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

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:03 (f8a58e9)

Introduction

- Fundamental structure in digital designs

Introduction

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

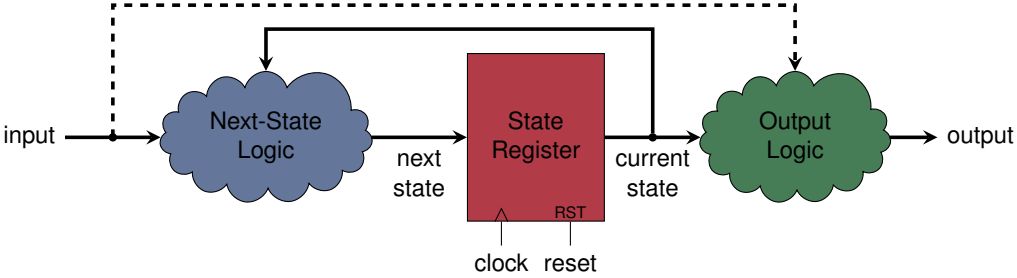
Introduction

- 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!

Introduction

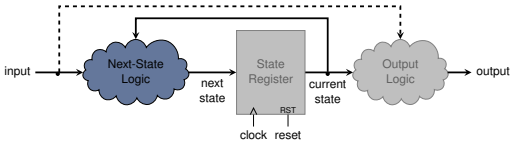
- Fundamental structure in digital designs
- An FSM consists of a
  - set of inputs, outputs and states
  - transition function
- Synchronous FSMs: State changes **only** happen at clock events!
- Important VHDL concepts used in the lecture
  - Enumeration and record types
  - Behavioral modeling of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

State Machine as Digital Circuits



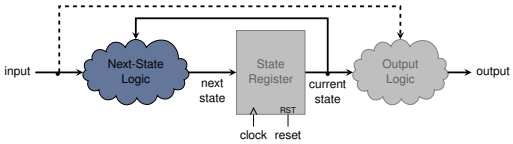
Next-State Logic

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

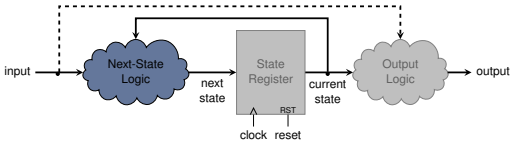


Next-State Logic

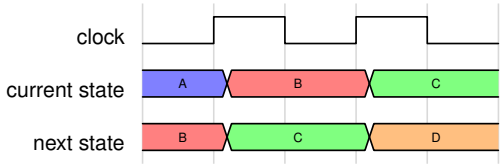
- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Implements the transition function
  - by reading the **input** and the **current state**
  - to calculate the **next state** of the FSM



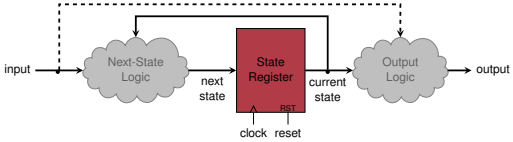
Next-State Logic



- 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!

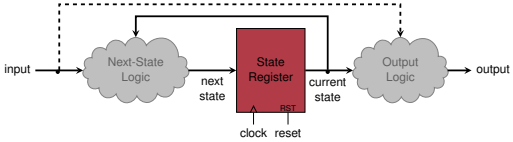


State Register



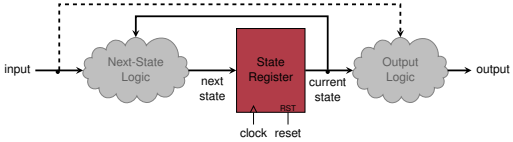
- Only synchronous component in the FSM
- Reads **next state** and outputs **current state**

State Register



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

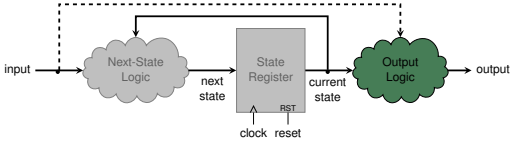
State Register



- 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

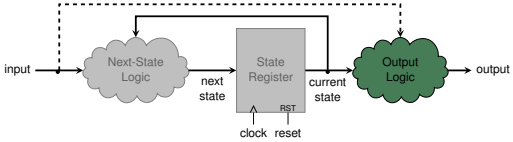
Output Logic

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

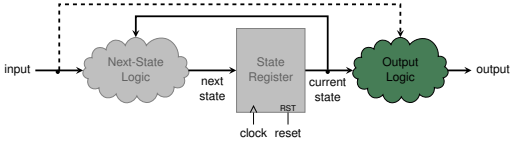


Output Logic

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

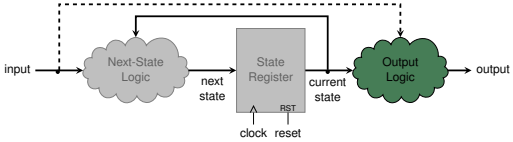


Output Logic



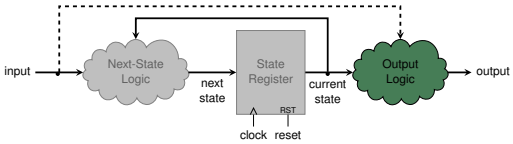
- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals
- Defines the FSM type
  - Moore: Output depends solely on **current state**
  - Mealy: Output depends on **current state** and **input**

Output Logic



- 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

Output Logic



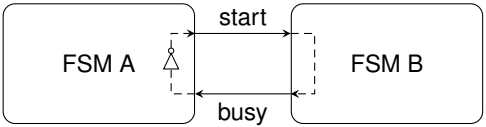
- Purely combinational
  - **Must not** read the clock or reset
  - **Must not** contain latches
- Produces the actual **output** signals
- Defines the FSM type
  - Moore: Output depends solely on **current state**
  - Mealy: Output depends on **current state** and **input**
- Mealy machines
  - can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

Output Logic - Combinational Loops

- Interacting Mealy FSMs can lead to combinational loops!

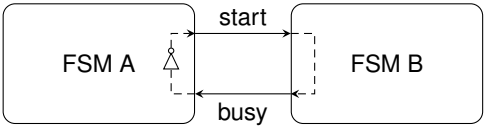
Output Logic - Combinational Loops

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



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”



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

Simple Timer - Specification

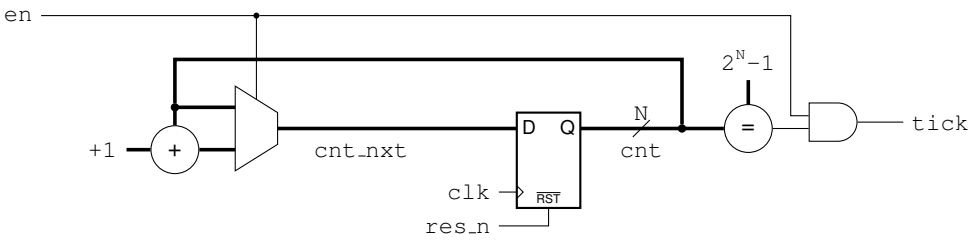
Behavior Description

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.

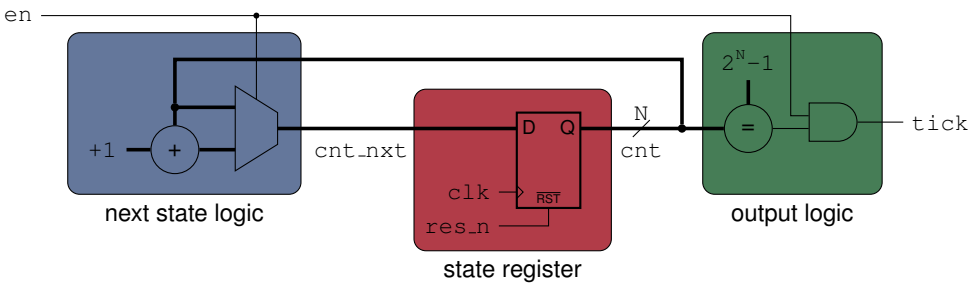
Interface

```
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;
```

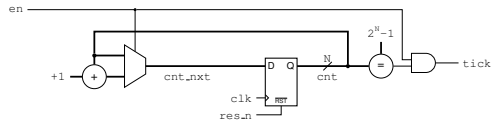
Simple Timer - Circuit



Simple Timer - Circuit

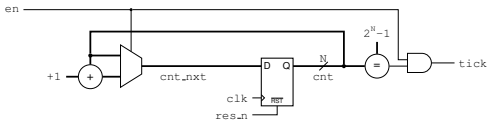


Simple Timer - Implementation



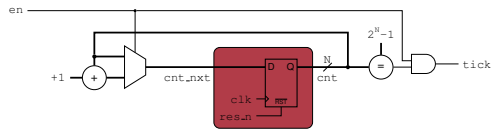

```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) 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;
```

Simple Timer - Implementation



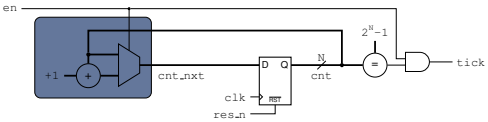

```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) 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;
```

Simple Timer - Implementation



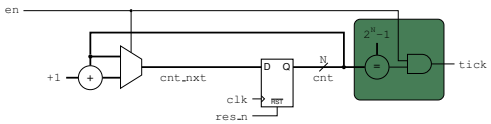

```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) 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;
```

Simple Timer - Implementation



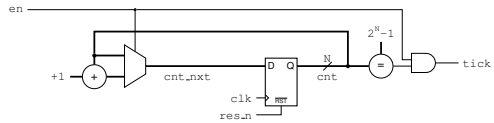

```
1  architecture arch of simple_timer is
2    subtype cnt_t is
3      unsigned(N-1 downto 0);
4    signal cnt, cnt_nxt : cnt_t;
5    constant M : natural := 2**N-1;
6  begin
7
8    state_reg : process(clk, res_n)
9    begin
10     if res_n = '0' then
11       cnt <= (others => '0');
12     elsif rising_edge(clk) 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;
```

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;
```

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;
```

Lecture Complete!