VHDL

5051158-xxxx

Lecture 3

Digital logic basics - sequential logic

VHDL Basics – part 2

V 1.0
Jarno Tuominen

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Contents of this lecture

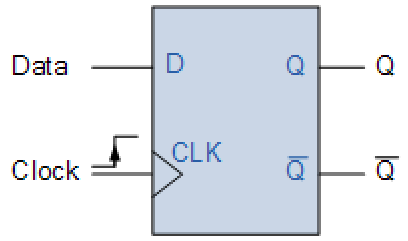
- Digital logic basics (sequential logic)
- VHDL basics
    - for creating sequential logic
    - VHDL “program structure”

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Digital logic basics – sequential logic

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Synchronous logic – flip flops



CLK	D	Q	$\overline{Q}$
↑	H	H	L
↑	L	L	H
L	X	no change	

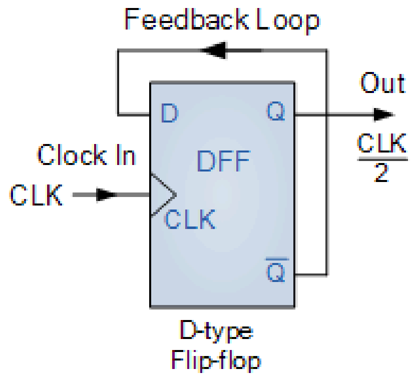
- Most commonly used: D-flip-flop
- D is sampled on a rising edge of CLK signal:
  - D is copied to Q
  - An inverted output, Q' is provided as well
- The output state is maintained until the next rising edge of the clock
- In essence, this is a single bit memory

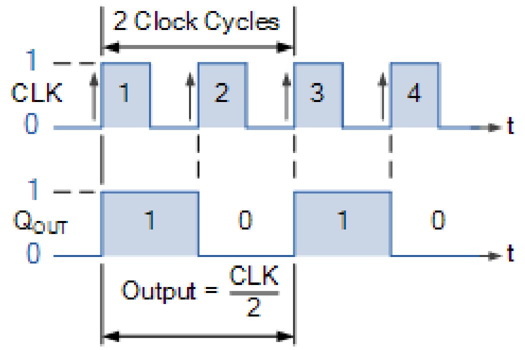
- Can have (and usually has) other ports:
  - Reset/Clear (RST, CLR), often active low): Forces Q to '0' (and Q' to '1')
  - Set (S): Forces Q to '1' (and Q' to '0')
  - Enable (ENA): If '0', clock is "masked", clock edge has no effect

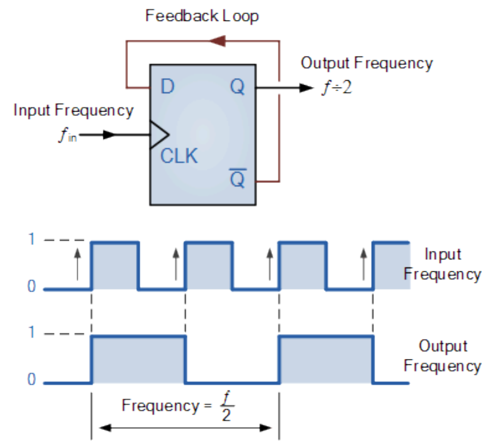
```
entity DFF_rstLow is
  port(
    Q : out std_logic;
    Clk,n_Rst,D :in std_logic);
end DFF_rstLow;

architecture rtl of DFF_rstLow is
begin
  process(Clk,n_Rst)
  begin
    if(n_Rst='0') then
      Q <= '0';
    elsif(rising_edge(Clk)) then
      Q <= D;
    end if;
  end process;
end rtl;
```

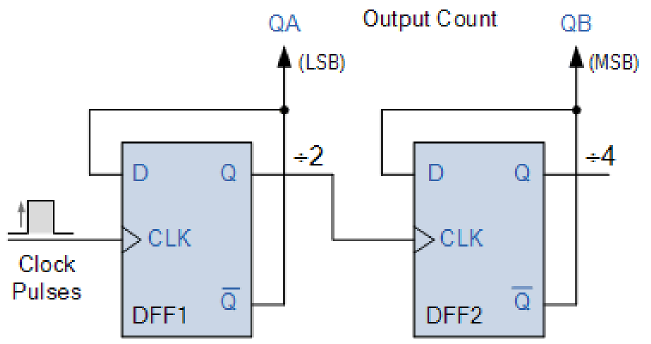
Synchronous logic – (async) counters (1)

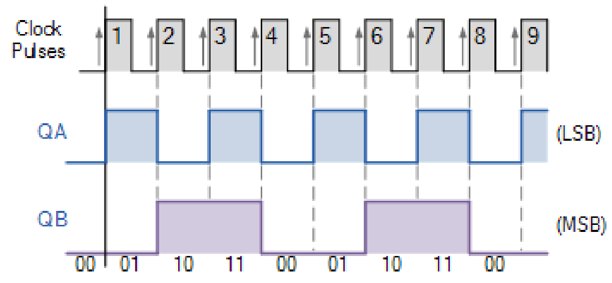
- A D-flipflop can be used as a clock divider: Route Q’ to D and apply CLK to clock input: Q changes state every other clock cycle
  - Divide-by-Two-Counter (modulo 2 counter)
- By cascading these, i.e. connecting Q’ to CLK of the next register, we have a ripple counter (Mod-4-counter)







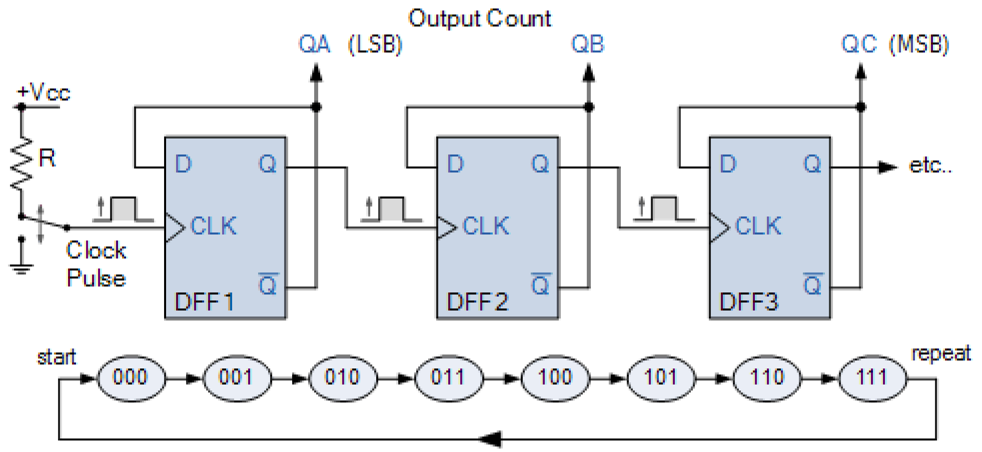




TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Synchronous logic – (async) counters (2)

- In a similar way, we can create longer counters
- Long ripple counters have one fundamental problem: After a clock edge, it takes some time until all the bits are changed -> This limits the maximum clock speed

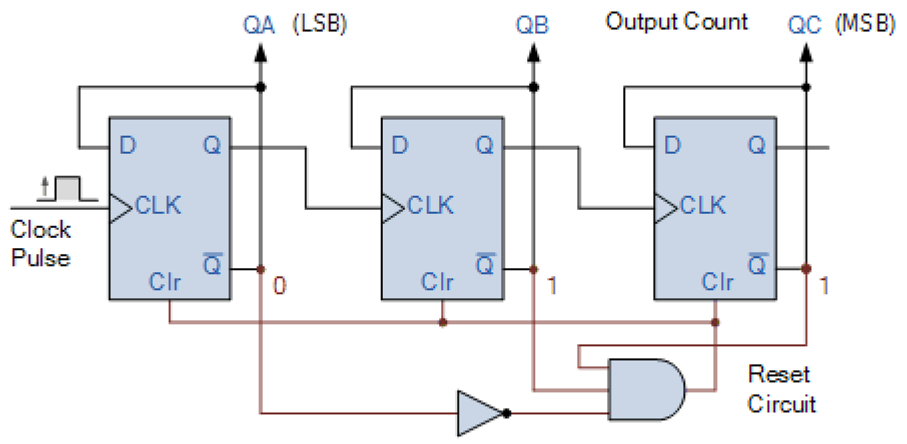


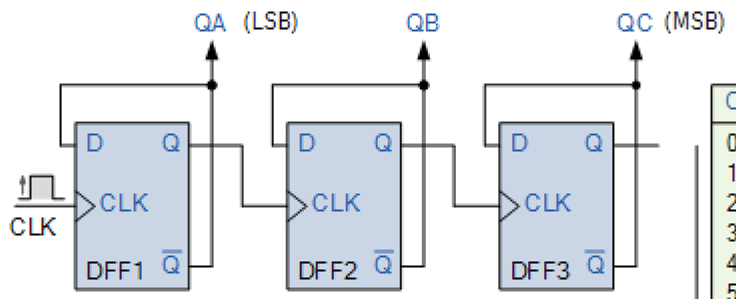
State transition diagram for a MOD-8 binary counter

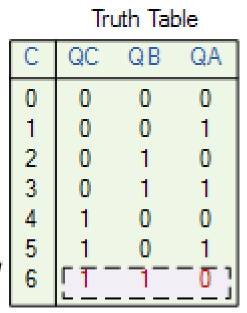
TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Modulo-N (async) counters

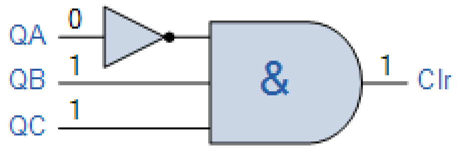
- Modulo N-counter, where N= 2^x are easy: the counter wraps over. If not, a reset condition is required
- Example: Mod-5 - counter

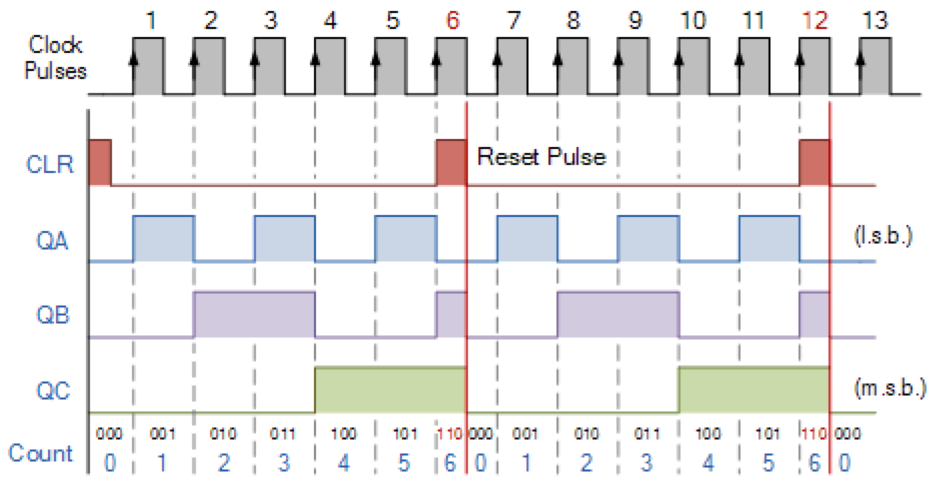





Truth Table

C	QC	QB	QA
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0

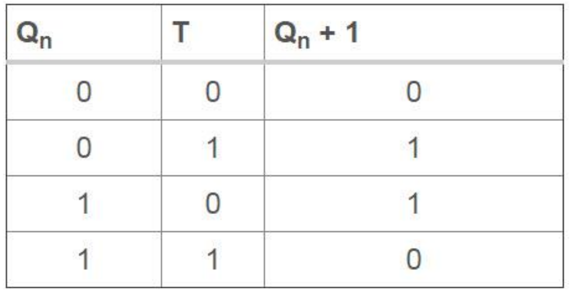




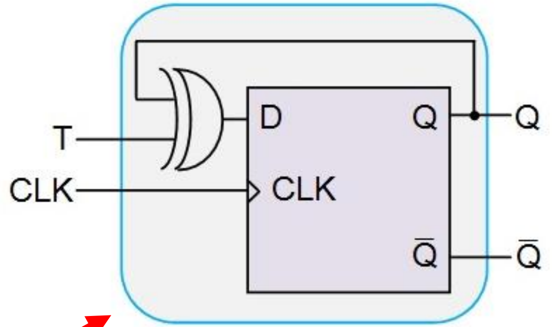
https://www.electronics-tutorials.ws/counter/mod-counters.html

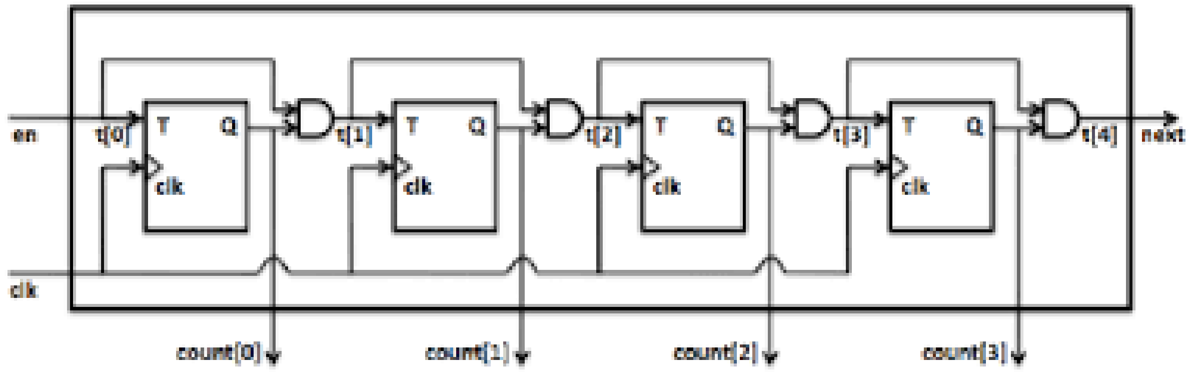
Synchronous counters

- To overcome the timing problem with asynchronous ripple-counters, we need to use synchronous counters instead
- This can be achieved (for example) by using T-flip-flops instead
    - The output toggles (changes state) on clock edge IF the input T is ‘1’ – otherwise no change
- D-flip-flops can be converted to T-flip-flop
- By cascading these, we’ll get a synchronous (BCD) counter



Q_n	T	$Q_n + 1$
0	0	0
0	1	1
1	0	1
1	1	0





Counters in VHDL – Modulo 16

These libraries define how to handle arithmetic operations with std_logic

```
library IEEE;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use IEEE.STD_LOGIC_1164.ALL;

entity Counter16 is
  port( Clk, Ena, n_Rst: in std_logic;
    Q: out std_logic_vector(3 downto 0));
end Counter16;
```

```
architecture rtl of Counter16 is
  signal Cnt_int: std_logic_vector(3 downto 0);
begin
   process(Clk,n_Rst)
   begin
      if n_Rst='0' then
         Cnt_int <= (others => '0');
      elsif(rising_edge(Clk)) then
         if Ena='1' then
               Cnt_int <= Cnt_int + 1;
         end if;
      end if;
   end process;
   Q <= Cnt_int;
end rtl;
```

Counters in VHDL – Modulo 10

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter16 is
  port( Clk, Ena, n_Rst: in std_logic;
     Q: out std_logic_vector(3 downto 0));
end Counter16;
```

```
architecture rtl of Counter16 is
  signal Cnt_int: std_logic_vector(3 downto 0);
begin
  process(Clk,n_Rst)
  begin
   if n_Rst='0' then
     Cnt_int <= (others => '0');
   elsif(rising_edge(Clk)) then
     if Ena='1' then
       if Cnt_int="1001" then
         Cnt_int <= (others => '0');
       else
         Cnt_int <= Cnt_int + 1;
       end if;
     end if;
   end if;
 end process;
 Q <= Cnt_int;
end rtl;
```

Reset (wrap around) condition defined here

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

VHDL “program” structure

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

VHDL "program" structure

- Libraries
    - Adds extra packages, like *#include*-statements in C, or *import* in Python
- Entity
    - Generics
    - Ports
- Architecture
    - Component declarations
    - Signal declarations
    - Component instantiations
    - Concurrent statements
    - Processes

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity led_thingy_top_2 is
    Port ( btn : in STD_LOGIC_VECTOR (3 downto 0);
           led4_r : out std_logic;
            led4_g : out std_logic;
            led4_b : out std_logic;
);
end led_thingy_top_2;

architecture rtl of led_thingy_top_2 is

    -- RGB driver
    component RGB_Driver
        Port (
          Buttons : in std_logic_vector (1 downto 0);
          RGB_out : out std_logic_vector (0 to 2));
    end component;

    -- group of RGB led signals
    signal RGB_Led_4: std_logic_vector(0 to 2);

begin
```

VHDL “program” structure (cont)

- Architecture
  - …
  - Component instantiations
  - Concurrent statements
  - Processes (not shown here)

```
architecture rtl of led_thingy_top_2 is

…

begin

    -- component instantiations
    i_RGB4_Driver: RGB_Driver
    port map (
        Buttons => btn(1 downto 0),
        RGB_Out => RGB_Led_4
    );

    -- map signal "RGB_Led_4" to actual output ports
    led4_r <= RGB_Led_4(2);
    led4_g <= RGB_Led_4(1);
    led4_b <= RGB_Led_4(0);


end rtl;
```

VHDL Components and hierarchical design

- A component declaration declares a virtual design entity interface that may be used in component **instantiation** statement
- A component represents an entity/architecture pair. It specifies a subsystem, which can be ***instantiated* in another architecture** leading to a hierarchical specification.
  - Component instantiation is like plugging a hardware component into a socket in a board
- A component must be **declared** before it is **instantiated**
  - The component declaration defines the virtual interface of the instantiated design entity ("the socket")
- Most often, the declaration takes in the main code (in the architecture, before "begin") or in separate packages (more on this later)
- Generics and ports of a component are **copies** of generics and ports of the entity the component represents.

```
component component_name is
    generic (generic_list);
    port (port_list);
end
component component_name;
```

```
architecture rtl of xxx is
  component XOR_4 is
    port(A,B: in std_logic_vector(0 to 3);
             C: out std_logic_vector(0 to 3));
  end component XOR_4;

  signal S1,S2 : std_logic_vector(0 to 3);
  signal S3 : std_logic_vector(0 to 3);

begin
  X1 : XOR_4
  port map(A => S1,B => S2,C => S3);

end architecture rtl;
```