VHDL

5051158-xxxx

Lecture 5
State machines
PWM

V 1.0, 28.10.2020
Jarno Tuominen

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Contents of this lecture

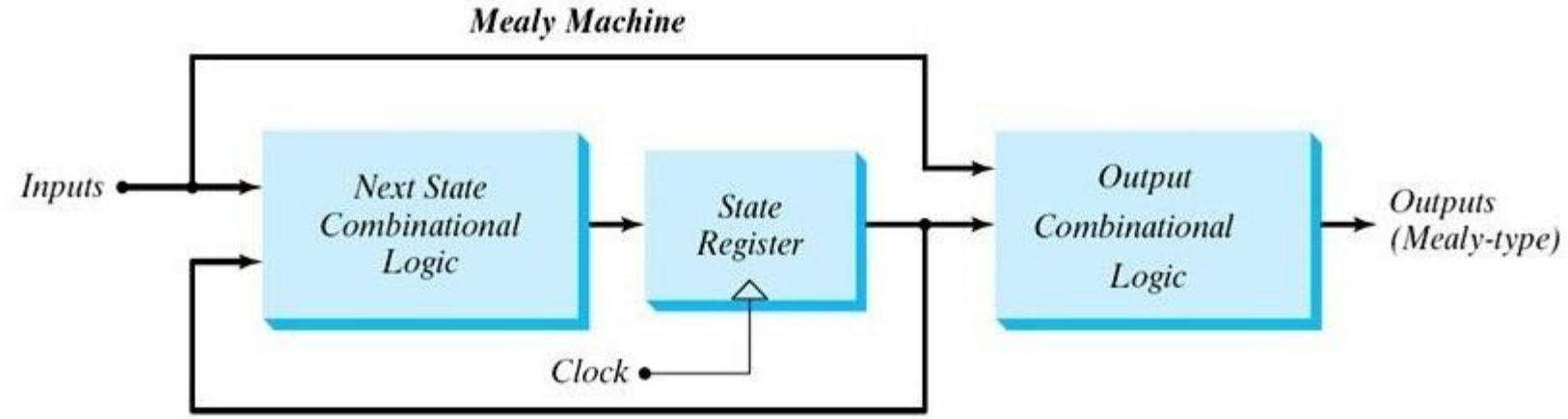
- (Finite) State Machines
  - Principles, classification
  - Implementation in VHDL
- PWM (Pulse Width modulator)
  - Principle of operation
  - Implementation in VHDL

Finite State Machines

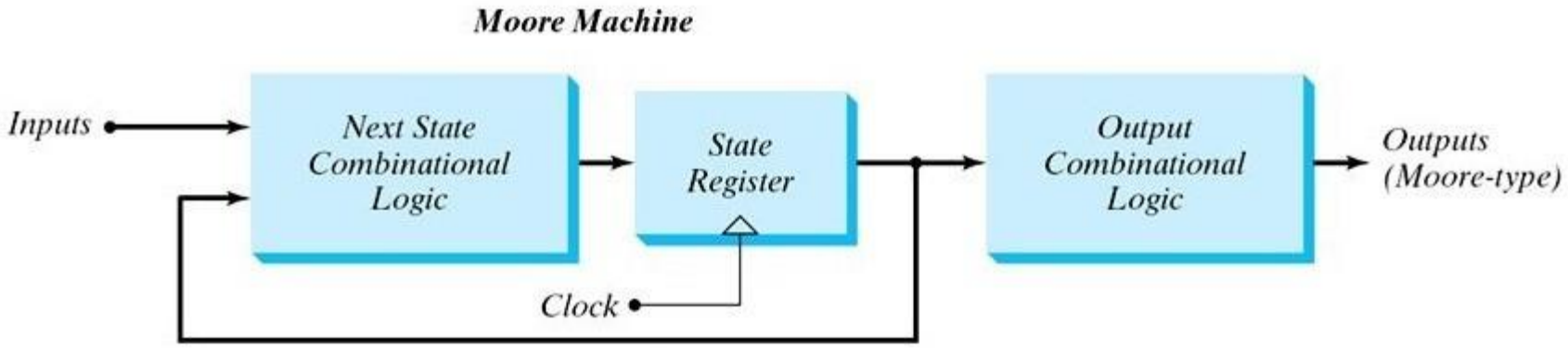
- State machine is a sequential design unit, with a “memory” - representing the **current state** of the machine
  - Memory in practice is D-Flip-Flop()s
  - Poll: Have you ever designed a state machine?
- State machines are usually the hardest part of logic design
  - Extra attention must be put to ensure that actions happen on a correct clock cycle
- Two basic types of FSM’s:
  - Mealy
  - Moore
- When making VHDL presentation of FSM, you need to decide:
  - The description style - how many processes?
  - Encoding of states (synthesis can do this automatically)

Moore vs Mealy

- Mealy FSM: The outputs depend on current state AND inputs



- Moore: The outputs depend only on current state

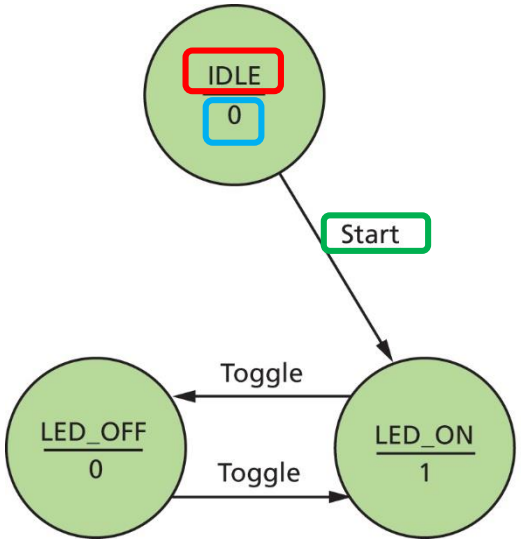


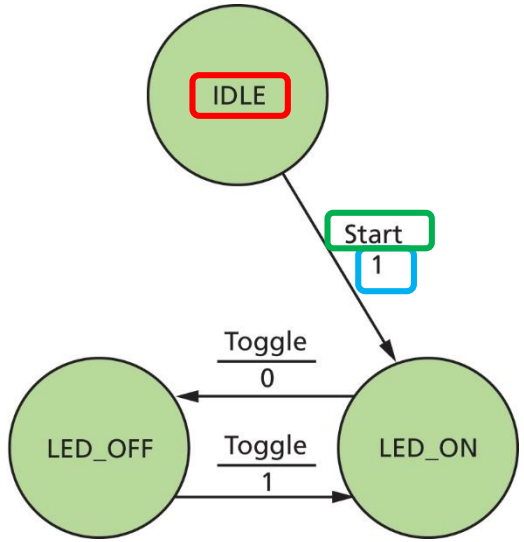
How to start FSM design?

- The first step is to draw a state diagram
- It shows the states, the transitions between the states and the outputs from the state machine

Moore:

- Outputs defined only by state
- Always 1 clock cycle delay from input to output





Mealy:

- Outputs defined by state AND inputs
- Reacts to input immediately

Note: You don't have to stick only to Moore Mealy types - you can write a hybrid, having both flavors

Implementation styles

- 1 Sequential process
  - Everything packed in a single process
  - If inputs affect output -> Mealy, otherwise Moore.

```
sync_all: process(clk,rst_n) begin
  if rst_n = '0' then
    --INIT STATE and OUTPUTs of the FSM
  elsif clk'event and clk = '1' then
    --Define new value of curr state
    --Define outputs. All these outputs
    --become registers!
  end if;
end process sync_all;
```

- 2 processes
  - 1 (sequential) process for determining the next state
  - 1 (combinatorial) process for driving output, based on current state and inputs (in case of Mealy machine)
  - Note: You can make 2nd process clocked as well - then your output will be 1 cycle delayed. This might be required because of strict timing requirements.

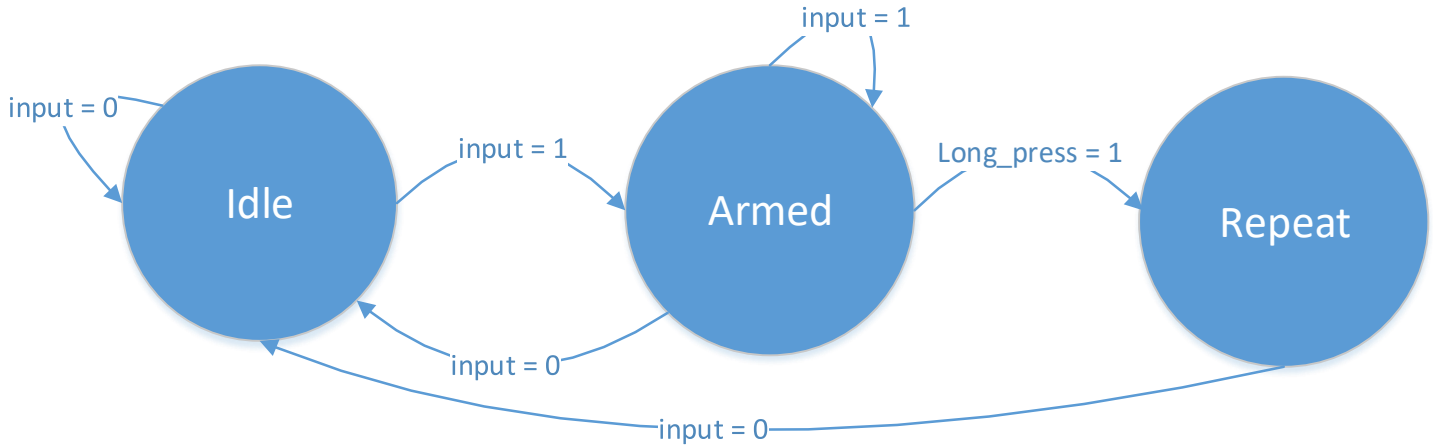
```
sync_ps: process(clk,rst_n) begin
  if rst_n = '0' then
    --INIT STATE of the FSM
  elsif clk'event and clk = '1' then
    --Synchronous part of the FSM,
    --assign next state to curr state
  end if;
end process sync_ps;

comb_output: process(curr_state,input) begin
  --Combinational part; define outputs
  --Moore looks same as Mealy, but does not
  --consider input when determining the output
end process comb_output;
```

FSM implementation in VHDL (1)

- Start by defining an own, enumerated type for the machine states
- Create a signal based on this type - this would hold the “current state”

```
type pulser_state_t is (Idle, Armed, Repeat);
signal pulser_state: pulser_state_t;
```



FSM implementation in VHDL (2)

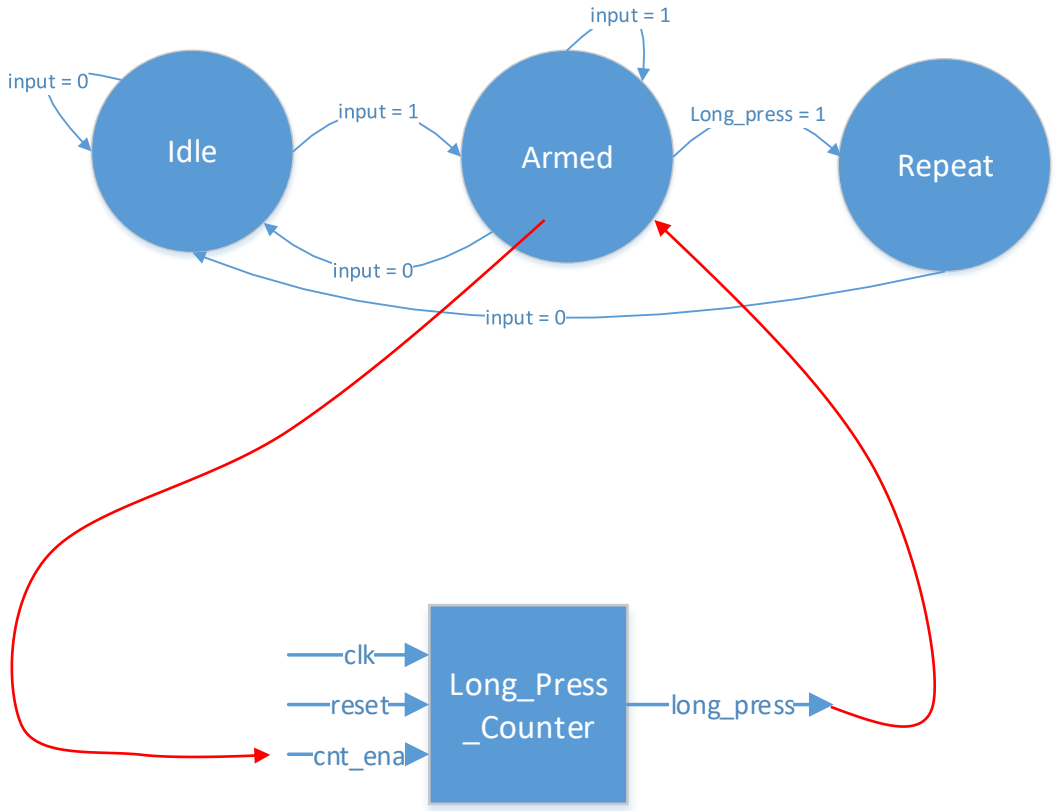
- At reset, define:
  - the initial state
  - the initial output
- On each clock cycle:
  - Evaluate the next state. Note: as we are inside a clocked process, the state variable is “in effect” on the next clock cycle
  - Define the outputs
- Be careful not to create dead locks (=states without exit)

```
button_pulser: process (clock, n_Reset) begin
  if n_Reset = '0' then
    output <= '0';
    pulser_state <= Idle;
  elsif clock'event and clock = '1' then
    case pulser_state is
      when Idle => -- no news yet!
        if input = '1' then
          output <= '1'; -- generate a single shot
          pulser_state <= Armed;
        end if;
      when Armed => -- waiting for long press to happen
        output <= '0';
        if (long_press = '1') then pulser_state <= Repeat;
      when Repeat => -- button pressed, generating repeated pulses
        if (long_press = '0') then -- exit when button released
          pulser_state <= Idle;
        end if;
      when others =>
        pulser_state <= Idle; -- just in case
    end case;
  end if; --clk/rst
end process button_pulser;
```

FSM implementation in VHDL (3)

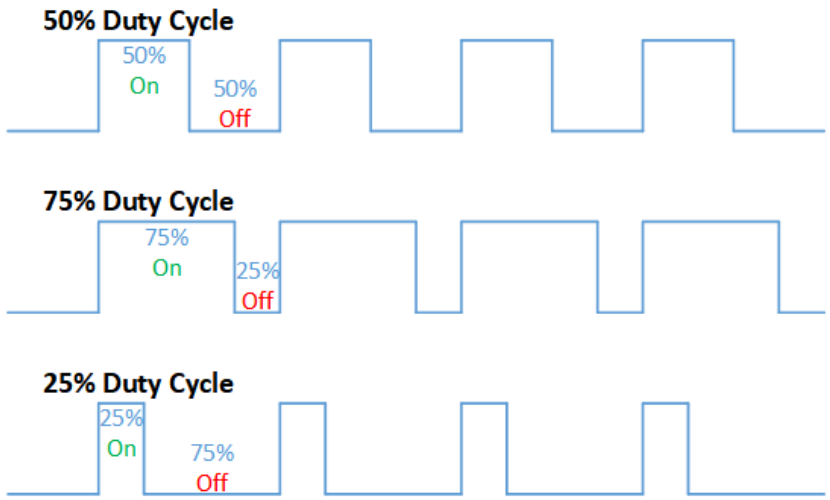
- Wait… how do we know when long press took place?

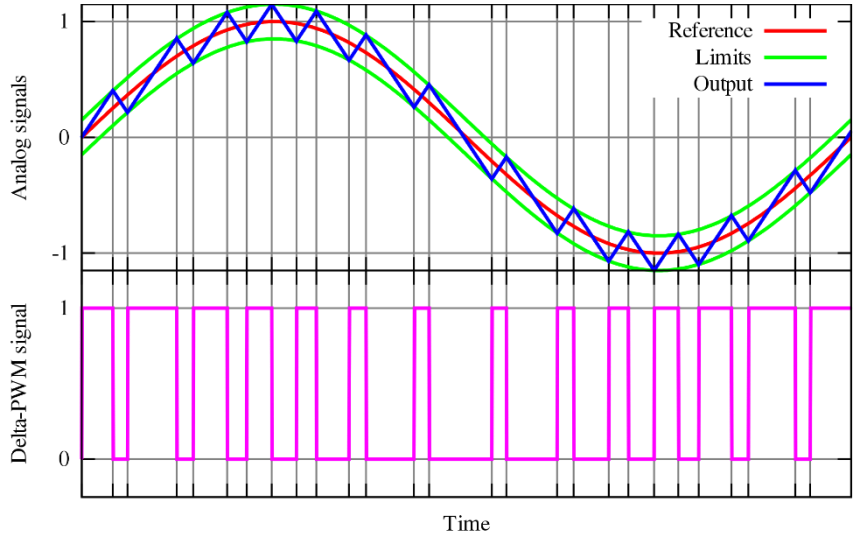
- -> Create an additional counter (own process), which has count_enable input
- Count_enable is driven by the state machine - when the state is “Armed”, the “long press counter” runs
- Advise: make this counter a separate component, where the max count value is configurable via generic. Why? Because you need one for repeat counter as well.



Pulse Width Modulator

- PWM is a component (or a modulation method), which alters the ratio between “on-time” and “off-time” of a signal. This ratio is also called **duty cycle**
- The key is the **average** value of the PWM signal - it tells how much power is transferred to the load during a cycle
- It can be used for several purposes, like:
  - Dimming the lights (LEDs)
  - Motor control
  - Audio amplifiers (D-class)



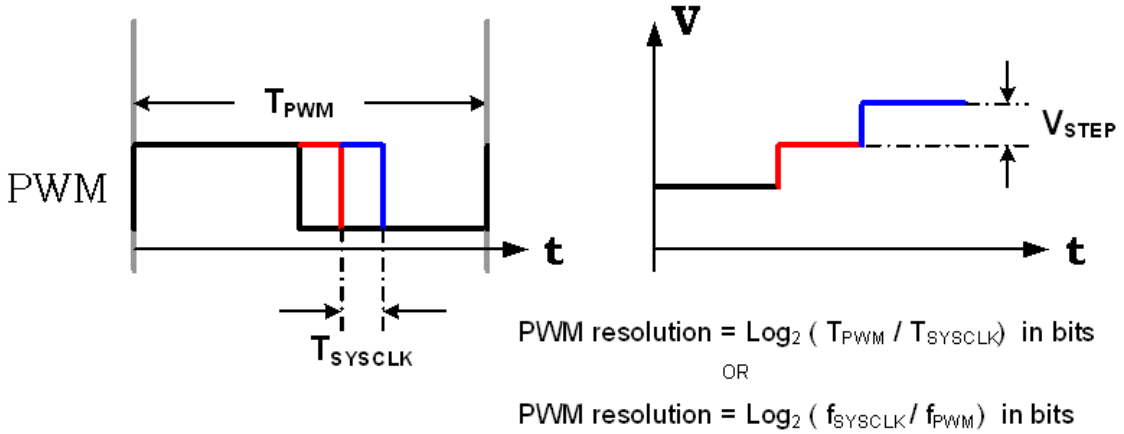


https://en.wikipedia.org/wiki/Pulse-width_modulation

Pulse Width Modulator (2)

- Two main parameters define the operation of a PWM:
  - PWM Rate / Switching frequency: Defines PWM cycles/time unit
  - PWM resolution: How many discrete steps you have from zero output (power) to full output (power)
- There is a relation between the system clock frequency, PWM rate and resolution(R):

$$R = Log_2\left[\frac{T_{PWM} \times \frac{V_{OUT}}{V_{IN}}}{T_{SYSCLK}}\right]$$



Voltage Adjustment (Resolution) as a Function of System Clock and PWM Frequency

PWM (KHz)	System Clock (MHz)			
	50	%	100	%
100	9.0	0.20	10.0	0.10
200	8.0	0.40	9.0	0.20
400	7.0	0.80	8.0	0.40
500	6.6	1.00	7.6	0.50
700	6.2	1.40	7.2	0.70
1000	5.6	2.00	6.6	1.00
1500	5.1	3.00	6.1	1.50
2000	4.6	4.00	5.6	2.00

Resolution Values in Bits and Percentages (%) for Various System and PWM Frequencies

PWM implementation in VHDL

- PWM controller can be implemented simply with Modulo-N-counters
- The design steps:
  1. Decide the PWM rate
  2. Decide the PWM resolution (in bits)
  3. Create a clock signal (a clock divider), where the output clock (PWM system clock) is PWM_Rate * PWM_steps (2^resolution_in_bits). This is actually a modulo-N-counter
  4. Create another counter, which is clocked by the PWM system clock, wraps over at PWM max_value (if resolution is 2^N you don't have to do anything - just let it wrap over
  5. Create an output logic: When PWM_counter is less than PWM-value, output is '0', otherwise '1'. PWM value is here an input to this design block.
- You can implement this all as a single module - or hierarchically so that PWM system clock generator is separated from the actual PWM