Hardware Programming HWP01

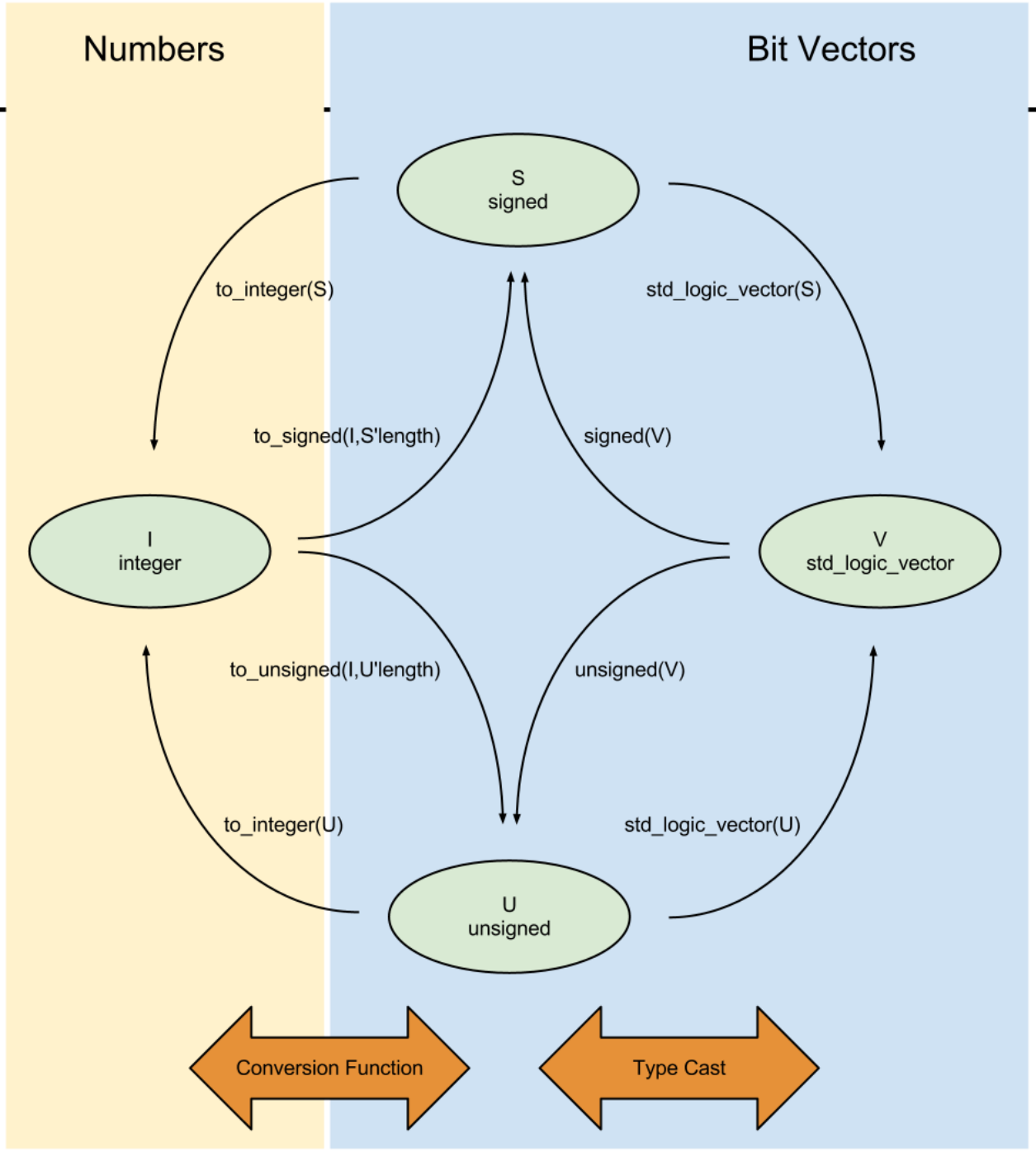
capturing a
FPGA
design with
VHDL

Planning: theory

- First week
  - Introduction digital systems
  - Structured digital Design
  - RTL
- Second week
  - Introduction VHDL
  - Code structure and data types
  - Design verification
- **Third week**
  - Combinational versus sequential design
  - Concurrent and sequential code
  - Signals and variables
- Fourth week
  - Introduction to state machines
- Fifth week
  - Designing state machines
  - Advanced VHDL design

Agenda

- **Discussion of previous week**
- Combinational versus sequential design
- Concurrent and sequential code
- Signals versus variables



Zie ook:
TABEL:
H.3 ; blz 76

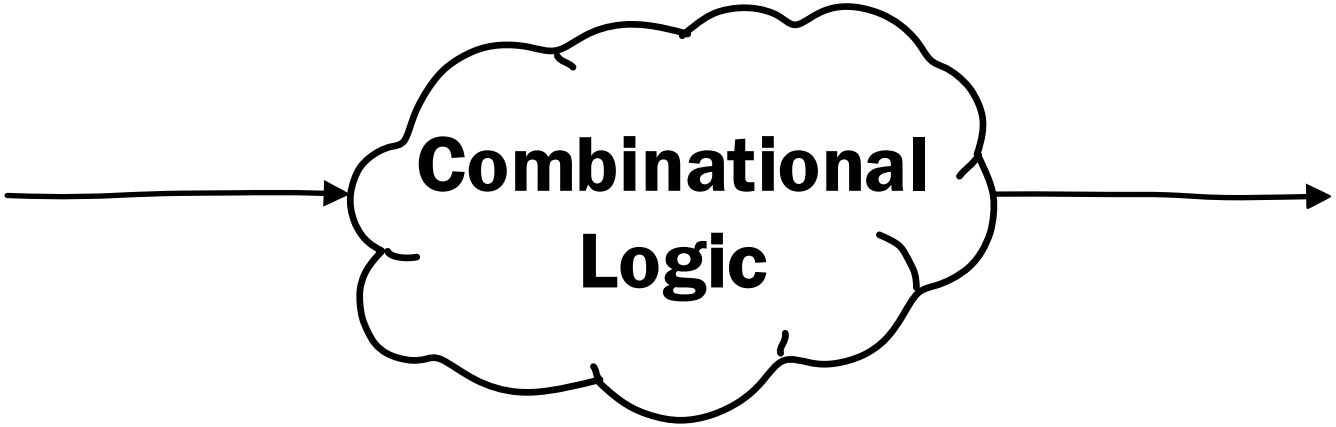
HOGESCHOOL ROTTERDAM

Agenda

- Discussion of previous week
- **Combinational versus sequential design**
- Concurrent and sequential code
- Signals versus variables

Combinational and sequential logic (1/2)

- Combinational: **no** memory

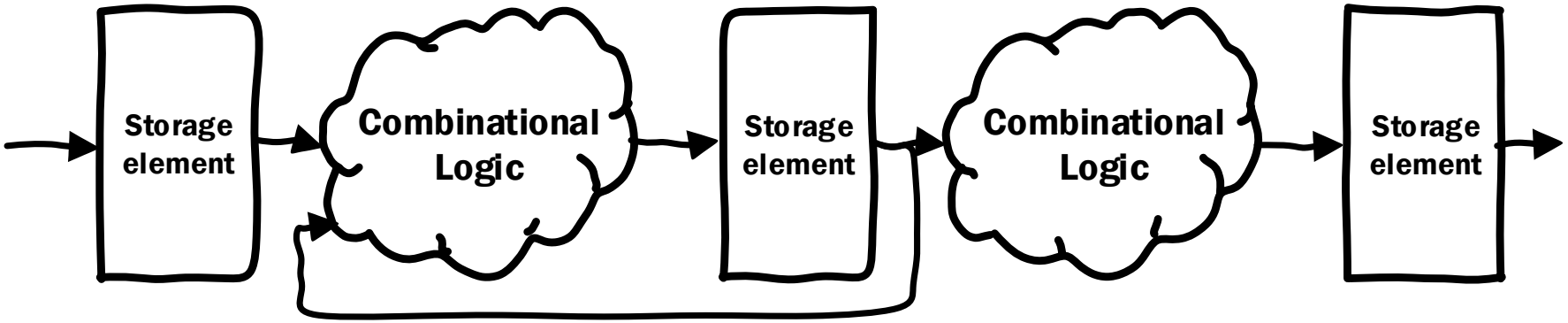


- The output is a function only of the current inputs.

- In any implementation there is a ***propagation delay***
  - For this course, we don't consider the propagation delay.

Combinational and sequential logic (2/2)

- Sequential logic:



- The output of sequential logic, is ...

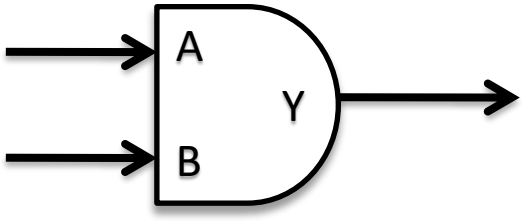
... a function of a *sequence* of operations ..

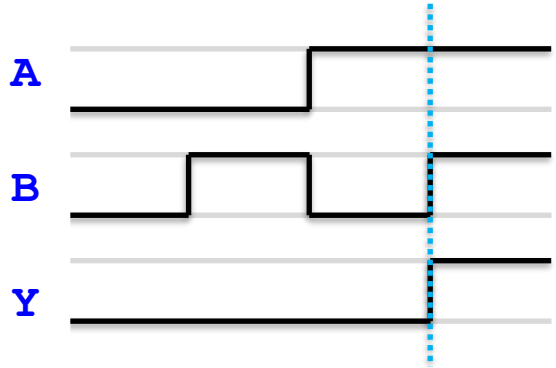
... on current and/or *previous inputs*.

- Advantages?

Combinational vs. sequential logic example

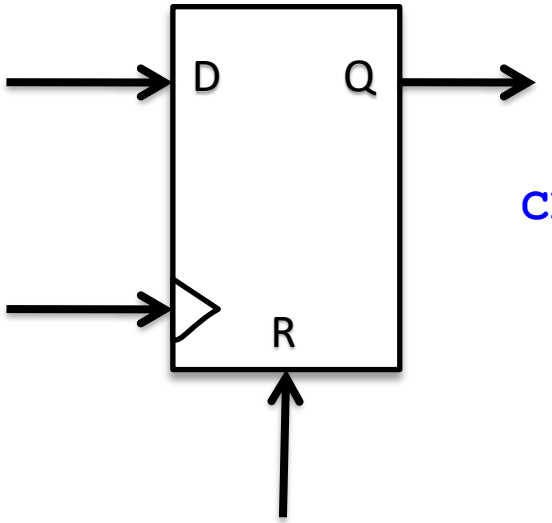
- Combinational

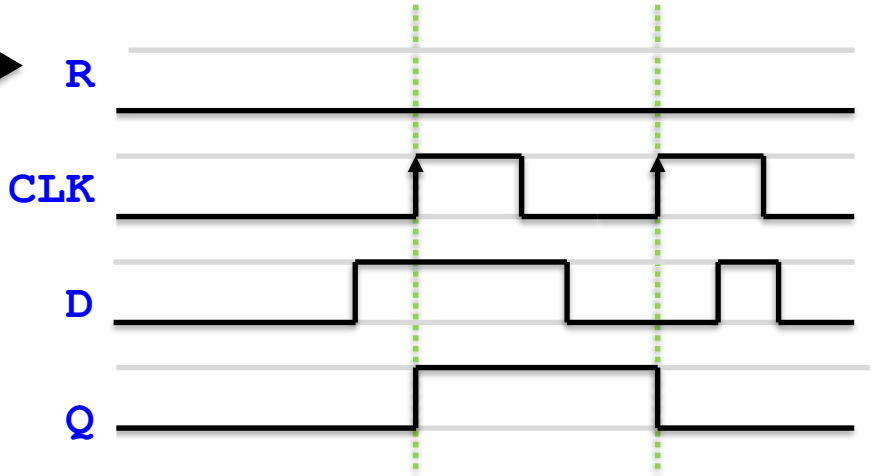




Updates instantly as input changes (in theory).

- Sequential





Updates only when CLK goes high. Until then it remembers the previous input (memory)

Combinational or sequential?

- Multiplexer vs. Synchronized Multiplexer?
- Timer?
- Encoder (for example binary to 7-seg display)?
- Comparators?
- Adder?
- Multiplier?
- ALU?
- CPU?
- Register?

HOGESCHOOL ROTTERDAM

Agenda

- Discussion of previous week
- Combinational versus sequential design
- **Concurrent and sequential code**
- Signals versus variables

HOGESCHOOL ROTTERDAM

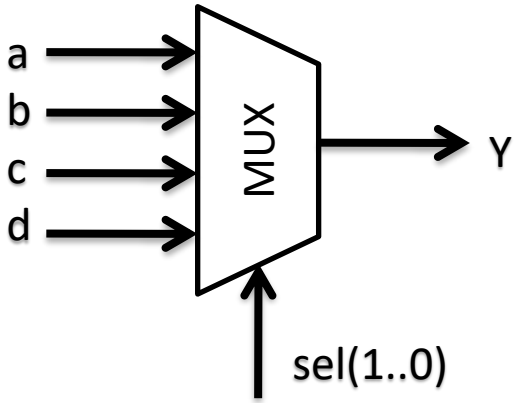
Concurrent Code

- concurrent code is intended **only for combinational circuits**.
  - often used for structural descriptions of a circuit.
- outputs activated asynchronously, at any time.
- statements for concurrent code:
  - **when ... else**
  - **with ... select**
  - **generate**
- can be placed outside **process**, **function** and **procedure**

HOGESCHOOL ROTTERDAM

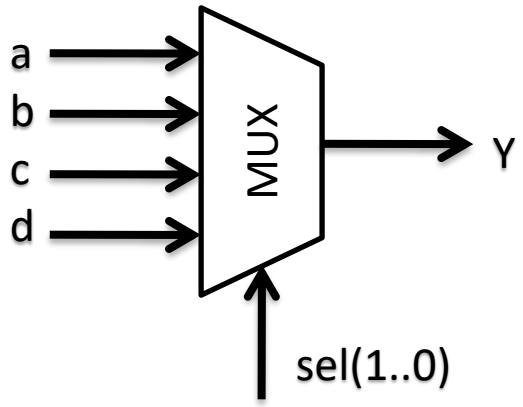
WITH/SELECT

- **with** ... **select** is used very often
- read: depending on sel, y becomes a when sel is 00, y becomes b when sel is 01, etc…
- note the usage of the **others** keyword here to cover *all* possibilities




```
architecture mux2 of mux is
begin
   with sel select
   y <= a when "00",    -- use "," not ";"
        b when "01",
        c when "10",
        d when others;
end mux2;
```

WHEN/ELSE

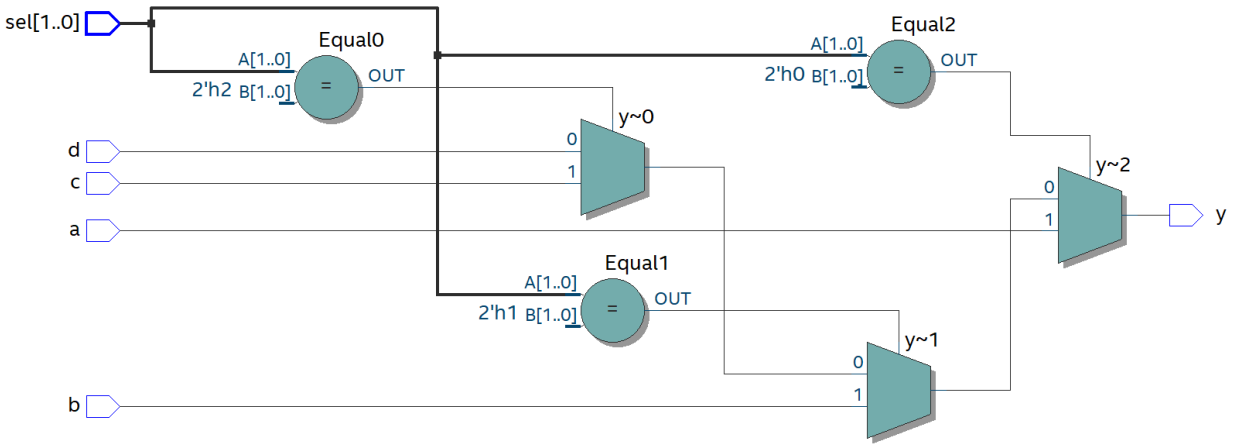



```
architecture mux1 of mux is
begin
    y <= a when sel="00" else
         b when sel="01" else
         c when sel="10" else
         d;
end mux1;
```

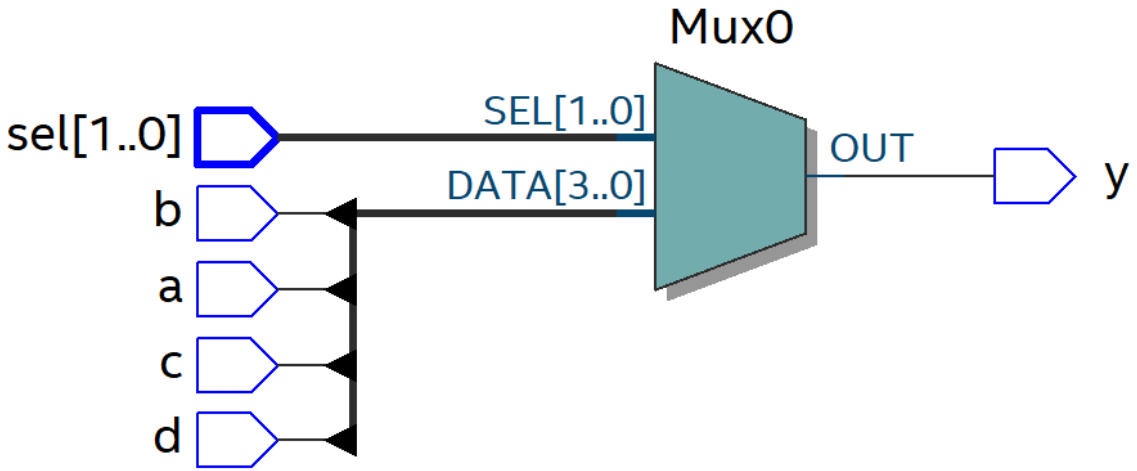
- Read: when sel is equal to “00”, y obtains the value of a, else when sel is equal to “01”, y obtains the value of b, etc…
- Cover all combinations
- Let the synthesizer do the K-map for us!
- Lijkt op een priority encoder

Difference

- When/Else

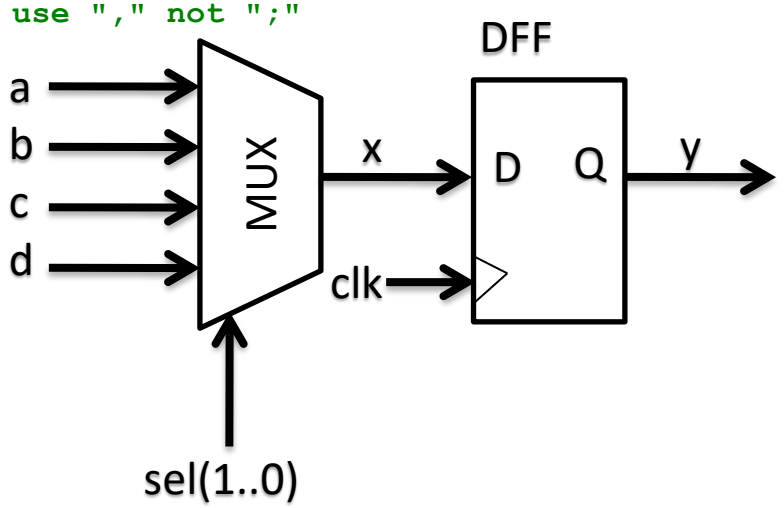


- With/Select



Synchronized Multiplexer

```
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------
entity synced_mux is
   port (a, b, c, d, clk: in  std_logic;
         sel        : in  std_logic_vector (1 downto 0);
         y       : out std_logic);
end gated_mux;
-----------------------------------------------
architecture behavioral of synced_mux is
        signal x: std_logic;
begin
        with sel select
                x <= a when "00",     -- use "," not ";"
                b when "01",
                c when "10",
                d when others;
process (clk)
begin
        if rising_edge(clk) then
        y <= x;
end process
end architecture
```



Sequential Code

- statements for sequential code:
  - **if**
  - **wait**
  - **loop**
  - **case**

- sequential code can be used to design both **sequential *and* combinational** circuits

- code within a **process**, **function** or **procedure** is sequential.

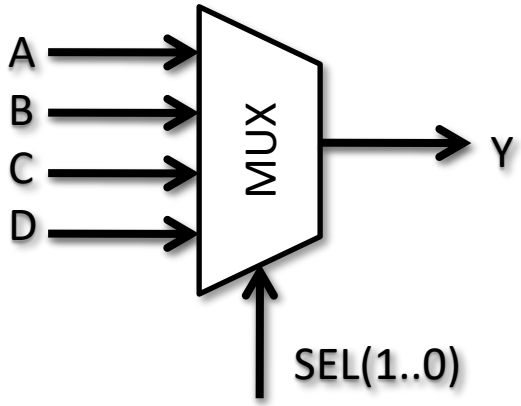
HOGESCHOOL ROTTERDAM

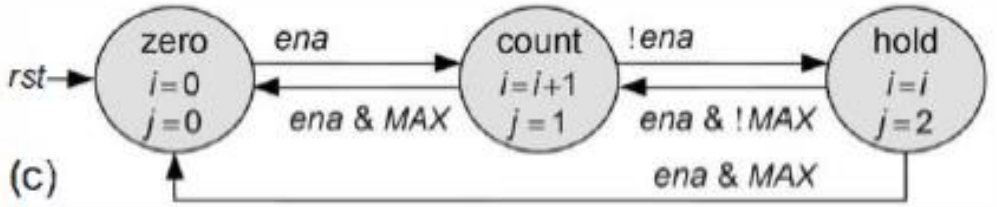
Sequential Code

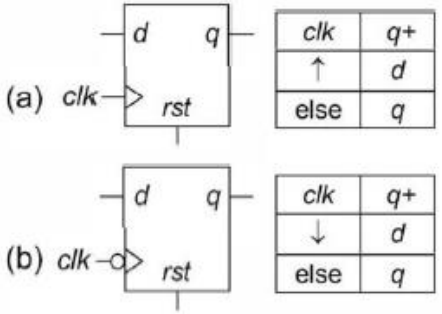
- **process** is a *sequential* section, located in the **architecture**
- note that multiple processes are allowed. they are *concurrent* to each other.
- support for the following *sequential* statements:
  - **if**
  - **wait**
  - **loop**
  - **case**
  - **with (since 2008)**
  - **when (since 2008)**

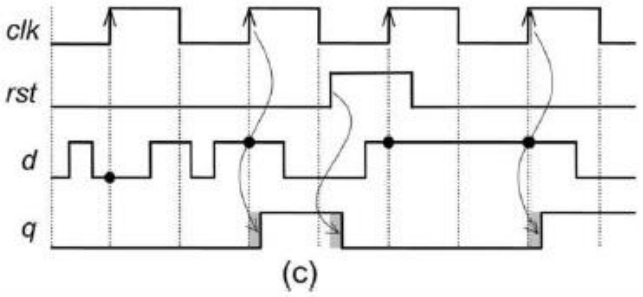
HOGESCHOOL ROTTERDAM

Sequential Code Circuit Examples

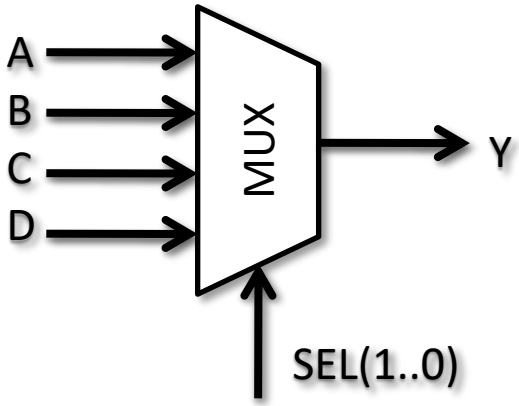








Multiplexer with sequential code



- A **process** has a **sensitivity list**
- The outputs of the **process** get updated if the value of an object in the list changes

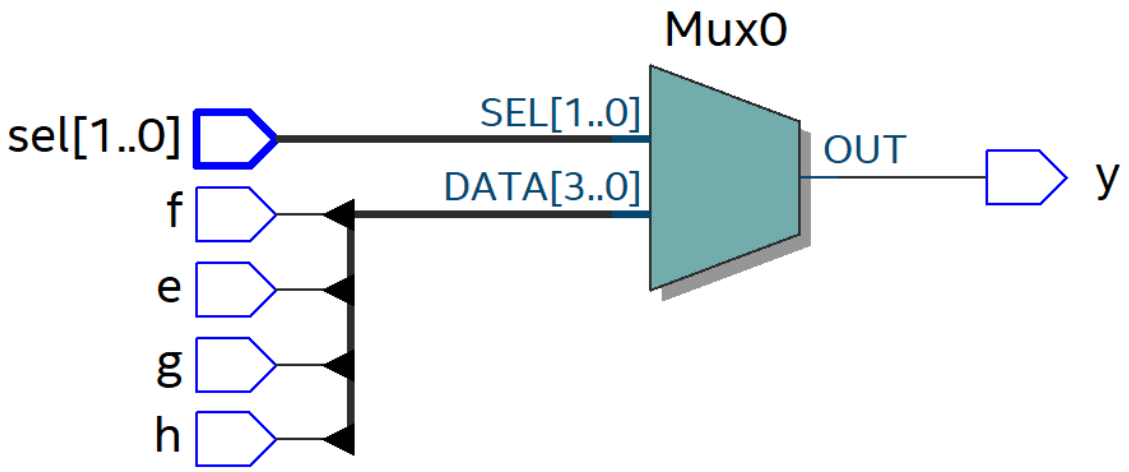
```
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------
entity seq_code_mux is
   port (a, b, c, d: in  std_logic;
         sel       : in  std_logic_vector (1 downto 0);
         y         : out std_logic);
end seq_code_mux;
-----------------------------------------
architecture seq_code_impl of seq_code_mux is
begin
         process(sel,a,b,c,d)
         begin
                  if    sel = "00" then
                      y <= a;
                  elsif sel = "01" then
                      y <= b;
                  elsif sel = "10" then
                      y <= c;
                  else
                      y <= d;
                  end if;
         end process;
end seq_code_impl;
```

Multiplexer with CASE

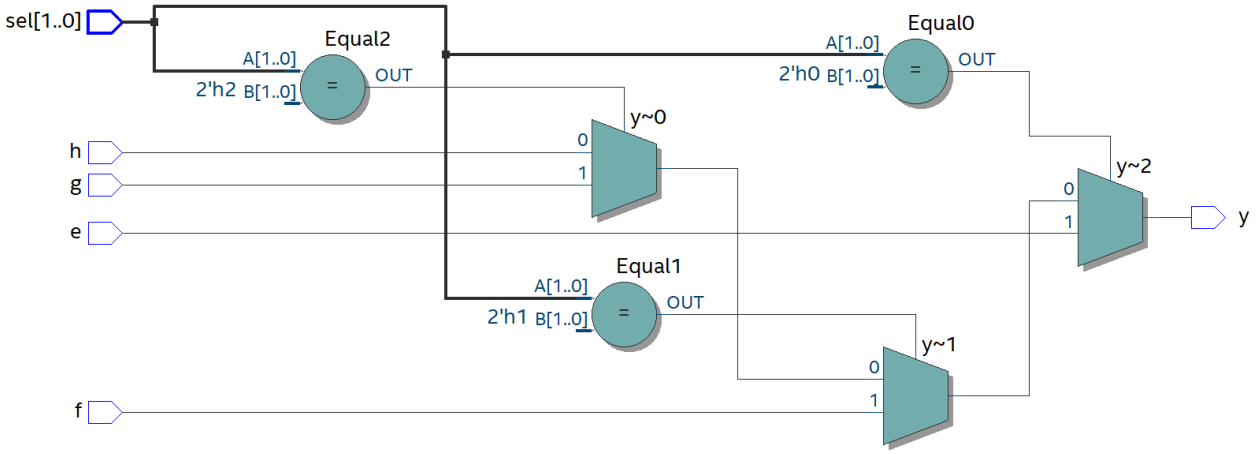
```
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------
entity seq_code_mux is
   port (a, b, c, d: in  std_logic;
         sel       : in  std_logic_vector (1 downto 0);
         y         : out std_logic);
end seq_code_mux;
-------------------------------------------
architecture seq_code_impl of seq_code_mux is
begin
         process(sel,a,b,c,d)
         begin
                  case sel is
                           when "00" =>
                                    y <= a;
                           when "01" =>
                                    y <= b;
                           when "10" =>
                                    y <= c;
                           when others =>
                                    y <= d;
                  end case;
         end process;
end seq_code_impl;
```

Difference

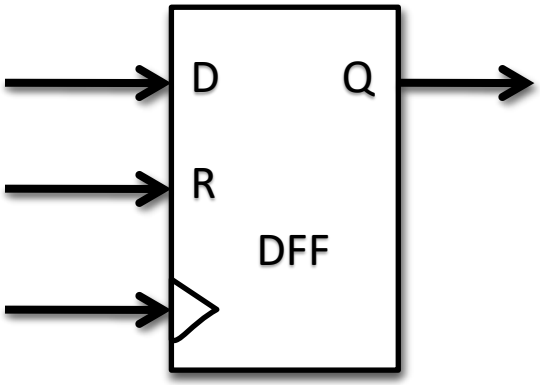
- Case



- If/Else



D Flip-Flop

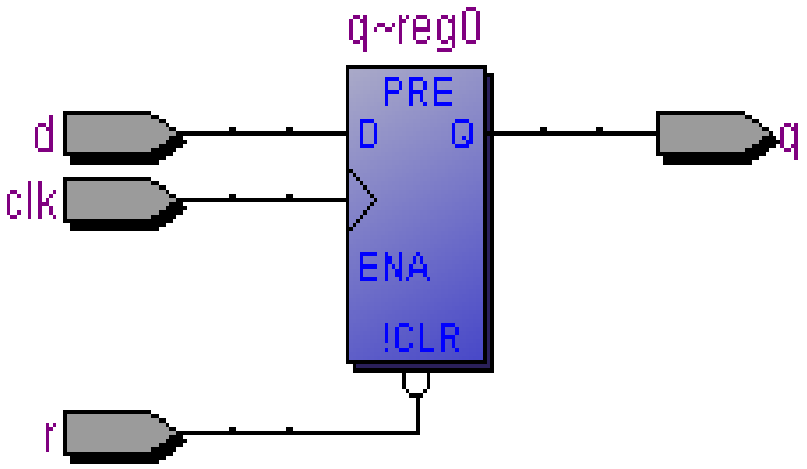


We only want to update Q when the CLK goes high, so only the CLK & R are in the sensitivity list

```
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------
entity dff_example is
   port (
            clk: in  std_logic;
            d,r: in  std_logic;
            q  : out std_logic
         );
end dff_example;
-----------------------------------------
architecture dff_implementation of dff_example is
begin
    process(clk, r )
    begin
        if r = '1' then
            q <= '0';
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end dff_implementation;
```

Technology Map of DFF

```
Sequential circuits can only be written with
sequential code
```

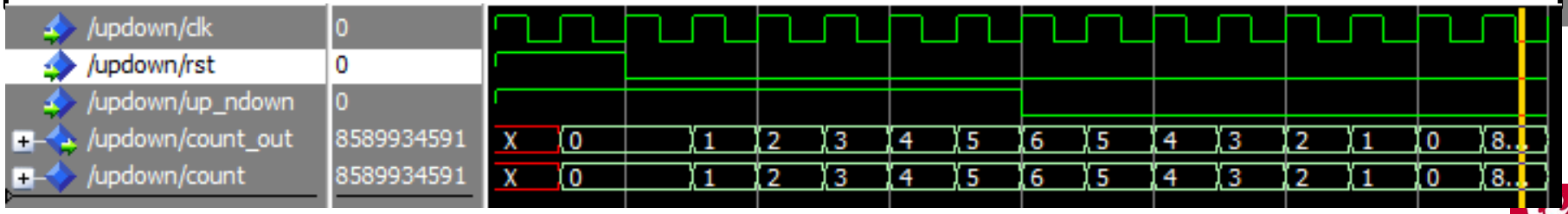


Up/down counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity updown is
        generic (
                counter_width : integer := 31
        );

        port (
                clk                 : in std_ulogic;
                rst                 : in std_ulogic;

                up_ndown  : in std_ulogic;

                count_out : out std_ulogic_vector(counter_width downto 0)
        );
end updown;
```

Up/down counter

```
architecture arch of updown is
        variable count : unsigned(counter_width downto 0);
begin
        process(clk)
        begin
                if rising_edge(clk) then
                        if rst = '1' then
                                count <= (others => '0');
                        else
                                if up_ndown = '1' then
                                        count <= count + 1;
                                else
                                        count <= count - 1;
                                end if;
                        end if;
                end if;
        end process;

        count_out <= std_ulogic_vector(count);

end arch;
```

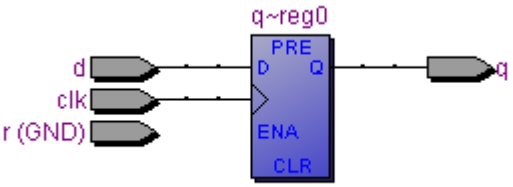


WAIT

- In a **process** you can use the **wait** keyword
- A process CANNOT have both a sensitivity list and **wait** statements
- Three types:
  - **wait until** *condition*
  - **wait on** *sig1, sig2, ..., sign* (sensitivity list)
  - **wait for** *time*
- **wait for** cannot be synthesized; only for simulation and test benches

D Flip-Flop with WAIT

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY dff_example IS
   PORT (
            clk: IN  STD_LOGIC;
            d,r: IN  STD_LOGIC;
          q  : OUT STD_LOGIC
            );
END dff_example;
---------------------------------------------
ARCHITECTURE dff_implementation OF dff_example IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL RISING_EDGE(clk);
          q <= d;
    END PROCESS;
END dff_implementation;
```



Not very handy without a reset ☹

Loops in VHDL

- **for** and **while** loops
- **for** ... **loop** repeats until the upperbound is reached
- static bounds; use constants to specify the upper bound of your loop

```
process(sel)
  ..
  begin
  for i in 0 to 5 loop
     x <= i;
  end loop;
end process;
```


HOGESCHOOL ROTTERDAM

Loops in VHDL

- **while** ... **loop** is similar in structure as the **for** ... **loop**

- see page 161 of your book for some examples with loops

```
process(sel)
  ..
  begin
  while i<10 loop
   ...do something...
  end loop;
end process;
```

Agenda

- Discussion of previous week
- Combinational versus sequential design
- Concurrent and sequential code
- **Signals versus variables**

Signals versus variables

- **SIGNAL**
  - Is eigenlijk een verbinding of draadje
  - Bedoeld om data over te brengen tussen verschillende blokken zoals concurrent statements en processen
- **VARIABLE**
  - Is meer een abstracte waarde
  - Het wordt gebruikt om iets te onthouden binnen een process

Signals versus variables

- **SIGNAL** properties:
  - Can *ONLY* be declared outside a **process** but can be used within a **process**
  - Within sequential code the signal is not 'updated immediately' (at the end of the **process**)
  - Only a *single* assignment is allowed to a signal in the whole code (multiple assignments in **processes** are fine, but only the last one will be effective!)
- **VARIABLE** properties:
  - Can *ONLY* be declared inside a **process**
  - Is 'updated immediately' and can be used in the next line of code
  - *Multiple* assignments are not a problem

HOGESCHOOL ROTTERDAM

Summary

- Combinational versus sequential design: no memory versus memory.
- In VHDL we can model combinational circuits with sequential statements
- Remember the differences between signals and variables

Homework

- Covered today:
  - Discussion of previous week
  - Combinatorial versus sequential design
  - Concurrent and sequential design
  - Signals versus variables

- Next week:
  - Chapter 12 (13) ‘Sequential Code’
    - ! Chapter 12.9 ‘SIGNAL and VARIABLE’
  - Chapter 14.3 – 14.5 ‘FUNCTION and PROCEDURE’

HOGESCHOOL ROTTERDAM