Hardware Modeling [VU] (191.011)
– WS25 –
Finite-State Machine Basics

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

This lecture gives a first introduction to finite-state machines as digital circuits, their components and how they can look like in VHDL. It forms the basis for upcoming lectures on state machine modeling and implementation.

FSM Basics
Introduction
FSM Circuits
Example: Timer

Hardware Modeling [VU] (191.011)
– WS25 –
Finite-State Machine Basics

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:06 (f8a58e9)

Fundamental structure in digital designs

You should already have encountered finite-state machines – also called finite-state automata or simply state machines – in other courses, where they have probably been introduced as abstract mathematical models for computations. However, in this lecture, we will not focus on the theory of automata, but rather look into concrete circuit implementations of this concept. Finite-state machines – or FSMs in short – are a fundamental structure in digital design. Understanding them is crucial for developing efficient, reliable, and maintainable hardware, as they allow for the abstraction of complex processes into manageable sub-tasks and components.

Introduction

- Fundamental structure in digital designs

- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function

On a basic level, an FSM is defined by a set of inputs, outputs, and states, as well as a transition function that defines how the system moves from one state to another based on the current state and the input values. The outputs are set according to the current state and, in some instances, also the input values.

Introduction

- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function

- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function
- Synchronous FSMs: State changes **only** happen at clock events!

Since we are talking about digital circuits, the inputs and outputs are sets of binary signals. The states are encoded into binary values as well and the current state is stored in some sequential circuit element. Because we want to implement synchronous circuits, this element is implemented using flip-flops, which are clocked by a common clock signal to ensure that all state transitions occur synchronously to the clock. Thus, at each active clock edge, the values stored by the flip-flops that hold the state are updated according to the transition function.

Introduction

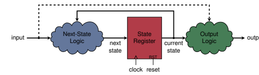
- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function
- Synchronous FSMs: State changes **only** happen at clock events!

- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function
- Synchronous FSMs: State changes **only** happen at clock events!
- Important VHDL concepts used in the lecture
  - Enumeration and record types
  - Behavioral modeling of combinational logic
  - Sequential circuit elements (i.e., registers)

Note that this lecture, and the following ones on modeling and implementing FSMs, heavily rely on other topics we have already covered. In particular, we will use enumeration and record types, behavioral circuit modeling and sequential circuit elements. If you cannot recall these topics please refer back to the respective lectures before starting your journey into the world of finite-state machines, as these topics are crucial for understanding the upcoming content.

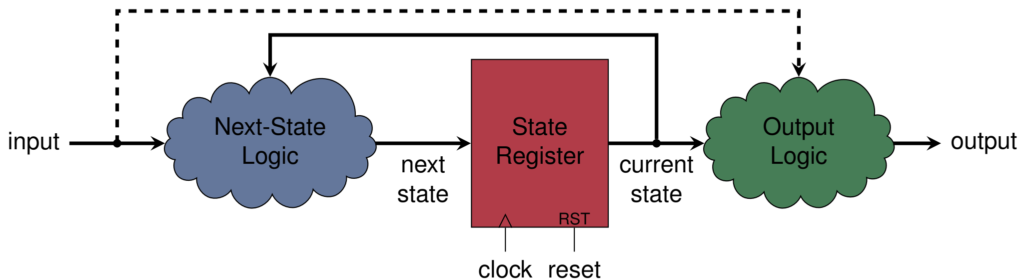
Introduction

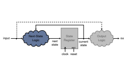
- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function
- Synchronous FSMs: State changes **only** happen at clock events!
- Important VHDL concepts used in the lecture
  - Enumeration and record types
  - Behavioral modeling of combinational logic
  - Sequential circuit elements (i.e., registers)



This slide shows the general structure of a circuit implementing a finite-state machine. It consists of three main components: the next-state logic, the state register and the output logic. Please note that the cloud shapes are used for purely combinational components. Over the next few slides we will explain each of the individual parts in detail.

State Machine as Digital Circuits



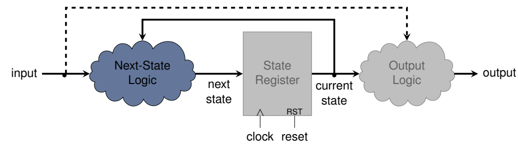


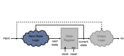
The next-state logic is a purely combinational circuit, meaning that it must not contain any sequential circuit elements. Hence, it cannot read or depend on the clock or reset signal in any way and – most importantly – must not contain any latches.

Next-State Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches



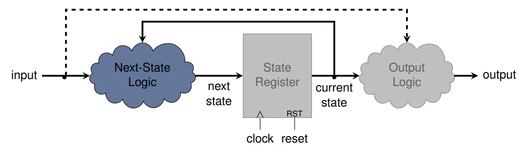


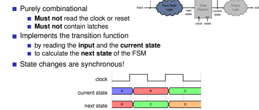
It implements the transition function of the FSM by mapping the applied inputs and the current state to the next state. Notice that the current state is provided by the output of the state register, while the next state signal is its input.

Next-State Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Implements the transition function
  - by reading the **input** and the **current state**
  - to calculate the **next state** of the FSM

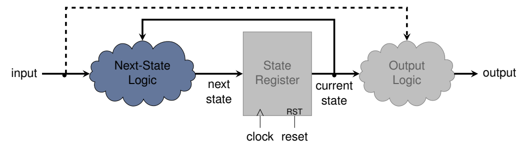




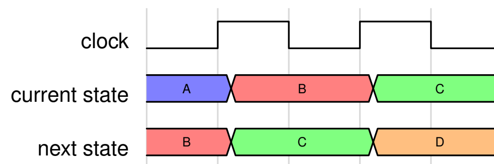
As a result state changes can only happen at the active clock edge, where the next state signal is captured by the state register and becomes the new current state. The timing diagram illustrates this behavior. It shows an FSM transitioning through the states A, B and C. At each rising clock edge the value of the next state signal becomes the current state.

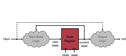
Next-State Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Implements the transition function
  - by reading the **input** and the **current state**
  - to calculate the **next state** of the FSM
- State changes are synchronous!



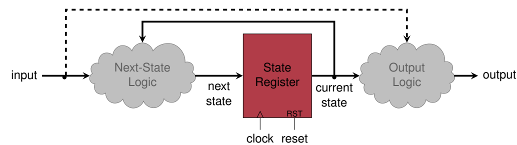


- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**

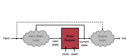
As already mentioned, the state register stores the actual state of the FSM and is the **only** sequential circuit element and implemented using a set of D flip-flops.

State Register

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



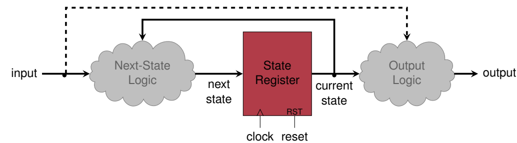
- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**



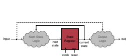
At this point we want to stress that, as always, you must strictly adhere to the code patterns for flip-flops discussed in the lecture about sequential circuit elements to avoid the introduction of bugs or non-synthesizable code.

State Register

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**
- **Strictly** stick to the code patterns for registers!



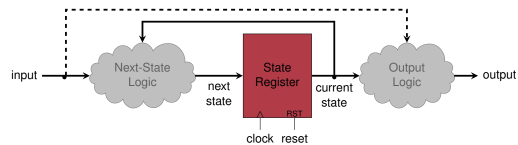
- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**
- **Strictly** stick to the code patterns for registers!
- Reset value defines initial state

The reset value of this register defines the initial state of the FSM. The reset can be implemented in a synchronous or asynchronous way. However, in this course we consistently only use the asynchronous variant as discussed in a previous lecture.

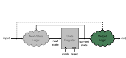
State Register

HWMod
WS25

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**
- **Strictly** stick to the code patterns for registers!
- Reset value defines initial state

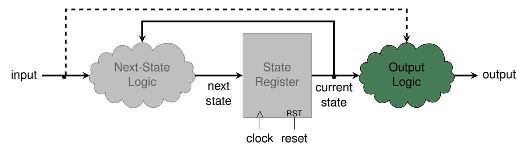


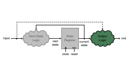
Exactly like the next-state logic, the output logic is also a purely combinational circuit and it must also under no circumstances contain latches or depend on the clock.

Output Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches



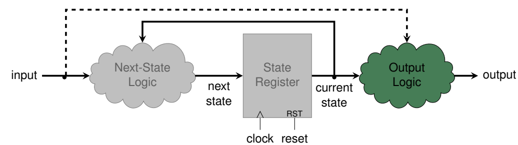


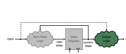
As its name suggests its purpose is to produce the actual output signals of the FSM.

Output Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals

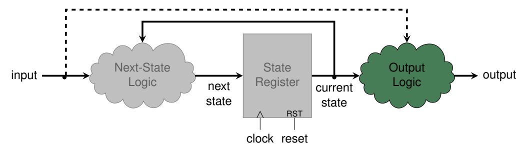




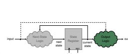
It also defines the state machine's type. Depending on whether the output logic only reads the current state or the current state **and** the inputs, the FSM is either classified as a Moore or a Mealy automaton, respectively.

Output Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



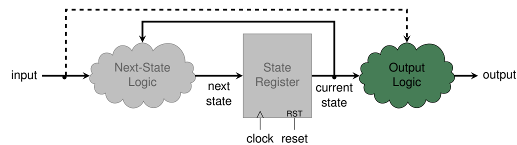
- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals
- Defines the FSM type
  - Moore: Output depends solely on **current state**
  - Mealy: Output depends on **current state** and **input**



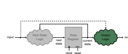
This type basically affects the way an FSM can react to input changes. In Mealy machines, as illustrated by the figure, there is a combinational path from the input to the output. Therefore, the circuit can adjust its output based on an input change in the same clock cycle without the need to wait for an active clock edge. In contrast, Moore machines only derive their outputs from their current state. Hence, they can only react to input changes after the next clock edge, that is, in the next cycle. Moreover, it is sometimes possible to implement the same functionality with a fewer number of states when using a Mealy instead of a Moore machine.

Output Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



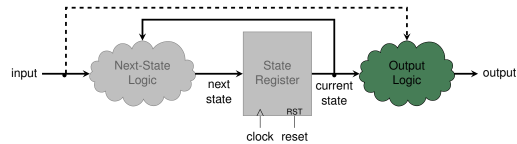
- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals
- Defines the FSM type
  - Moore: Output depends solely on **current state**
  - Mealy: Output depends on **current state** and **input**
- Mealy machines
  - can react to input changes in the **same** clock cycle (i.e., combinationally)
  - (sometimes) require fewer states than a similar Moore machine



Since the next-state and the output logic operate on overlapping sets of inputs, it is in practice often the case that logic can be shared between these two components. This holds true for both types of state machines, Mealy and Moore. However, this is nothing you need to worry about too much, as the synthesis tool will figure this out for you.

Output Logic

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer



- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals
- Defines the FSM type
  - Moore: Output depends solely on **current state**
  - Mealy: Output depends on **current state** and **input**
- Mealy machines
  - can react to input changes in the **same** clock cycle (i.e., combinationally)
  - (sometimes) require fewer states than a similar Moore machine
- Can share logic with the next-state logic

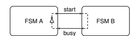
- Interacting Mealy FSMs can lead to combinational loops!

Although it might seem that Mealy machines have clear advantages over Moore machines, care must be taken as they can easily introduce combinational loops into a system.

Output Logic - Combinational Loops

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

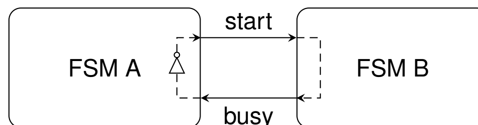
- Interacting Mealy FSMs can lead to combinational loops!



Consider the shown example, of two FSMs that interact with each other using a ”start” and a ”busy” signal. FSM A uses the start signal to initiate some operation in FSM B, while FSM B uses the busy signal to indicate to A that is ready to receive the start signal. Assume both A and B are Mealy machines. The output logic of A sets the start signal to high as soon as the busy signal it receives from B is low. At the same time, the output logic of B sets the busy signal to high as soon as it sees a high level on the start input. This will essentially lead to B simply forwarding the ”start” to its ”busy” signal, and to A negating the applied ”busy” signal. In this scenario we thus end up with a combinational loop formed by the output-logic of FSM A and B. A possible fix would be to implement B as a Moore state machine and only set the busy signal in the cycle after the start signal was set to high.

Output Logic - Combinational Loops

- Interacting Mealy FSMs can lead to combinational loops!
- Example
  - FSM A: “Start when not busy”
  - FSM B: “Assert busy when started”



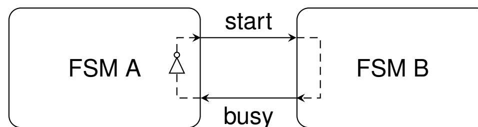
- Interacting Mealy FSMs can lead to combinational loops!
- Example
  - FSM A: "Start when not busy"
  - FSM B: "Assert busy when started"
- Only use Mealy machines when **really** necessary

In summary: Be aware of this potential issue and try to avoid Mealy machines unless, they are **really** necessary.

Output Logic - Combinational Loops

FSM Basics
Introduction
FSM Circuits
Next-State Logic
State Register
Output Logic
Example: Timer

- Interacting Mealy FSMs can lead to combinational loops!
- Example
  - FSM A: "Start when not busy"
  - FSM B: "Assert busy when started"



- Only use Mealy machines when **really** necessary

Behavior Description
The simple_timer module keeps an internal N-bit counter that is (synchronously) incremented as long as en is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and en is asserted the tick output is asserted.
Interface
entity simple_timer is generic (N : positive := 8); port (clk : in std_ulogic; res_n : in std_ulogic; en : in std_ulogic; tick : out std_ulogic); end entity;

As the introduction over the previous slides might have been a bit abstract, let's look at a simple concrete example to get a better understanding of the discussed components. The example that we will use for this is a very basic timer circuit, with the interface shown on the lower part of the slide. Its behavior is quite straightforward. The timer keeps an internal N-bit counter, which represents the state of the FSM. Whenever the `en` input is high, this counter is synchronously incremented. This is a task implemented by the next-state logic. When the counter reaches its maximum value ($2^N - 1$) it simply overflows to zero. The output `tick` is set to high whenever the maximum value is reached and the `en` input is high.

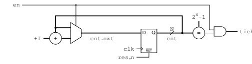
Simple Timer - Specification

Behavior Description

The `simple_timer` module keeps an internal `N`-bit counter that is (synchronously) incremented as long as `en` is asserted. When the counter reaches its maximum value ($2^N - 1$) it overflows back to zero. If the counter hits its maximum and `en` is asserted the `tick` output is asserted.

Interface

```
1  entity simple_timer is
2    generic (
3      N : positive := 8
4    );
5    port (
6      clk   : in std_ulogic;
7      res_n : in std_ulogic;
8      en    : in std_ulogic;
9      tick  : out std_ulogic
10   );
11 end entity;
```

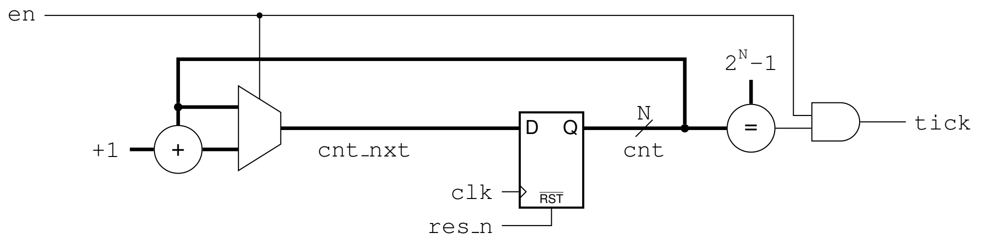


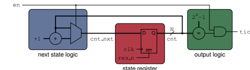
Here we see a circuit that implements the specification formulated on the previous slide. By now you should already recognize most of the shown circuit elements. However, for the sake of completeness: The circle-shaped components with the equals sign and plus sign represent an adder and a comparator, respectively.

Simple Timer - Circuit

HWMod
WS25

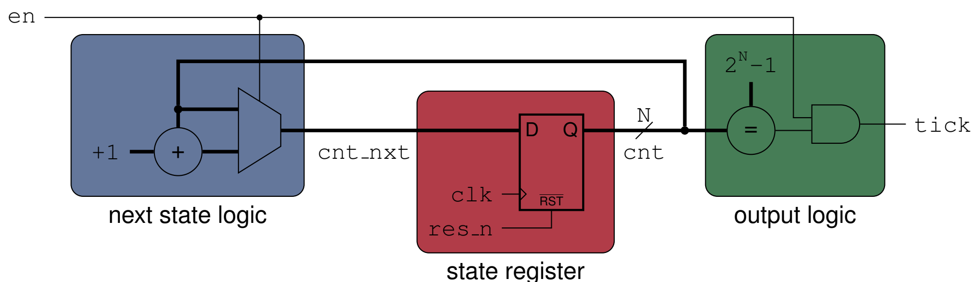
FSM Basics
Introduction
FSM Circuits
Example: Timer
Specification
Circuit
Implementation





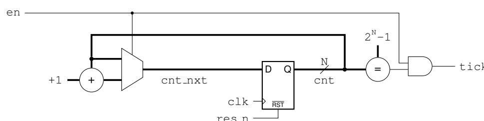
Of course, the central D flip-flops represent the state register. Its output, that is the signal labeled `cnt`, holds the current state of the FSM. It feeds the output as well as the next state logic. The next state logic uses a multiplexer to decide whether to increment the count value, based on the `en` input. It produces the `cnt_nxt` signal, which is connected to the input of the state register. At the next rising clock edge this signal will be captured by the state register, thus becoming the new state. Besides the current state, the output logic also reads the `en` input, making the circuit a Mealy state machine.

Simple Timer - Circuit



Let us now look at a possible architecture for the `simple_timer` entity.

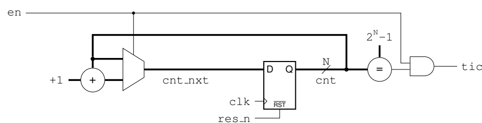
Simple Timer - Implementation



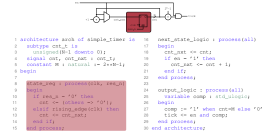

```vhdl
architecture arch of simple_timer is
  subtype cnt_t is
    unsigned(N-1 downto 0);
  signal cnt, cnt_nxt : cnt_t;
  constant M : natural := 2**N-1;
begin

  state_reg : process(clk, res_n)
  begin
    if res_n = '0' then
      cnt <= (others => '0');
    elsif rising_edge(clk) then
      cnt <= cnt_nxt;
    end if;
  end process;
  next_state_logic : process(all)
  begin
    cnt_nxt <= cnt;
    if en = '1' then
      cnt_nxt <= cnt + 1;
    end if;
  end process;

  output_logic : process(all)
    variable comp : std_ulogic;
  begin
    comp := '1' when cnt=M else '0';
    tick <= en and comp;
  end process;
end architecture;
```

To implement it, we first declare the signals `cnt` and `cnt_nxt`.

Simple Timer - Implementation

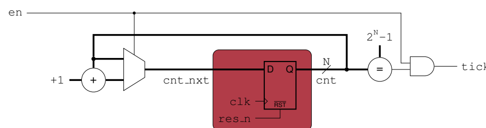



```vhdl
architecture arch of simple_timer is
  subtype cnt_t is
    unsigned(N-1 downto 0);
  signal cnt, cnt_nxt : cnt_t;
  constant M : natural := 2**N-1;
begin

  state_reg : process(clk, res_n)
  begin
    if res_n = '0' then
      cnt <= (others => '0');
    elsif rising_edge(clk) then
      cnt <= cnt_nxt;
    end if;
  end process;

  next_state_logic : process(all)
  begin
    cnt_nxt <= cnt;
    if en = '1' then
      cnt_nxt <= cnt + 1;
    end if;
  end process;

  output_logic : process(all)
    variable comp : std_ulogic;
  begin
    comp := '1' when cnt=M else '0';
    tick <= en and comp;
  end process;
end architecture;
```



Next, using these signals, we can implement the state register via a synchronous process. For that we simply follow the code pattern presented in the lecture about sequential circuit elements. During the asynchronous reset, the counter is set to zero, which defines the initial state of our FSM. On rising clock edges, we simply assign the next signal to the counter to update the state register.

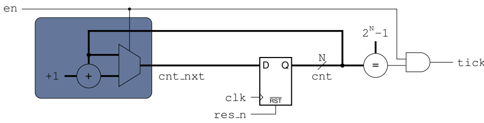
Simple Timer - Implementation



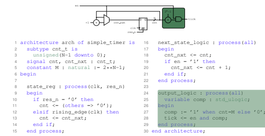

```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) then
13       cnt <= cnt_nxt;
14     end if;
15   end process;
16   next_state_logic : process(all)
17   begin
18     cnt_nxt <= cnt;
19     if en = '1' then
20       cnt_nxt <= cnt + 1;
21     end if;
22   end process;
23
24   output_logic : process(all)
25     variable comp : std_ulogic;
26   begin
27     comp := '1' when cnt=M else '0';
28     tick <= en and comp;
29   end process;
30 end architecture;
```

Let us now turn our attention to the next-state logic. Recall that this function is purely combinational and is hence also implemented in a purely combinational process. The required multiplexer is described using an if-condition which checks the `en` signal. Notice, that we only write to the "next" signal, as the task of the next state logic is to prepare the next value that will be captured into the state register.

Simple Timer - Implementation

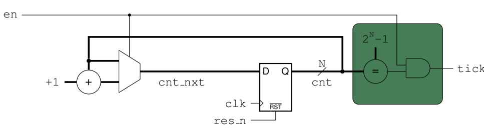



```
 1 architecture arch of simple_timer is
 2   subtype cnt_t is
 3     unsigned(N-1 downto 0);
 4   signal cnt, cnt_nxt : cnt_t;
 5   constant M : natural := 2**N-1;
 6 begin
 7
 8   state_reg : process(clk, res_n)
 9   begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) then
13       cnt <= cnt_nxt;
14     end if;
15   end process;
16   next_state_logic : process(all)
17   begin
18     cnt_nxt <= cnt;
19     if en = '1' then
20       cnt_nxt <= cnt + 1;
21     end if;
22   end process;
23
24   output_logic : process(all)
25     variable comp : std_ulogic;
26   begin
27     comp := '1' when cnt=M else '0';
28     tick <= en and comp;
29   end process;
30 end architecture;
```



Finally, the output logic is also implemented using its own process. As with the next state logic, this process must also be purely combinational. The constant `M`, representing the maximum value of the counter, is declared in the declaration part of the architecture.

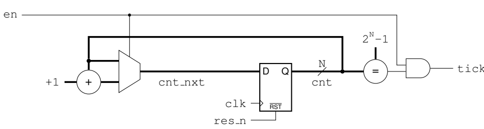
Simple Timer - Implementation




```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) then
13       cnt <= cnt_nxt;
14     end if;
15   end process;
16
17   next_state_logic : process(all)
18   begin
19     cnt_nxt <= cnt;
20     if en = '1' then
21       cnt_nxt <= cnt + 1;
22     end if;
23   end process;
24
25   output_logic : process(all)
26     variable comp : std_ulogic;
27   begin
28     comp := '1' when cnt=M else '0';
29     tick <= en and comp;
30   end process;
31 end architecture;
```

The FSM implementation style we used here is referred to as the 3-process-method, as it uses three separate processes for each of the three state machine components. As we will see in an upcoming lecture, there also exists a 2- and 1-process-method.

Simple Timer - Implementation




```
 1 architecture arch of simple_timer is
 2   subtype cnt_t is
 3     unsigned(N-1 downto 0);
 4   signal cnt, cnt_nxt : cnt_t;
 5   constant M : natural := 2**N-1;
 6 begin
 7
 8   state_reg : process(clk, res_n)
 9   begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) then
13       cnt <= cnt_nxt;
14     end if;
15   end process;
16   next_state_logic : process(all)
17   begin
18     cnt_nxt <= cnt;
19     if en = '1' then
20       cnt_nxt <= cnt + 1;
21     end if;
22   end process;
23
24   output_logic : process(all)
25     variable comp : std_ulogic;
26   begin
27     comp := '1' when cnt=M else '0';
28     tick <= en and comp;
29   end process;
30 end architecture;
```

Lecture Complete!

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

FSM Basics
Introduction
FSM Circuits
Example: Timer
Specification
Circuit
Implementation

Lecture Complete!