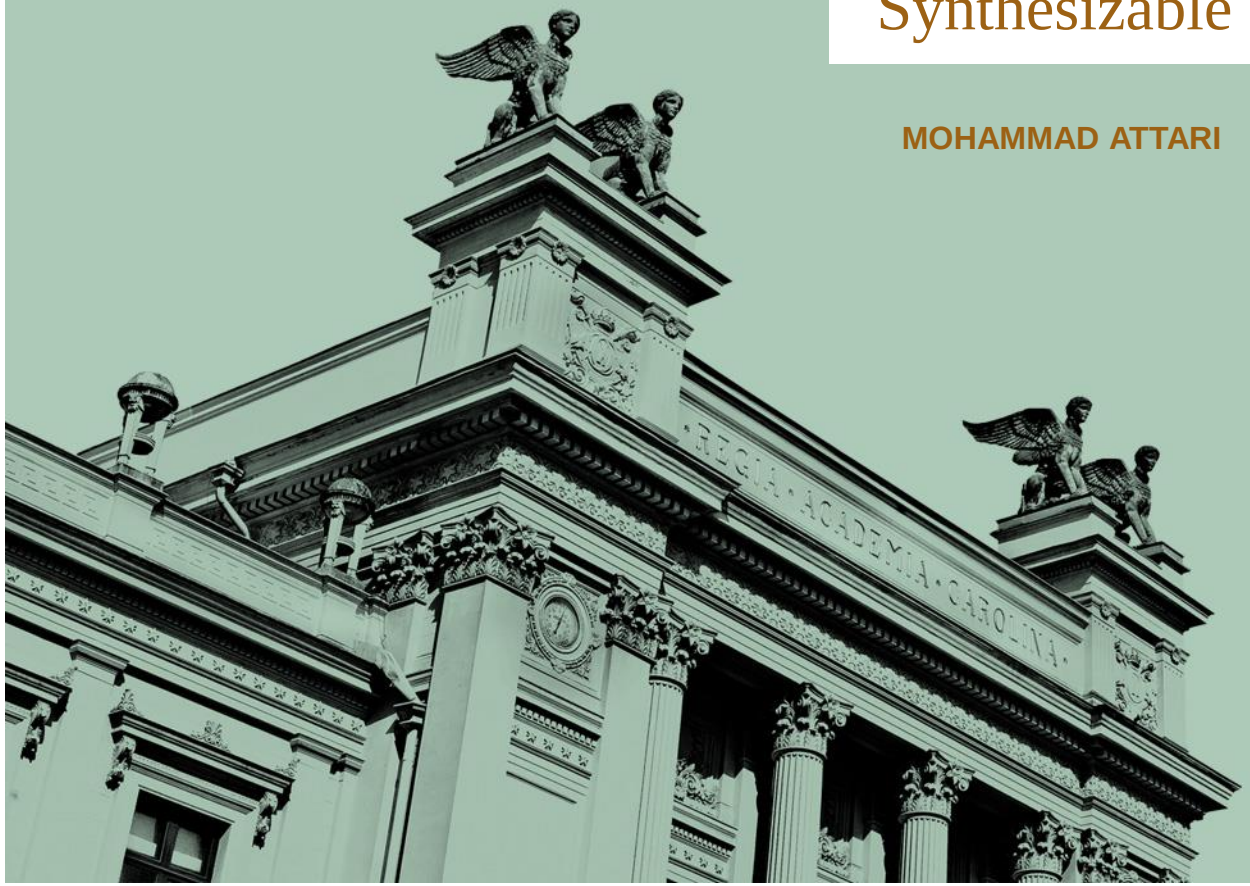
# Introduction to Structured VLSI Design

## Synthesizable VHDL Coding Styles

MOHAMMAD ATTARI



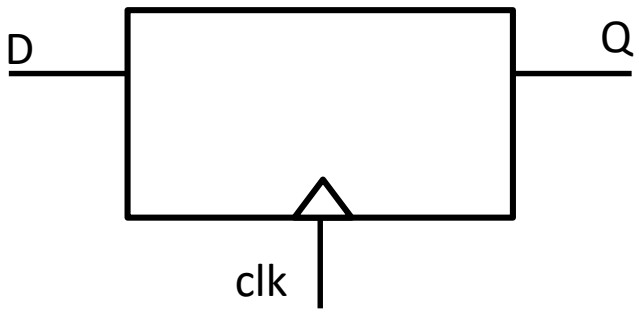
# Two Coding Styles

- There are 2 styles of writing synthesizable VHDL code
  1) One-process
  2) Two-process
- Which one is better?
- Let's consider two simple examples
  - A flipflop
  - A counter
- Let's do it using both styles
  - For simplicity libraries and reset signal are omitted

# Example 1, Flipflop – One-process Method

```
entity flipflop is
     port(clk : in std_logic;
          D : in std_logic;
          Q : out std_logic);
end flipflop;

architecture behavior of flipflop is
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               Q <= D;
          end if;
     end process;
end behavior;
```

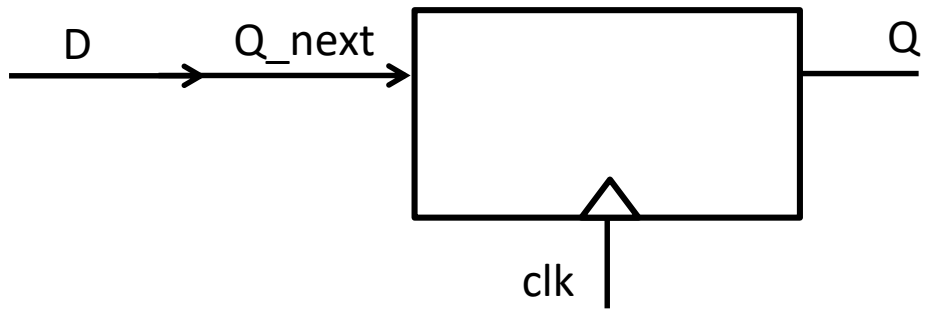


# Example 1, Flipflop – Two-process Method

```
entity flipflop is
     port(clk : in std_logic;
          D : in std_logic;
          Q : out std_logic);
end flipflop;

architecture behavior of flipflop is
     signal Q_next : std_logic;
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               Q <= Q_next;
          end if;
     end process;

     process(D)
     begin
          Q_next <= D;
     end process;
end behavior;
```



# Example 1 Assessment

- Previous codes both synthesize into a flipflop
- But which one is preferred?
- To answer that let’s design an up-counter

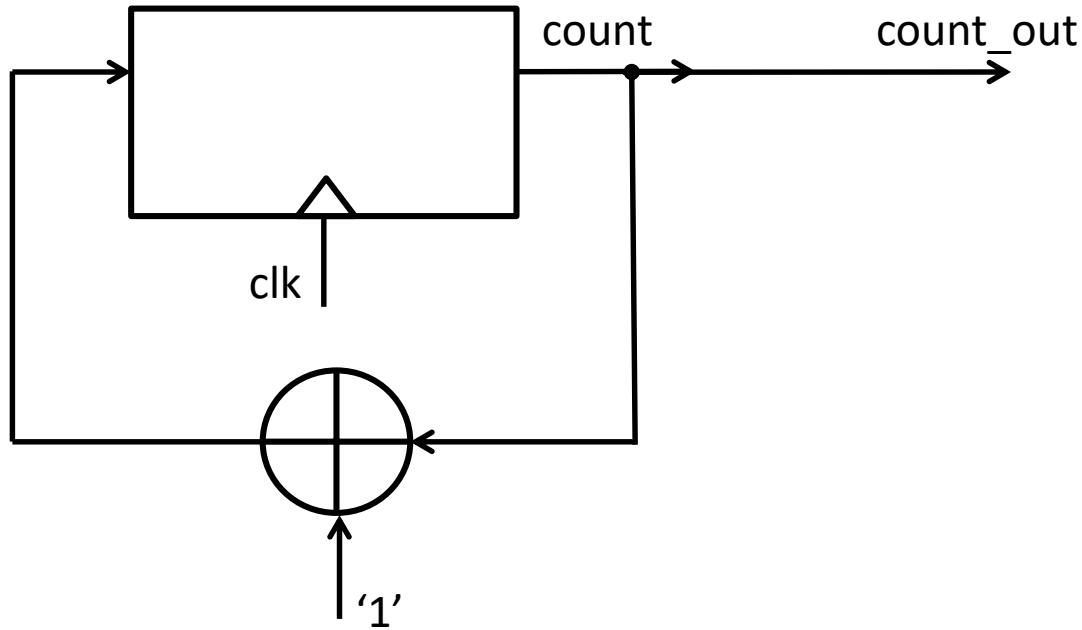
# Example 2, Counter – One-process Method

```
entity up_counter is
     port(clk : in std_logic;
          count_out : out std_logic_vector(7 downto 0));
end up_counter;

architecture behavior of up_counter is
     signal count : std_logic_vector(7 downto 0);
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               count <= count + 1;
          end if;
     end process;

     count_out <= count;

end behavior;
```



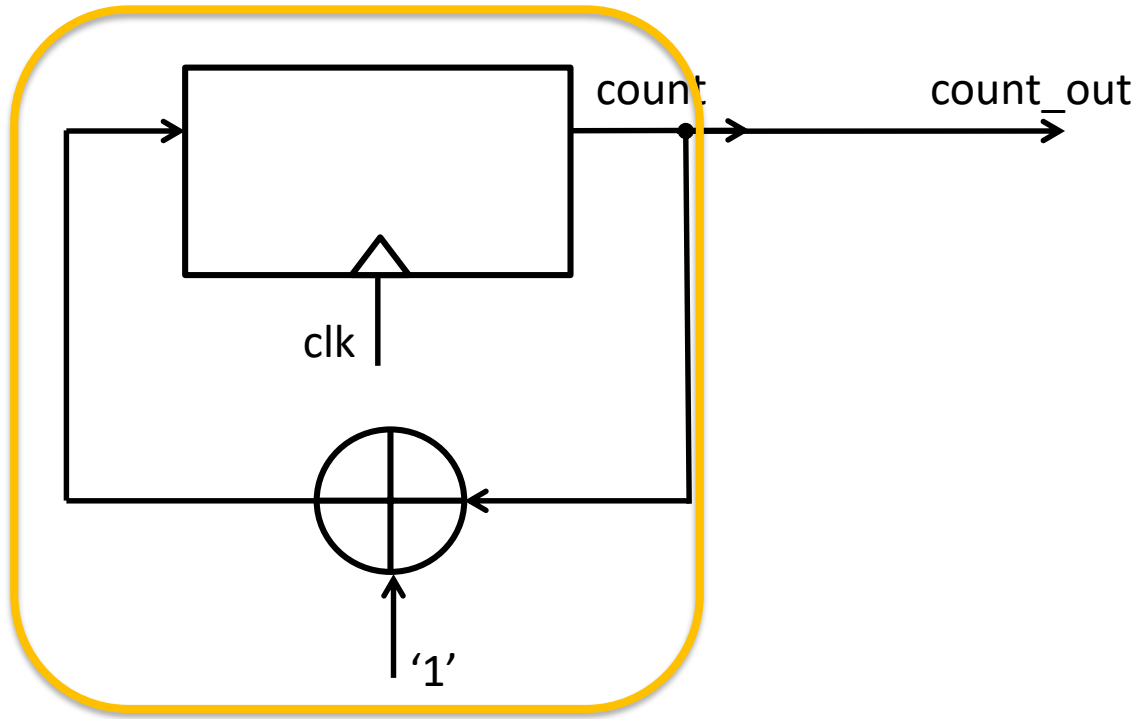
# Example 2, Counter – One-process Method

```
entity up_counter is
     port(clk : in std_logic;
          count_out : out std_logic_vector(7 downto 0));
end up_counter;

architecture behavior of up_counter is
     signal count : std_logic_vector(7 downto 0);
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               count <= count + 1;
          end if;
     end process;

     count_out <= count;

end behavior;
```




Combinational & sequential are mixed

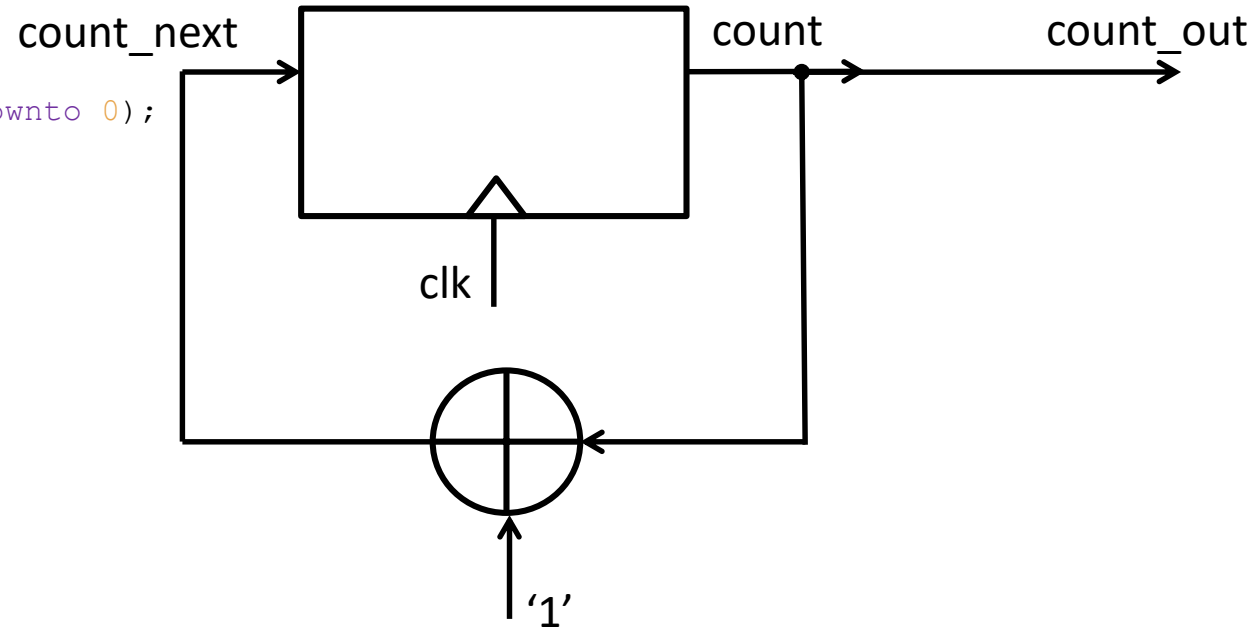
# Example 2, Counter – Two-process Method

```
entity up_counter is
     port(clk : in std_logic;
          count_out : out std_logic_vector(7 downto 0));
end up_counter;

architecture behavior of up_counter is
     signal count_next, count : std_logic_vector(7 downto 0);
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               count <= count_next;
          end if;
     end process;

     process(count)
     begin
          count_next <= count + 1;
     end process;

     count_out <= count;
end behavior;
```



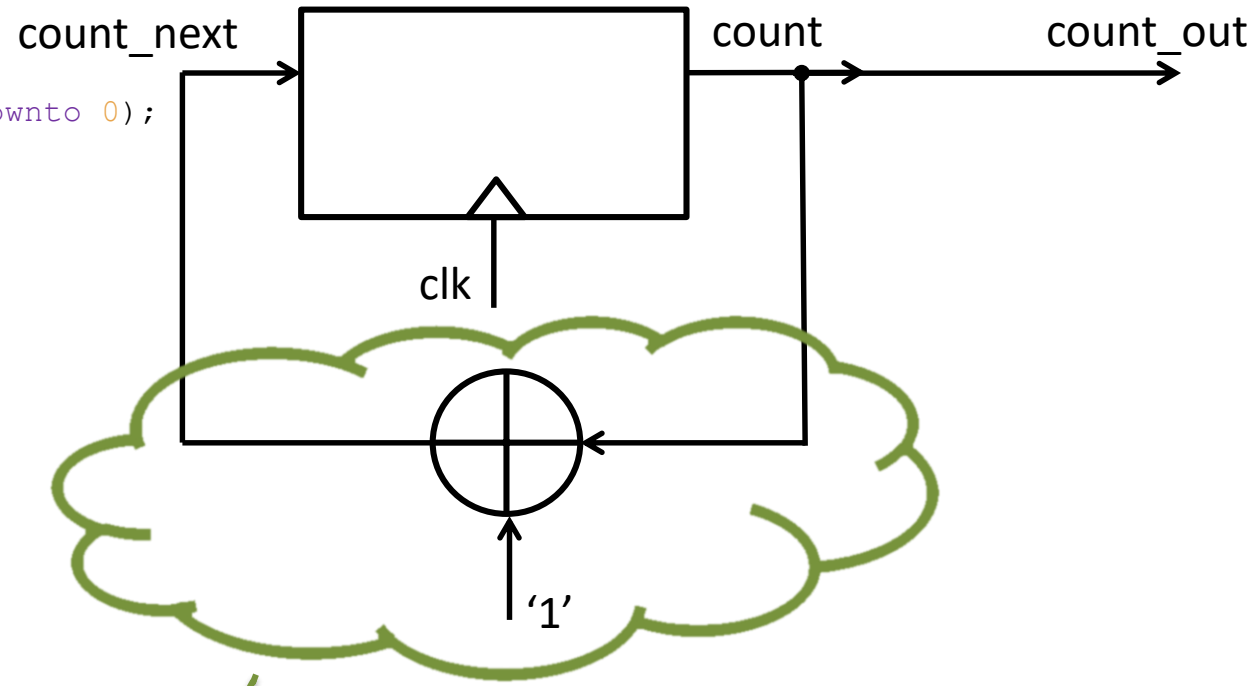
# Example 2, Counter – Two-process Method

```
entity up_counter is
     port(clk : in std_logic;
          count_out : out std_logic_vector(7 downto 0));
end up_counter;

architecture behavior of up_counter is
     signal count_next, count : std_logic_vector(7 downto 0);
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               count <= count_next;
          end if;
     end process;

     process(count)
     begin
          count_next <= count + 1;
     end process;

     count_out <= count;
end behavior;
```




Combinational

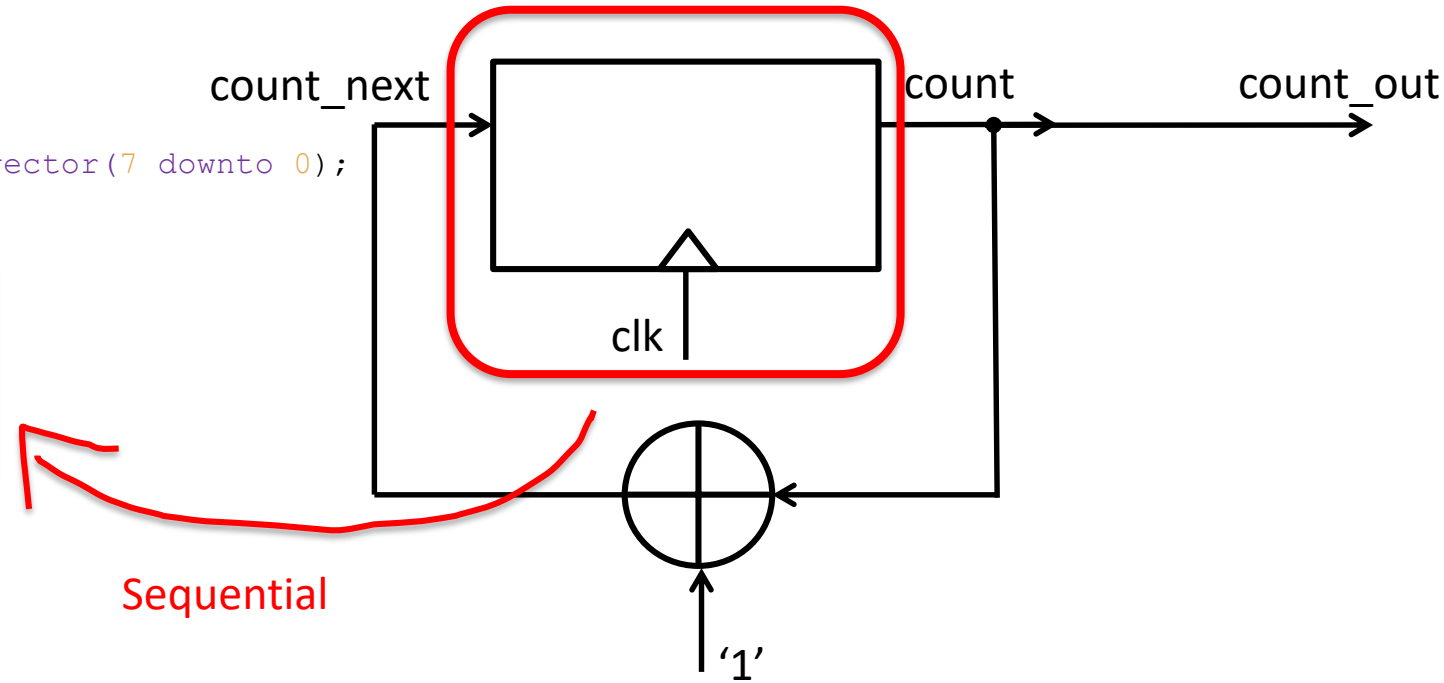
# Example 2, Counter – Two-process Method

```
entity up_counter is
     port(clk : in std_logic;
          count_out : out std_logic_vector(7 downto 0));
end up_counter;

architecture behavior of up_counter is
     signal count_next, count : std_logic_vector(7 downto 0);
begin
     process(clk)
     begin
          if clk'event and clk = '1' then
               count <= count_next;
          end if;
     end process;

     process(count)
     begin
          count_next <= count + 1;
     end process;

     count_out <= count;
end behavior;
```

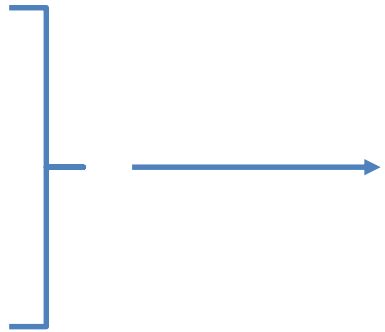


# Assessment

- One-process style
  - Fewer lines of code
  - Less descriptive
- Two-process style
  - More verbose
  - Less susceptible to errors
  - Better modularity

# Assessment

- One-process style
  - Fewer lines of code
  - Less descriptive
- Two-process style
  - More verbose 
  - Less susceptible to errors 
  - Better modularity 



You must use this one in the course