FYS4220/9220

UNIVERSITY OF OSLO

VHDL – combinational and synchronous logic

Lecture #3

Reading: 2.5, chapter 4, 5.1 and chapter 6 in Zwolinski

J. K. Bekkeng, 2.07.2011

Combinational vs Sequential logic

- In **combinational logic** the output is only dependent on the present input.
- In **sequential logic** the output is dependent on both the present input and the state (memory, based on earlier inputs).
- Therefore, sequential logic has memory, while combinational logic does not.

Comparator - Behavioral (I) Style

UNIVERSITY OF OSLO

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity eqcomp4 is port(
4         a, b:           in std_logic_vector(3 downto 0);
5         equals:         out std_logic);
6 end eqcomp4;
7
8 architecture behavioral of eqcomp4 is
9 begin
10 comp: process (a, b)
11         begin
12                 if a = b then
13                         equals <= '1';
14                 else
15                         equals <= '0';
16                 end if;
17         end process comp;
18 end behavioral;
```

Sensitivity list

Sequential statements

Comparator - Behavioral (II)

UNIVERSITY OF OSLO

```
1 architecture behavioral of eqcomp4 is
2 begin
3 comp: process (a, b)
4         begin
5                 equals <= '0';      -- Default value
6                 if a = b then
7                         equals <= '1';
8                 end if;
9         end process comp;
10 end behavioral;
```

Note: Signals are set when the process terminates

- **The order of the statements is important!**
- **Only the last assignment of a signal has any effect!**

”Process”

UNIVERSITY OF OSLO

- The process is executed when one of the signals in the sensitivity list has a change (an event)
- Then, the sequential signal assignments are executed
- The process continue to the last signal assignment, and terminates
- **The signals are updated just before the process terminates!**
- The process is not executed again before one of the signals in the sensitivity list has a new event (change)

```
proc1: process (a, b, c)
   begin
      x <= a and b and c;
   end process;
```

```
process (<sens list>)
< declaration>
begin
    <signal assignment1>
    .
    .
    <signal assignment n>
end process;
```

```
clk: process is  -- without sensitivity list
begin
clock <= ’0’;
wait for 50 ns;
clock <= ’1’;
wait for 50 ns;  -- wait needed!
end process;
```

Combinational logic and ”process”

UNIVERSITY OF OSLO

- Remember to include all inputs in the sensitivity list!

```
proc4: process (a, b, c)
    begin
        x <= a and b and c;
    end process;
```

Comparator - Dataflow (I)

UNIVERSITY OF OSLO

- Does not use process!

```
1 -- eqcomp4 is a four bit equality comparator
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity eqcomp4 is port(
5         a, b:           in std_logic_vector(3 downto 0);
6         equals:         out std_logic);
7 end eqcomp4;
8
9 architecture dataflow of eqcomp4 is
8 begin
9           equals <= '1' when (a = b) else '0';
10 end dataflow
```

when - else

Combinational logic

UNIVERSITY OF OSLO

- Does not have memory (only dependent on present input)
- To avoid unwanted memory:
  - Include ***else*** in *if then else*
  - Include ***when others*** in *case*
  - and/or use ***"default"*** values

Gives a circuit with memory:

```
process (ctrl, A)
begin
  if ctrl = '1' then
    Z <= 'A';
  end if;
end process;
```

Gives a combinational circuit:

```
process (ctrl, A)
begin
  if ctrl = '1' then
    Z <= 'A';
  else
    Z <= '0';
  end if;
end process;
```

UNIVERSITY OF OSLO

More about unwanted memory

```
process (sel, A, B)
begin
  case sel is
    when "00" => Y <= A;
    when "10" => Y <= B;
end process;
```

Gives memory, because "**when others**" or "**default**" values are missing

```
process (sel, A, B)
begin
  case sel is
    when "00"   => Y <= A;
    when "10"   => Y <= B;
    when others => Y <= '0';
end process;
```

```
process (sel, A, B)
begin
  Y <= '0';
  case sel is
    when "00"   => Y <= A;
    when "10"   => Y <= B;
end process;
```

← Default value

Equivalent descriptions, which gives combinational logic

Data objects

UNIVERSITY OF OSLO

- Constants: increase readability
  - **constant** width: integer := 8;
- Signals – a signal line or a memory element
  - **signal** count: std_logic_vector (3 **downto** 0);
- Variables – synthesis of variables is not well defined
  - **variable** result: std_logic := '0';
- Aliases – not a new object

```
signal address: std_logic_vector(31 downto 0);
alias top_ad: std_logic_vector(3 downto 0) is address(31 downto 28);
alias bank: std_logic_vector(3 downto 0) is address(27 downto 24);
alias row_ad: std_logic_vector(11 downto 0) is address(23 downto 12);
```

Signals and variables

UNIVERSITY OF OSLO

Signals:

- Signal assignment <=
- Defined in architecture (before ***begin***)
- Signals are updated just before the process terminates!
- Use signals instead of variables when possible!

Variable:

- Variable assignment : =
- Variable assignment is instantaneous
- In synthesis they are used as index variables and temporal storage of data
- Can be used to simplify algorithms
- Can be used inside a process
- Must be defined inside a process

Signals

signal count: std_logic_vector(3 downto 0);

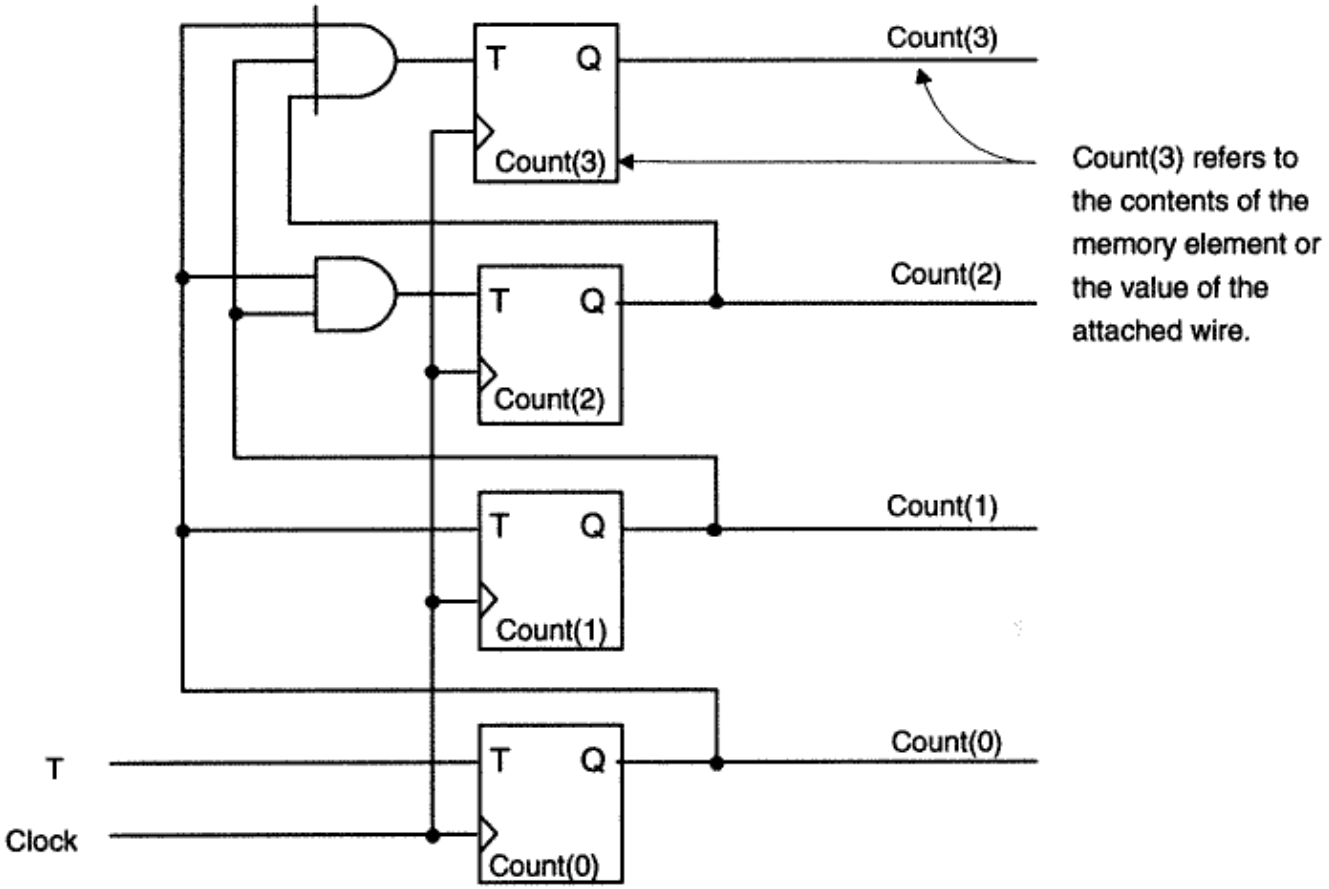
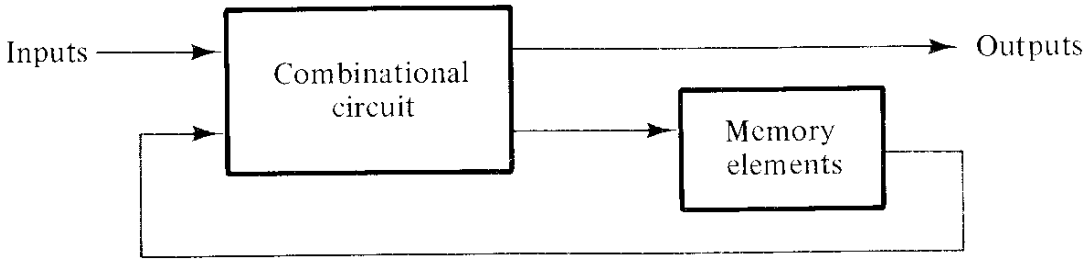


Figure 3-17 A signal can refer to the memory elements or the wires attached to the outputs of the memory elements

Sequential and synchronous logic

- Most digital systems have memory elements (e.g. flip-flops) in addition to combinational logic, and is then called **sequential logic**
- The output in a sequential circuit is dependent on both present input and present state (of a memory element)



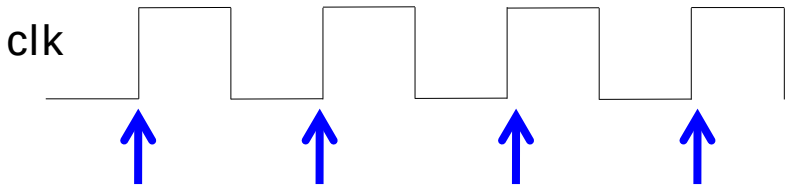
- Synchronous logic use a clock such that the memory elements are updated only at specific times (at the rising/falling clock edge)

rising_edge og falling_edge

- Defined by the package **std_logic_1164**
- The signal must be of the type **std_logic** in order to use these two functions
- Detects rising/falling edge on the signal

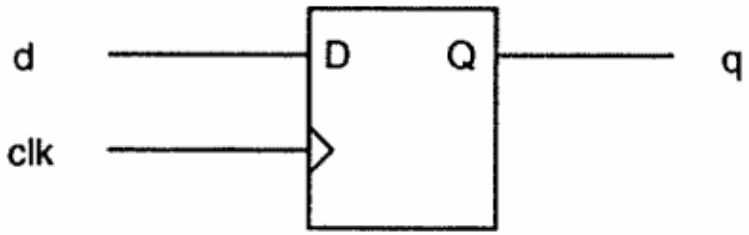
Only one single test on rising/falling clock edge for each process!

```
FSM : process(clk) IS
begin

if (rising_edge(clk1MHz)) then
    Datain    <= Dout;
    ncs    <= nCS_control;
end if;

if(falling_edge(clk1MHz)) then

    Case present_state is
```



```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
        d, clk : in std_logic;
        q      : out std_logic);
end dff_logic;

architecture example of dff_logic is
begin
        process (clk) begin
                if (clk'event and clk = '1') then
                        q <= d;
                end if;
        end process;
end example;
```

D-flip-flop



no **else**; gives implicit memory

```
if (clk'event and clk = '1') then
if rising_edge(clk)  then
```

-- rising edge

```
if (clk'event and clk = '0') then
if falling_edge(clk)  then
```

-- falling edge

Example: 3-bit counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- for + operator
use ieee.std_logic_unsigned.all
.

signal bit_cnt :  std_logic_vector(2 downto 0);

.

BITCOUNTER:
process (sclk, reset_sync)
begin
  if (reset_sync = '1') then
            bit_cnt <= (others  => '0');
  elsif falling_edge(sclk) then
            bit_cnt <= bit_cnt + 1;
  end if;
end process;
```

Operator overloading & important functions I

UNIVERSITY OF OSLO

- To add a constant to a signal of type **std_logic**, an overloaded operator is required (in addition to the native VHDL operators)
  - Solution: Add the package **std_logic_arith**
- The expression ***if a = " 1--1"*** is always evaluated to false in native VHDL, except for ***" 1--1"***
  - Solution*:* include the package **std_logic_arith**, and use the function **std_match**: ***if std_match(a, " 1--1")***
- Overloading of the = operator
  - The expression **a = "00001**" only true if array sizes are equal in native VHDL
  - Solution: Include a package that overloads the = operator, e.g. the **numeric_std** package

Operator overloading & important functions II

UNIVERSITY OF OSLO

- The following packages solve most of these problems:

  library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_unsigned.all;

- Some other packages:

  math_real
  math_complex
  std_logic_textio

Clock divider (using variable)

UNIVERSITY OF OSLO

- Clock division different from 2^n
- Wants ~ 50 % clock duty-cycle

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity clock12div is
  port
    (
    clk_in  : in  std_logic;
    reset   : in  std_logic;
    clk_out : out std_logic
    );
end clock12div;

architecture clock_div_arch of clock12div is

constant DivFactor : integer := 12;
constant DivFactor_half : integer := 6;

begin
```

```
CLK_DIV: process (clk_in, reset)

variable div_cnt : integer range 0 to DivFactor - 1;

begin
  if reset = '1' then              -- asynchronous reset
    div_cnt := 0;
    clk_out<= '0';
  elsif rising_edge(clk_in) then
    if (div_cnt = DivFactor - 1) then
       div_cnt := 0;                -- reset the counter
    else
     div_cnt := div_cnt + 1;     -- increment the counter
    end if;
    if (div_cnt >= DivFactor_half) then
     clk_out <= '0';
    else
     clk_out <= '1';
    end if;
  end if;
end process CLK_DIV;
end clock_div_arch;
```

Some Data types

UNIVERSITY OF OSLO

- Enumeration – important for state machines
  - **type state is (idle, preamble, data, error);**
- integer
  - **variable a: integer range 0 to 255**

- Physical - time is only predefined type, not used in sythesis
  - ns, us, ms
- Floating - usually not supported directly in programmable logic

- Integers : 011_2 represents 3_{10}
- Fixed-point numbers : 110.01_2 represents 6.25_{10}
  $(2^2 + 2^1 . 2^{-2})$
  store $11001_2 = 25_{10}$ and divide by 2^2
- Floating-point numbers : $(-1)^{sign} * mantissa * 2^{exponent}$

Sequential statements (if-then-else)

UNIVERSITY OF OSLO

- Used in **process**, functions and procedures

```
if (condition) then
   do something;
else
   do something different;
end if;
```

Note: The order of the signal assignments affects the logic which is produced!

```
signal step: std_logic;
signal addr: std_logic_vector(7 downto 0);
      .
      .
      .

similar1: process (addr)
   begin
      step <= '0';
      if addr > x"0F" then
         step <= '1';
      end if;
   end process;
```

Functional identical processes

```
similar2: process (addr)
   begin
      if addr > x"0F" then
         step <= '1';
      else
         step <= '0';
      end if;
   end process similar2;
```

Sequential vs concurrent statements

UNIVERSITY OF OSLO

Sequential statements

process(.....)

```
if (condition) then
   do something;
else
   do something different;
end if;
```

process(.....)

```
if (condition) then
   do something;
else
   do something different;
end if;
```

- The order of the sequential statements (in the process) is important!
- If there are multiple processes they are all executed in parallel and concurrent with other ”concurrent statements” in the architecture!

Concurrent statements

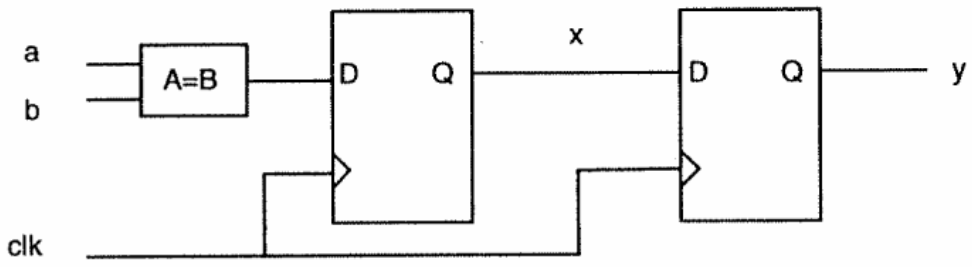
```
aeqb <= '1' when (a = b) else '0';
ceqd <= '1' when (c = d) else '0';
```

- ”*concurrent statements*” are used outside ”*process*”
- Executed concurrently (samtidig)
- The order of ”concurrent statements” is arbitrary

Important about seq. statem.

UNIVERSITY OF OSLO

```
architecture careful of dangerous is
   signal x: bit;
begin
p1: process begin
   wait until clk = '1';
      x <= '0';
      y <= '0';
      if a = b then
         x <= '1';
      end if;

      if x = '1' then
         y <= '1';
      end if;
   end process p1;
   end careful;
```



***x* is <u>not</u> assigned the new value here! The comparison is with the value x got the last time the process was executed!**

Listing 4-37 A signal in an assignment and as an operand within the same process

with-select-when : Multiplexer (I)

UNIVERSITY OF OSLO

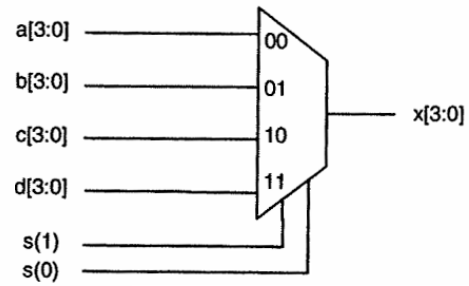
```
with selection_signal select
   signal_name <=  value_a when value_1_of_selection_signal,
                   value_b when value_2_of_selection_signal,
                   value_c when value_3_of_selection_signal, ...
                   value_x when last_value_of_selection_signal;
```

All values of *selection signal* must be listed using ***when***, and they must be "mutually exclusive". This demands use of ***when others***

Example – 4 to 1 multiplexer

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
        a, b, c, d:     in std_logic_vector(3 downto 0);
        s:              in std_logic_vector(1 downto 0);
        x:              out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
with s select
    x <= a when "00",
         b when "01",
         c when "10",
         d when others;
end archmux;
```

s is of the type std_logic which has 9 possible values. This gives 81 possibilities for simulation (for synthesis "11" is the only additional value)



when-else : Multiplexer (II)

UNIVERSITY OF OSLO

```
signal_name <=  value_a when condition1 else
                value_b when condition2 else
                value_c when condition3 else ...
                value_x;
```

signal_name is assigned to the first condition which is true (inherent priority)

Example – 4 to 1 multiplexer

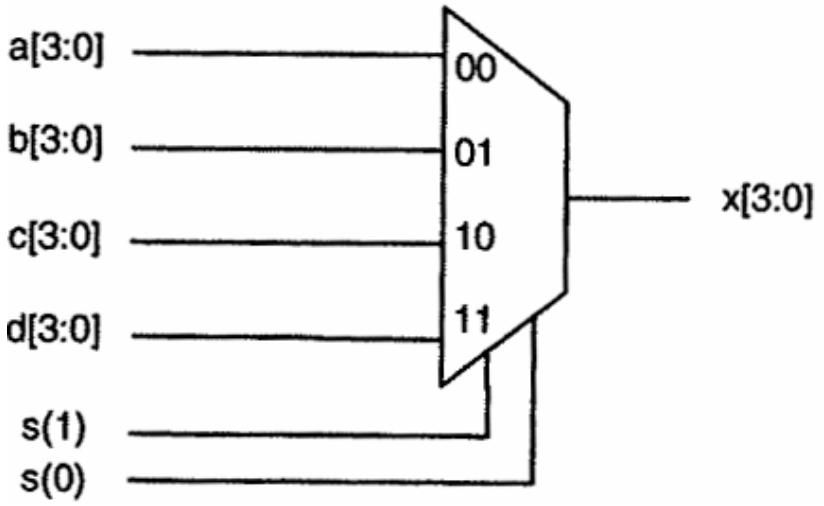
```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
        a, b, c, d:     in std_logic_vector(3 downto 0);
        s:              in std_logic_vector(1 downto 0);
        x:              out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
        x <= a when (s = "00") else
             b when (s = "01") else
             c when (s = "10") else
             d;
end archmux;
```

case-when : Multiplexer (III)

UNIVERSITY OF OSLO

```
architecture mux_arch of mux is
begin


process(a, b ,c, d, s)
begin
  case s is
    when "00“    =>  x<=a;
    when "01“    =>  x<=b;
    when “10“    =>  x<=c;
    when others=>  x<=d;
  end case;
end process;
end  mux_arch;
```

Combinational logic demands that all input signals must be included in the sensitivity list of the process!



if-else : Multiplexer (IV)

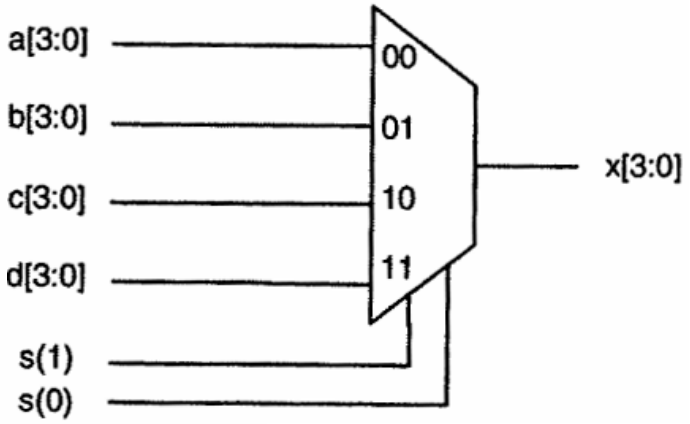
UNIVERSITY OF OSLO

```
if (condition1) then
   do something;
elsif (condition2) then
   do something different;
else
   do something completely different;
end if;
```

Example : 4-1 multiplexer

Combinational logic demands that all input signals must be included in the sensitivity list of the process!

```
architecture archmux of mux is
begin
mux4_1: process (a, b, c, d, s)
        begin
                if s = "00" then
                        x <= a;
                elsif s = "01" then
                        x <= b;
                elsif s = "10" then
                        x <= c;
                else
                        x <= d;
                end if;
        end process mux4_1;
end archmux;
```



Listing 4-12 Describing a multiplexer with an `if-then` statement

Loops

UNIVERSITY OF OSLO

for loop

```
for i in 7 downto 0 loop
        fifo(i) <= (others => '0');
end loop;
```

Loop variable *i* automatically declared in a *for loop*

while loop

```
reg_array: process (rst, clk)
        variable i: integer :=0;
 begin
     if rst = '1' then
         while i < 7 loop
             fifo(i) <= (others => '0');
              i := i + 1;
         end loop;
```

Declaration and initializing of the loop variable *i*

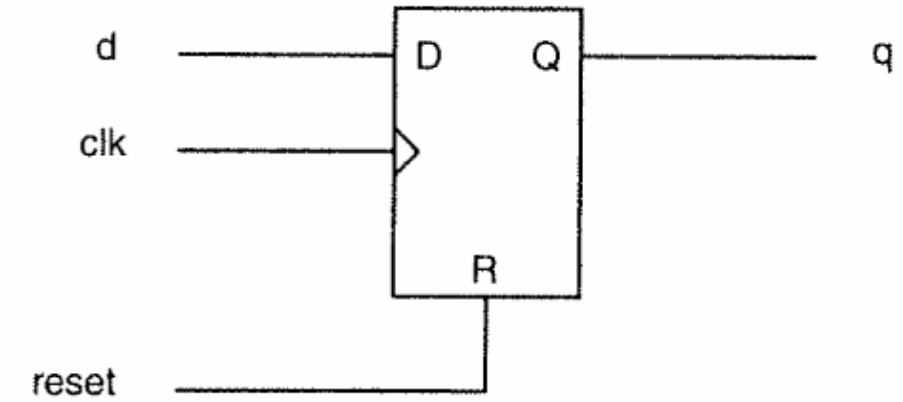
Increments the loop variable *i*

Synchronous reset

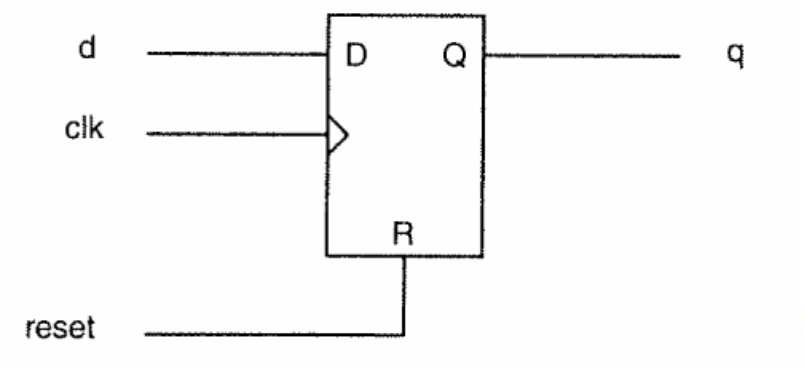
```
architecture sync_rexample of dff_logic is
begin
        process (clk) begin
                if rising_edge(clk) then
                        if (reset = '1') then
                                q <= '0';
                        else
                                q <= d;
                        end if;
                end if;
        end process;
end sync_rexample;
```

Synchronous logic

reset located inside the part of the process which is synchronous to the clock



Reset in synchronous logic; **asynchronous reset**




```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
        d, clk, reset: in std_logic;
        q              : out std_logic);
end dff_logic;

architecture rexample of dff_logic is
begin
        process (clk, reset) begin
                if reset = '1' then
                        q <= '0';
                elsif rising_edge(clk) then
                        q <= d;
                end if;
        end process;
end rexample;
```

reset and **preset** are used to set the logic in a known state

For preset function:

```
if preset = '1' then
        q <= '1'
```

Asynchronous Reset and Preset, synchronous Load

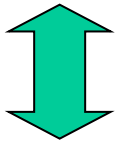
UNIVERSITY OF OSLO

Example : 8 bit counter

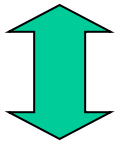
```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity cnt8 is port(
        txclk, grst, gpst:      in std_logic;
        enable, load:           in std_logic;
        data:                   in unsigned(7 downto 0);
        cnt:                    buffer unsigned(7 downto 0));
end cnt8;

architecture archcnt8 of cnt8 is
begin
count: process (grst, gpst, txclk)
  begin
        if grst = '1' then
                cnt <= (others => '0');
        elsif gpst = '1' then
                cnt <= (others => '1');
        elsif (txclk'event and txclk='1') then
                if load = '1' then
                        cnt <= data;
                elsif enable = '1' then
                        cnt <= cnt + 1;
                end if;
        end if;
  end process count;
end archcnt8;
```

Listing 4-28 A counter with asynchronous reset and preset

cnt <= (others => ’0’)



cnt <= ”00000000”;



for i in 0 to 7 loop
cnt(i) <= ‘0’;
end loop;

Avoid latches

- Latches are created by "if" statements which are not completely specified.
- A Latch is created when an "else" statement is omitted, when values are not assigned a value, or when the "event" statement is missing.
- To avoid a Latch being developed assign an output for all possible input conditions.
  - Use an "else" statement instead of an "elsif" statement in the final branch of an "if" statement to avoid a latch.
  - Be sure to assign default values for all outputs at the beginning of a process.

```
-- VHDL Latch example
    process (enable, data_in)
    begin
            if enable = '1' then
            q <= data_in;
            end if;
    end process;
```

```
-- VHDL D flip-flop  example
process (clk) begin
            if (clk'event and clk = '1') then
                        q <= d;
            end if;
end process;
end example;
```

Coding for Synthesis

- Omit the **wait for XX ns** statement
- Omit the **... after XX ns** statement
- Omit initial values
  - Do not assign signals and variables initial values because initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design. For example, do not use initialization statements like the following: **variable SUM:INTEGER:=0;**
- Make sure that all outputs are defined in all branches of an *if statement*. If not it can creates latches
  - A good way to prevent this is to have default values for all outputs before the ***if*** *statements.*

```
similar1: process (addr)
   begin
      step <= '0';
      if addr > x"0F" then
         step <= '1';
      end if;
   end process;
```