CPE 462
VHDL: Simulation and Synthesis

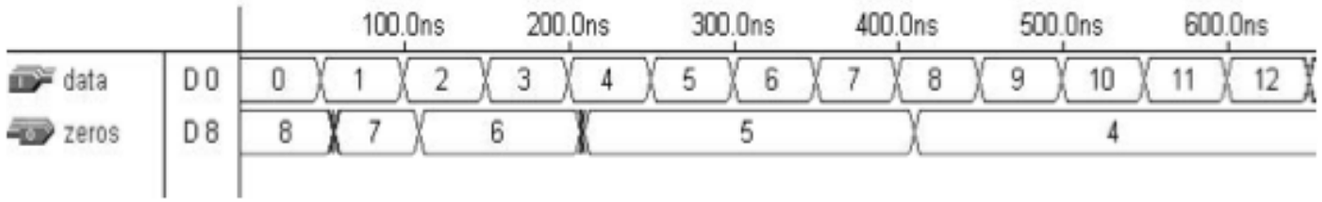
Topic #06 - d) Final topics in sequential code

Loop example

- This design counts the number of leading zeros in a binary vector, starting from the left end.
- EXIT implies not a escape from the current iteration of the loop, but rather a definite exit from it (that is, even if i is still within the specified range, the LOOP statement will be considered as concluded).
- In this example, the loop will end as soon as a '1' is found in the data vector.
- Therefore, it is appropriate for counting the number of zeros that precedes the first one.

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY LeadingZeros IS
6     PORT ( data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            zeros: OUT INTEGER RANGE 0 TO 8);
8  END LeadingZeros;
9  ----------------------------------------
10 ARCHITECTURE behavior OF LeadingZeros IS
11 BEGIN
12    PROCESS (data)
13       VARIABLE count: INTEGER RANGE 0 TO 8;
14    BEGIN
15       count := 0;
16       FOR i IN data'RANGE LOOP
17          CASE data(i) IS
18             WHEN '0' => count := count + 1;
19             WHEN OTHERS => EXIT;
20          END CASE;
21       END LOOP;
22       zeros <= count;
23    END PROCESS;
24 END behavior;
25 ----------------------------------------
```

Loop example: simulation



- With data="00000000" (decimal 0), eight zeros are detected; when data="00000001" (decimal 1), seven zeros are encountered; etc

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY LeadingZeros IS
6     PORT ( data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            zeros: OUT INTEGER RANGE 0 TO 8);
8  END LeadingZeros;
9  ----------------------------------------
10 ARCHITECTURE behavior OF LeadingZeros IS
11 BEGIN
12    PROCESS (data)
13       VARIABLE count: INTEGER RANGE 0 TO 8;
14    BEGIN
15       count := 0;
16       FOR i IN data'RANGE LOOP
17          CASE data(i) IS
18             WHEN '0' => count := count + 1;
19             WHEN OTHERS => EXIT;
20          END CASE;
21       END LOOP;
22       zeros <= count;
23    END PROCESS;
24 END behavior;
25 ----------------------------------------
```

Bad Clocking

- The compiler will generally not be able to synthesize codes that contain assignments to the same signal at both transitions of the reference (clock) signal (that is, at the rising edge plus at the falling edge).
- This is particularly true when the target technology contains only single-edge flip-flops.
- As an example, let us consider the case of a counter that must be incremented at every clock transition (rising plus falling edge).

Bad Clocking Example #1

- The compiler will generally not be able to synthesize codes that contain assignments to the same signal at both transitions of the reference (clock) signal (that is, at the rising edge plus at the falling edge).

```
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      counter <= counter + 1;
   ELSIF(clk'EVENT AND clk='0') THEN
      counter <= counter + 1;
   END IF;
   ...
END PROCESS;
```

Bad Clocking Example #2

- The compiler does not know if we are dealing with rising or falling edge.
  - It may issue a message of the type “clock not locally stable”
  - ... Or it may assume **clk’event=’0’**.
  - Don’t do this!

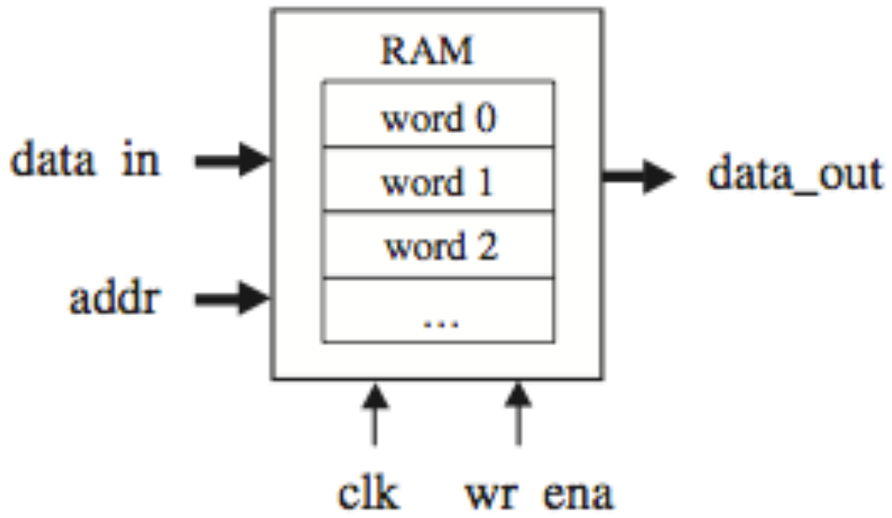
```
PROCESS (clk)
BEGIN
   IF(clk'EVENT) THEN
      counter := counter + 1;
   END IF;
   ...
END PROCESS;
```

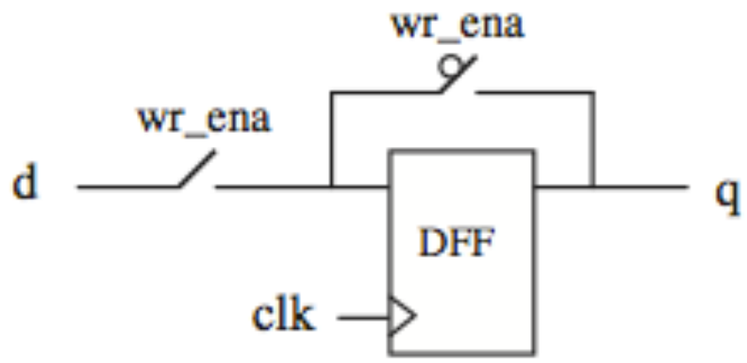
Good Clocking Example #1

- If I want to perform an operation on both clk edges I need to declare two different processes

```
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      x <= d;
   END IF;
END PROCESS;
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='0') THEN
      y <= d;
   END IF;
END PROCESS;
----------------------
```

Good Clocking Example #2: RAM

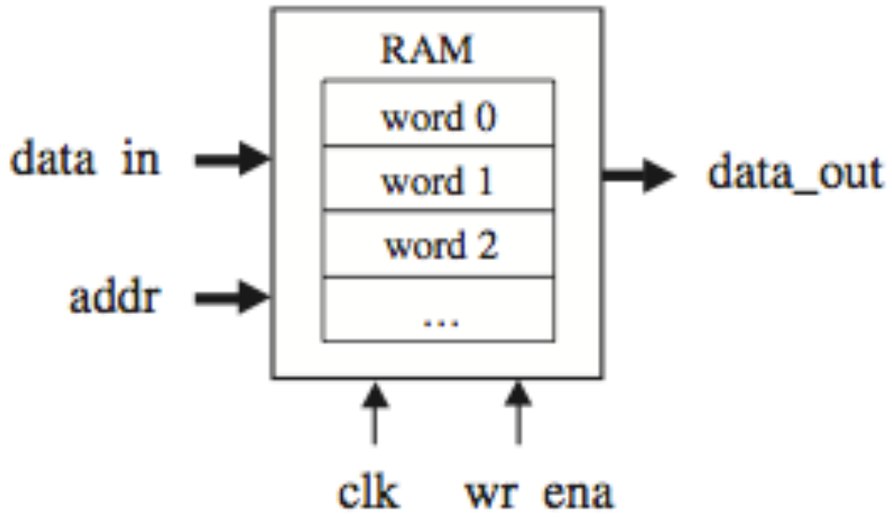


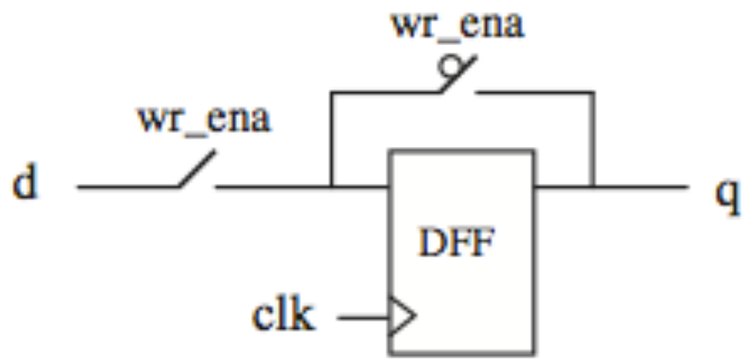


- The circuit has a data input bus (data_in), a data output bus (data_out), an address bus (addr), plus clock (clk) and write enable(wr_ena) pins.

- When **wr_ena** is asserted, at the next rising edge of **clk** the vector present at **data_in** must be stored in the position specified by **addr**. The output, **data_out**, on the other hand, must constantly display the data selected by **addr**.

Good Clocking Example #2: RAM





- When **wr_ena** is low, **q** is connected to the input of the flip-flop, and terminal **d** is open, so no new data will be written into the memory.
- However, when **wr_ena** is turned high, **d** is connected to the input of the register, so at the next rising edge of **clk**, **d** will overwrite its previous value.

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY ram IS
6  GENERIC ( bits: INTEGER := 8;      -- # of bits per word
7            words: INTEGER := 16);  -- # of words in the memory
8     PORT ( wr_ena, clk: IN STD_LOGIC;
9            addr: IN INTEGER RANGE 0 TO words-1;
10           data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
11           data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
12 END ram;
13 ------------------------------------------------
14 ARCHITECTURE ram OF ram IS
15    TYPE vector_array IS ARRAY (0 TO words-1) OF
16       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
17    SIGNAL memory: vector_array;
18 BEGIN
19    PROCESS (clk, wr_ena)
20    BEGIN
21       IF (wr_ena='1') THEN
22          IF (clk'EVENT AND clk='1') THEN
23             memory(addr) <= data_in;
24          END IF;
25       END IF;
26    END PROCESS;
27    data_out <= memory(addr);
28 END ram;
```

- The capacity chosen for the RAM is 16 words of length 8 bits each.
- Notice that the code is totally generic.