end process;
```

FSM VHDL Template (2)

```
process(state, input1, …)
Begin
  -- Set a default value
  -- for next state
  state_next <= state;
  -- Calculate the next
  -- state
  case state is
    when AAA =>
      state_next <= XXX;
    when BBB =>
      if input1 = ‘1‘ then
        state <= CCC;
      end if;
    ....
  end case;
end process;
```

```
process(state)
begin
  -- Set default values
  -- for the outputs
  output1 <= ‘0‘;
  ....
  -- Calculate the outputs
  -- based on the current
  -- state
  case state is
    when AAA =>
      output1 <= ‘1‘;
    when BBB =>
      ....
  end case;
end process;
```

FSM VHDL Example (1)

```
type STATE_TYPE is (OFF, ACTIVATED, ALERT);
signal state, state_next : STATE_TYPE;
```

```
process(sys_clk, sys_res_n)
begin
  if sys_res_n = '0' then
    -- Set reset state
    state <= OFF;
  elsif rising_edge(sys_clk) then
    -- Store next state
    state <= state_next;
  end if;
end process;
```

FSM VHDL Example (2)

```
process(state, active_btn, door_contact, code_panel)
Begin
  -- Set a default value for next state
  state_next <= state;
  -- Calculate the next state
  case state is
    when OFF =>
      if active_btn = '1' and code_panel = '0' then
        state_next <= ACTIVATED;
      end if;
    when ACTIVATED =>
      if code_panel = '1' then
        state_next <= OFF;
      elsif door_contact = '1' then
        state_next <= ALERT;
      end if;
    when ALERT =>
      if code_panel = '1' then
        state_next <= OFF;
      end if;
  end case;
end process;
```

FSM VHDL Example (3)

```
process(state)
begin
  -- Set default values for the outputs
  activation_led <= '0';
  alarm_siren <= '0';
  -- Calculate the outputs based on the current state
  case state is
    when OFF =>
      null;
    when ACTIVATED =>
      activation_led <= '1';
    when ALERT =>
      activation_led <= '1';
      alarm_siren <= '1';
  end case;
end process;
```

Contents

- Synchronous design style
- Reset and external inputs
- Two process method
- State machines
- Platform specific components

Platform Specific Components

- PLL, PCI-Express endpoints, …
- Encapsulate behind common entity (wrapper)
- Use configuration to select right implementation
- Use documentation examples/wizards to instantiate the component in your wrapper

Summary

- Synchronous design style
  - Single clock
  - Timing analysis
- Reset and external inputs
  - Synchronizer
  - Debouncing
- Two process method
  - Separate sequential and combinational logic

Summary

- State machines
  - Sequence of states
  - Synchronous state change
  - Mealy vs. Moore
  - Three process method
- Platform specific components
  - Wrapper
  - Use wizards