

Designing with VHDL

(control questions)

Simple concurrent assignments

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;
```

```
entity a2of3 is  
port (a0 : in std_logic;  
      a1 : in std_logic;  
      a2 : in std_logic;  
      y  : out std_logic);  
end a2of3;
```

```
architecture a1 of a2of3 is  
signal g0, g1, g2 : std_logic;
```

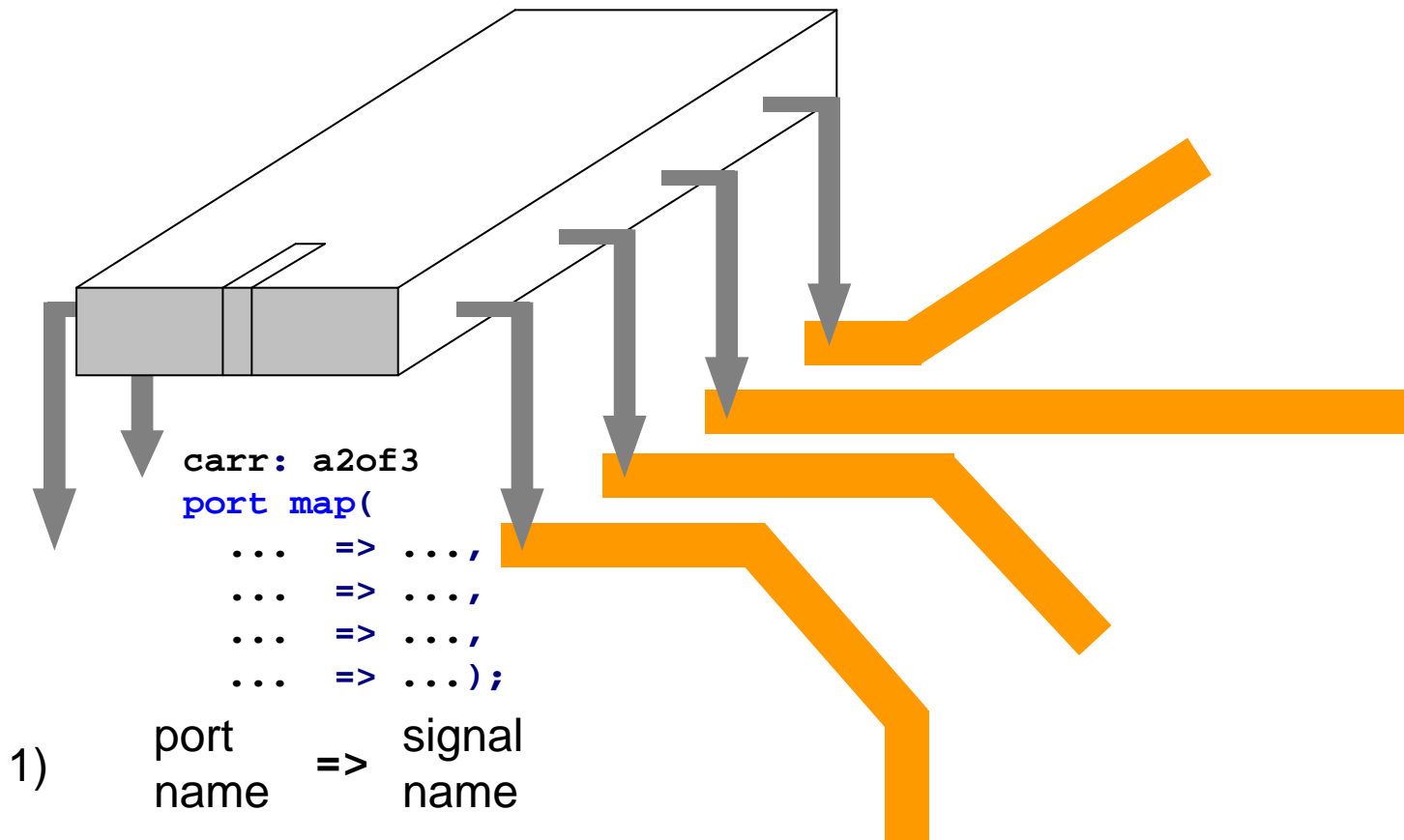
```
begin  
  g0 <= a0 and a1;  
  g1 <= a1 and a2;  
  g2 <= a2 and a0;  
  y <= g0 or g1 or g2;  
end;
```

```
architecture a2 of a2of3 is  
signal g0, g1, g2 : std_logic;
```

```
begin  
  g2 <= a2 and a0;  
  y <= g0 or (g1 or g2);  
  g1 <= a1 and a2;  
  g0 <= a0 and a1;  
end;
```

The two architectures are 1) equivalent; 2) different

Port-signal mapping



Instantiation of sub-blocks

```
-- component declaration
component a2of3 is
port (a0 : in  std_logic;
      a1 : in  std_logic;
      a2 : in  std_logic;
      y  : out std_logic);
end component;
```

a)

```
...
carr: a2of3
port map(
  a0 => a,
  a1 => b,
  a2 => cin,
  y  => cout);
...
```

b)

```
...
carr: a2of3
port map(a, b, cin, cout);
...
```

c)

```
...
carr: a2of3
port map(a, cin, b, cout);
...
```

The three instantiations are 1) equivalent; 2) all different;
3) one (which?) differs from the others

Multiple drivers

```
...  
(A, B, C : in std_logic;  
  Y       : out std_logic);  
...  
begin
```

→

<pre>Y <= not C; Y <= A or B;</pre>

```
end;
```

This code is

- 1) OK, the first assignment will be just ignored;
- 2) not allowed;
- 3) allowed, but represents an inverter and an or-gate with outputs shorted together

Multiple drivers

```
...  
begin  
→ Y <= A when OE_A='1' else 'Z';  
  Y <= B when OE_B='1' else 'Z';  
end;
```

This is allowed only when the signals are of the type

- 1) `std_logic`
- 2) `std_ulogic`
- 3) `bit`

Ports and signals

```
port (  
    a      : in  std_logic;  
    b      : in  std_logic;  
    c      : in  std_logic;  
    ya     : out std_logic;  
    yao    : out std_logic);  
  
...  
begin  
    → ya  <= a and b;  
      yao <= ya or c;  
...  
end;
```

1) This code is OK

2) A modification is necessary to get this code compiled (what?)

Data types

```
subtype reg_data is std_logic_vector(31 downto 0);
subtype reg_cmd  is std_logic_vector( 0 downto 3);
subtype byte     is std_logic_vector( 8 downto 1);
subtype fixedp   is std_logic_vector( 7 downto -2);
type  mem_array is array(0 to 63) of reg_data;
type  dat_array is array(7 to  0) of reg_data;
type  addr_data is std_logic_vector(15 downto 0);
type  state_type is (idle, run, stop, finish);
type  state_type is (idle, out, stop, finish);
```

Which declarations are not correct and why?

Ranges

```
generic (N : Natural := 4);  
port (  
    a : in  std_logic_vector(0 to N-1);  
    c : out std_logic_vector(N-1 downto 0);
```

→ ...
c <= a;

- 1) The code is not correct, as the indexes of the two vectors have different directions
- 2) The code is correct, $c(0)$ is connected to $a(N-1)$
- 3) The code is correct, $c(0)$ is connected to $a(0)$

Constants

```
constant Nbits   : Integer := 8;  
constant Nwords  : Natural  := 6;  
constant LowIdx  : Positive := 0;  
  
constant all0 : std_logic_vector(Nbits-1 downto 0) = (others => '0');  
  
constant Tco      : real := 5 ns;  
constant Tsetup   : time := 2 ns;  
constant Thold    : integer := 1 ns;  
constant Tdel     : time := 3;
```

Which declarations are not correct and why?

Concurrent assignments

```
y <= (a or b) and not c;  
y <= a or b and not c;  
y <= a and b or not c;  
y <= a and b and not c;  
y <= (a nor b) nor c;  
y <= a nor b nor c;  
y <= (a nand b) nand c;  
y <= a nand b nand c;  
y <= a xor b and c;  
y <= (a xor b) and c;
```

Which assignments are not correct and why?

Conditional and selected assignments

```
y <= (a or b) when c = '0';
```

```
with a & b & c select  
  y <= '1' when "110" | "100" | "010",  
       '0' when "011" | "111";
```

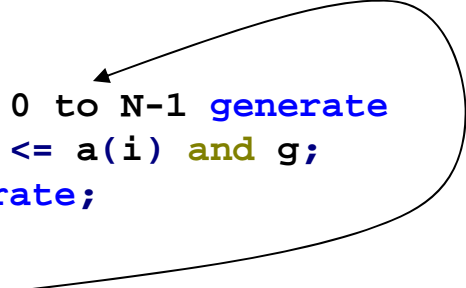
```
with a & b & c select  
  y <= '1' when "110" | "100" | "010",  
       '0' when "011" | "110",  
       '-' when others;
```

Why these assignments are not correct?

Generate

```
...
generic (N : Natural := 4);
port (
  a : in  std_logic_vector(N-1 downto 0);
  g : in  std_logic;
  c : out std_logic_vector(N-1 downto 0) );
end for_gen;
architecture ... of for_gen is
→ signal i : Integer;
begin

gn: for i in 0 to N-1 generate
→   c(i) <= a(i) and g;
   end generate;
```



The declaration of `i` is necessary

?

`i` is automatically declared within the `for...generate` loop

The range must be exactly the same incl. the direction (`downto`) as in the declaration of `c ()` and `a ()`

?

The index `i` of `c(i)` and `a(i)` must be within the limits `0..N-1`

The `integer` type

How many bits will be used to store the following integers:

```
signal my_int1 : Integer range -1 to 16;
```

5

6

4

```
signal my_int2 : Integer range -32 to 2;
```

6

8

7

Mathematical operations with **integer_s**

```
process
variable byte : Integer range 0 to 16#FF#;
variable sint : Integer range -128 to 127;
variable word : Integer range 0 to 16#FFFF#;
variable intg : Integer range -2**15 to 2**15-1;
begin
```

byte := 20;

byte := -20;

sint := -20;

sint := 150;

word := -1;

word := 1000**2;

word := 16#1000#;

Which assignments are
not correct and why?

Adder

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity adder is
generic (N : Natural := 8);
port (
  cin : in  std_logic;
  a    : in  std_logic_vector(N-1 downto 0);
  b    : in  std_logic_vector(N-1 downto 0);
  cout: out std_logic;
  y    : out std_logic_vector(N-1 downto 0));
end adder;
architecture behav of adder is
signal sum : std_logic_vector(N downto 0);
begin
  sum  <= cin + ('0' & a) + ('0' & b);
  y    <= sum(y'range);
  cout <= sum(sum'high);
end;
```

This adder was designed for adding two `std_logic_vectors` representing unsigned integers.

Can we use the generated hardware to add correctly **a** and **b** if they represent signed integers?

Signal vs. variable

```

signal si : Integer range -7 to 7;
begin
process
variable vi : Integer range -7 to 7;
begin

```

Hint: the leftmost is the initial value

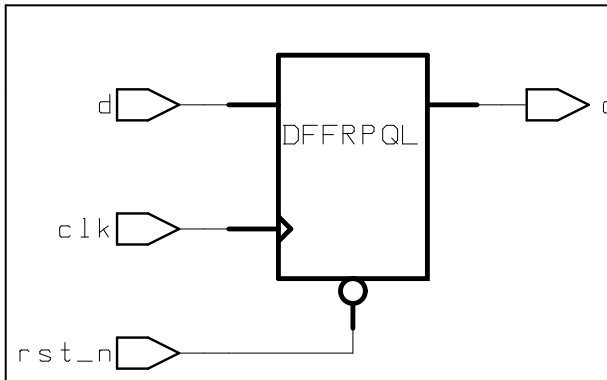
	si	vi
si <= 0;		
vi := 0;		
si <= si + 1;		
vi := vi + 1;		
wait for 10 ns;	1 -6	1 -6
si <= si + 1;		
vi := vi + 1;		
wait for 10 ns;	-5 2	-5 2
si <= si + 1;		
vi := si;		
wait for 10 ns;	-4 3	-5 -4 2 3
wait;		
end process;		

Which is the correct value of **si** and **vi** after the **waits**?

DFF with asynchronous reset

```
process(clk, rst_n)
begin
  if rst_n = '0' then q <= '0'; end if;
  if clk'event and clk='1' then
    q <= d;
  end if;
end process;
```

```
process(clk, rst_n)
begin
  if rst_n = '0' then q := '0';
  elsif clk'event and clk='1' then
    q := d;
  end if;
end process;
```



Find the errors!

```
process(clk)
begin
  if rst_n = '0' then q <= '0';
  elsif clk'event and clk='1' then
    q <= d;
  end if;
end process;
```

```
process(clk, rst_n)
begin
  if rst_n = '0' then q <= '0';
  elsif clk'event and clk then
    q <= d;
  end if;
end process;
```

Decoder with enable

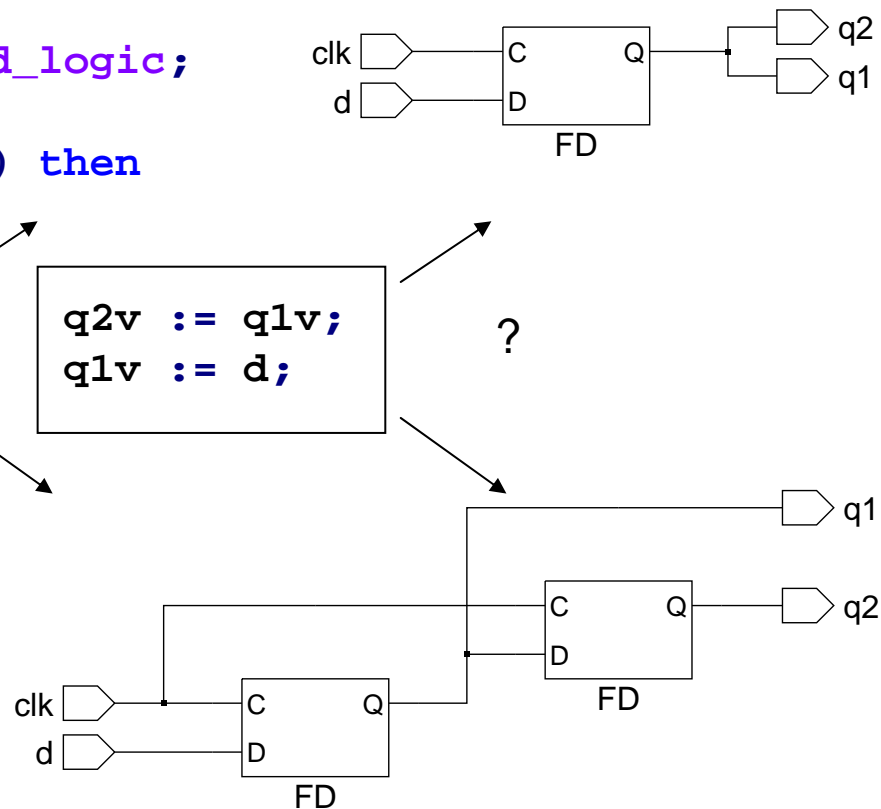
This entity is supposed to be a decoder with enable, but has a bug (syntax & compile is OK)

```
...
port (
    a      : in  std_logic_vector(1 downto 0);
    e      : in  std_logic;
    y      : out std_logic_vector(2 downto 0));
...
process(a, e)
begin
    if      a = "00" then y <= "00" & e;
    elsif  a = "01" then y <= '0' & e & '0';
    elsif  a = "10" then y <= e & "00";
end if;
end process;
```

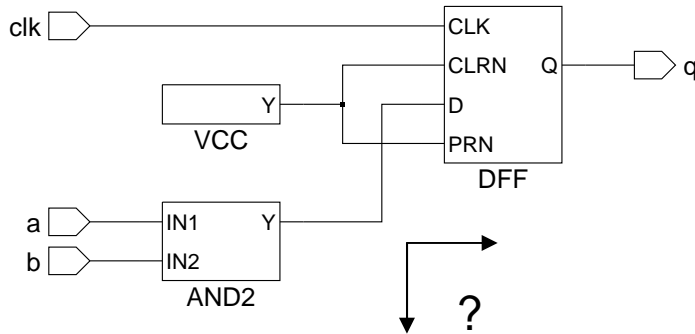


DFFs with variables(1)

```
process(clk)
variable q1v, q2v : std_logic;
begin
    if rising_edge(clk) then
        q1v := d;
        q2v := q1v;
        q2v := q1v;
        q1v := d;
    end if;
    q1 <= q1v;
    q2 <= q2v;
end process;
```



DFFs with variables(2)



```

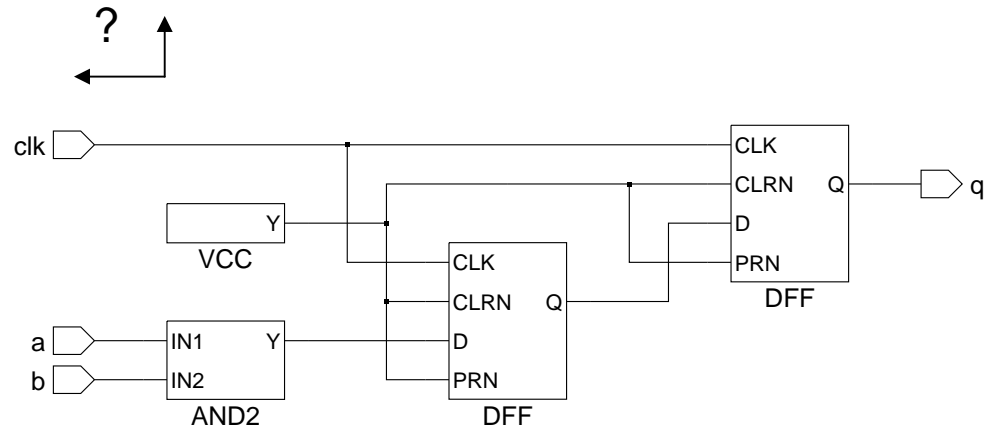
signal qv : std_logic;
...
process(clk)
begin
    if rising_edge(clk) then
        qv <= a and b;
        q <= qv;
    end if;
end process;

```

```

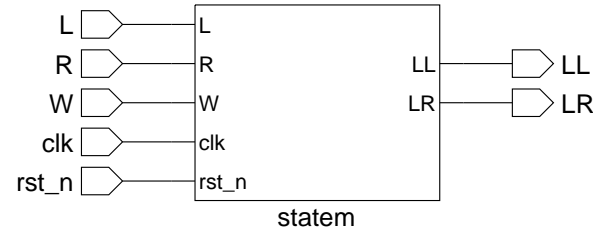
process(clk)
variable qv : std_logic;
begin
    if rising_edge(clk) then
        qv := a and b;
        q <= qv;
    end if;
end process;

```

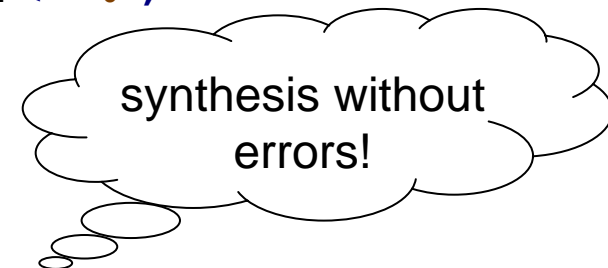


State machines in VHDL

```
type state_type is (S0, SL, SR, SA);
signal present_st, next_st : state_type;
begin
process(present_st, L, R, W)
begin
```



```
    next_st <= present_st;
    case present_st is
    when S0 => if      W = '1' then next_st <= SA; LR <= '1'; LL <= '1';
                elsif L = '1' then next_st <= SL; LL <= '1';
                elsif R = '1' then next_st <= SR; LR <= '1';
                end if;
    when SL => if L = '0' and R = '1' then next_st <= SR; LR <= '1'; LL <= '0';
                else next_st <= S0; LR <= '0'; LL <= '0';
                end if;
    when SR => if L = '1' then next_st <= SL; LL <= '1';
                else next_st <= S0; LL <= '0'; LR <= '0';
                end if;
    when SA => next_st <= S0; LL <= '0'; LR <= '0';
    end case;
end process;
process(clk, rst_n)
begin
    if      rst_n = '0'                then present_st <= S0;
    elsif clk'event and clk='1' then present_st <= next_st;
    end if;
end process;
end;
```

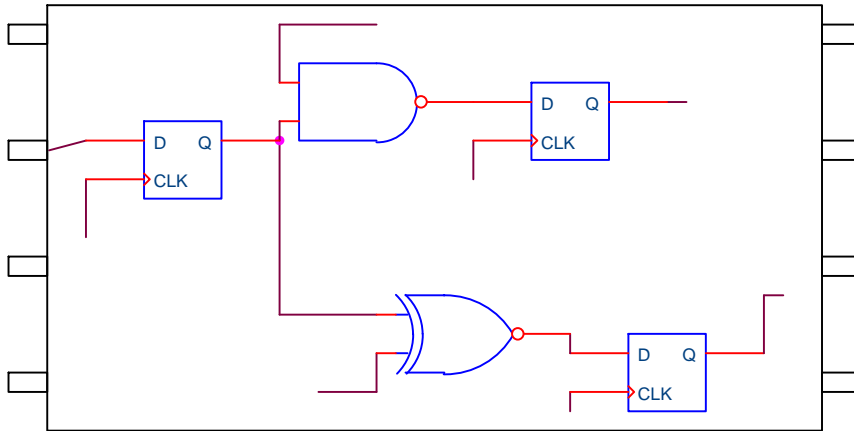
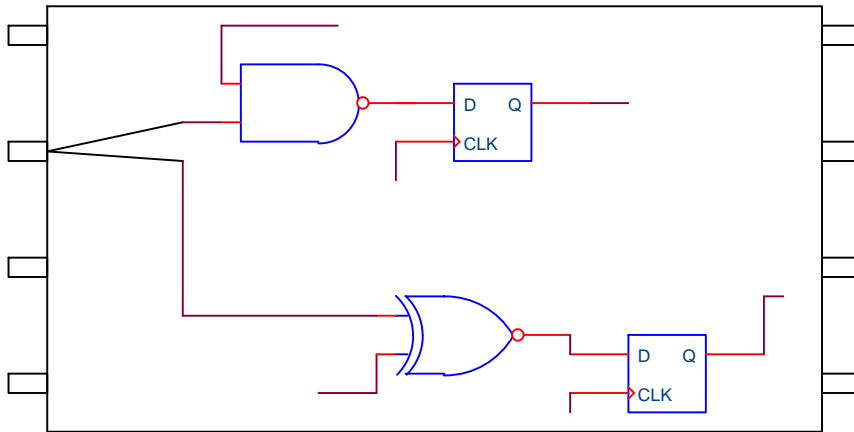


What is wrong in this description?

State machine - encoding

1. A state machine with 5 states, encoded binary will have:
 - a) 8 states in total
 - b) 5 states in total
 - c) 32 states in total
2. The same state machine encoded one-hot will have:
 - a) 5 states in total
 - b) 32 states in total
 - c) 8 states in total

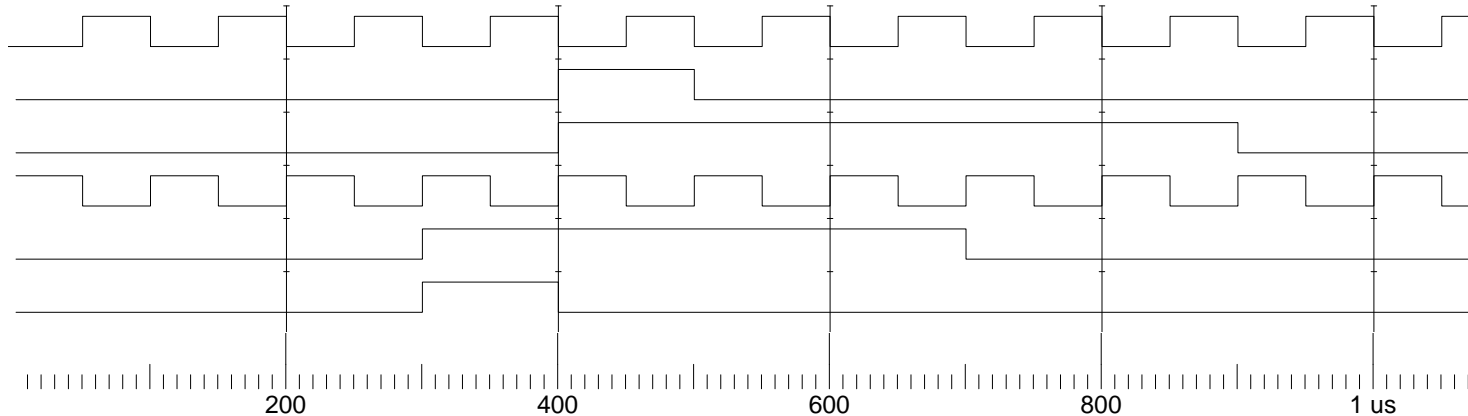
Synchronize input signals



Which one of the two schemes should be avoided and why?

Simple test bench example

rst1
clk1
d1
rst2
clk2
d2



Find the waveform of each signal!

```

signal clk1    : std_logic := '1';
signal rst1    : std_logic;
signal d1      : std_logic;
signal clk2    : std_logic;
signal rst2    : std_logic;
signal d2      : std_logic;
begin
    clk1 <= not clk1 after 50 ns;
    rst1 <= '0' after 0 ns,
           '1' after 300 ns,
           '0' after 400 ns;
    d1   <= '0' after 0 ns,
           '1' after 400 ns,
           '0' after 500 ns;

```

```

process
begin
    clk2 <= '0';
    wait for 50 ns;
    clk2 <= '1';
    wait for 50 ns;
end process;

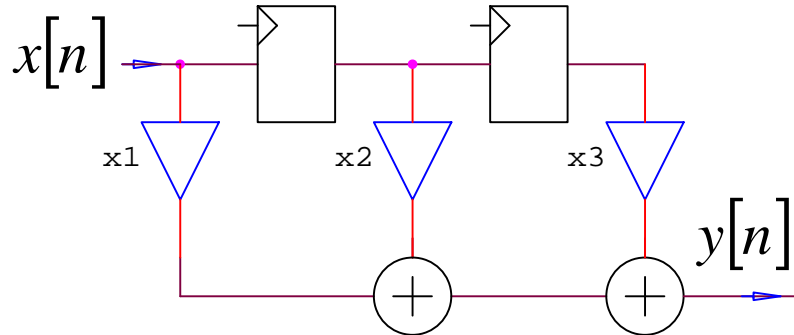
```

```

process
begin
    rst2 <= '0';
    d2   <= '0';
    wait for 300 ns;
    rst2 <= '1';
    wait for 100 ns;
    d2   <= '1';
    wait for 300 ns;
    rst2 <= '0';
    wait for 200 ns;
    d2   <= '0';
    wait;
end process;

```

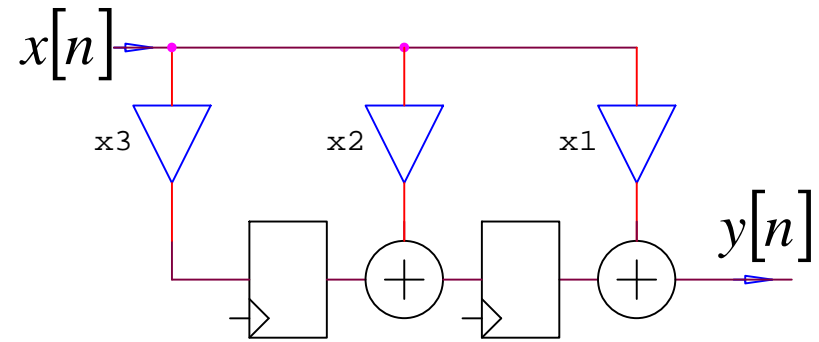
Digital Filters(1)



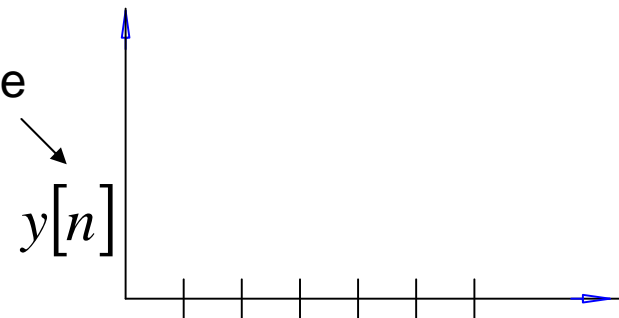
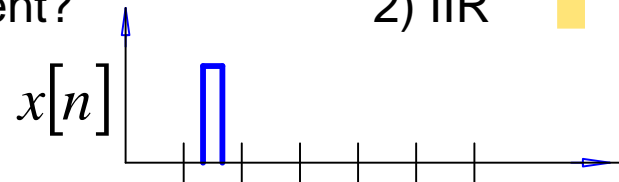
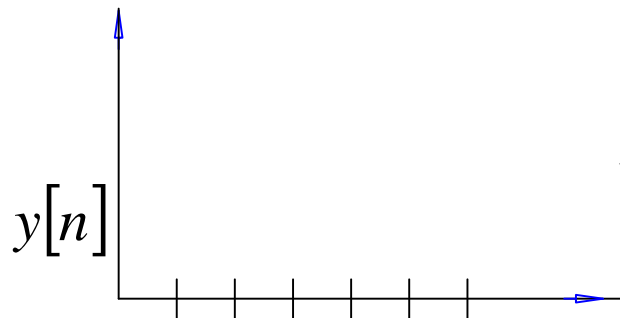
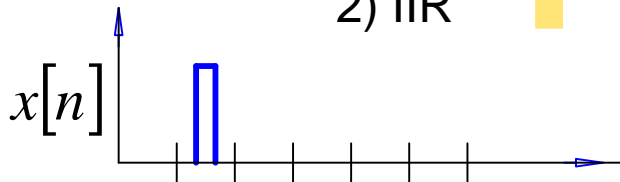
1) FIR
2) IIR



Are the two filters
equivalent?



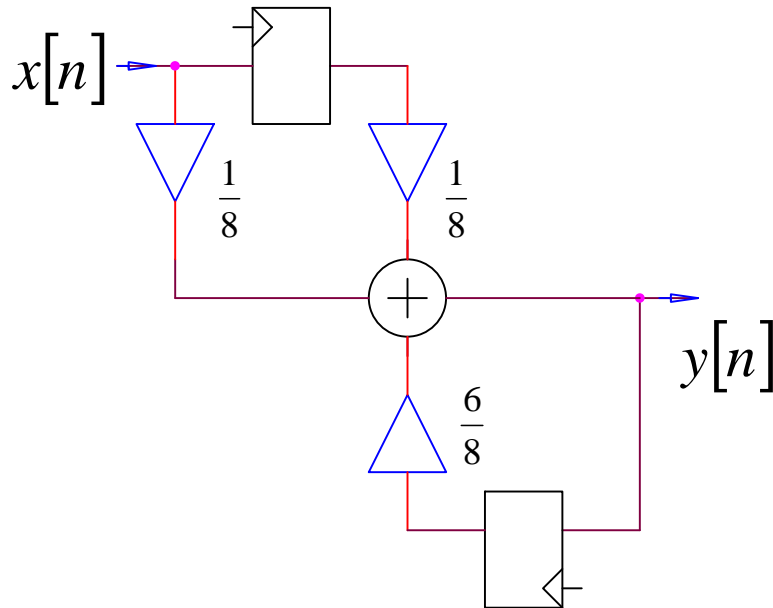
1) FIR
2) IIR



plot the
response

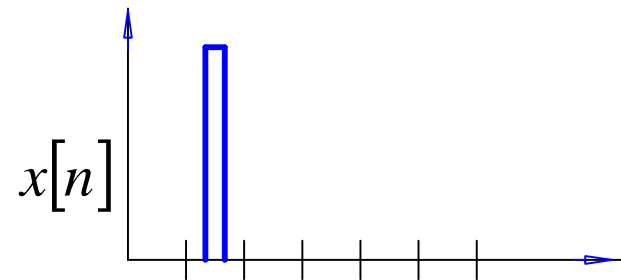


Digital Filters(2)

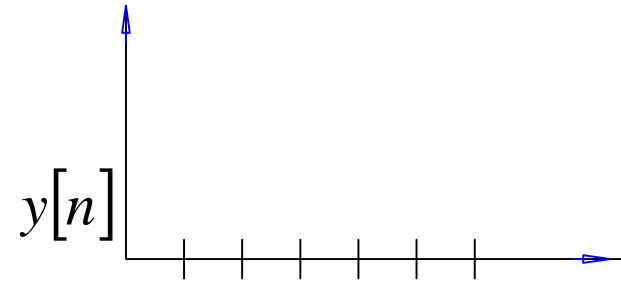


What kind of filter is this one?

- 1) FIR
 - 2) IIR
- ?

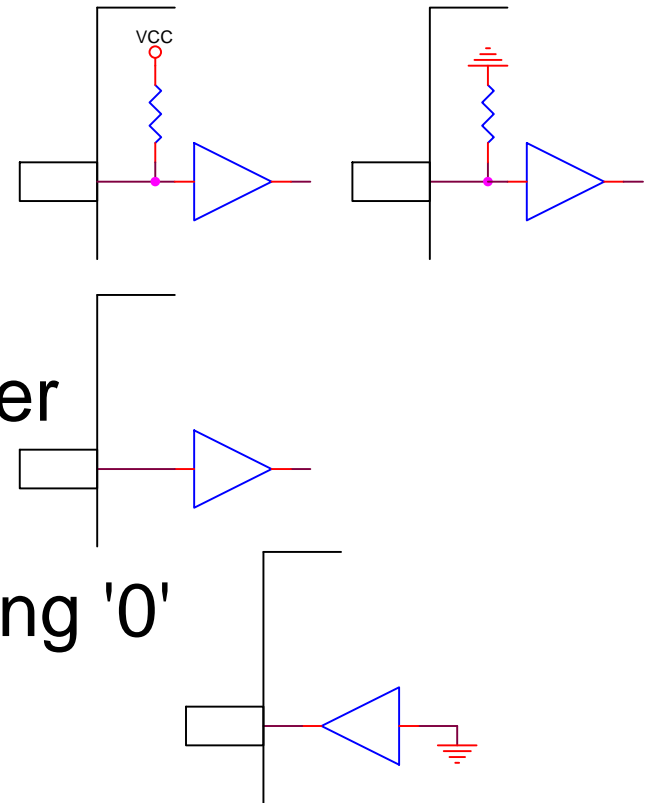


Sketch the response of the filter



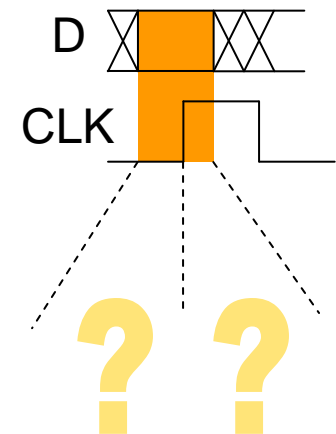
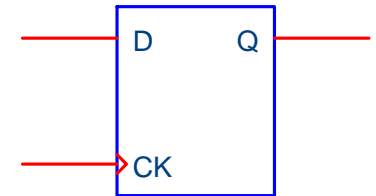
I/O

- The unused unconnected input pins should be
 - pulled high or low by internal resistors
 - left free in order to save power
 - programmed as outputs driving '0'



Timing(1)

- Positive setup time means
 - the D input must be stable some time before the rising edge of the clock
 - the D input must be stable some time after the rising edge of the clock
- Positive hold time means
 - the D input must be stable some time before the rising edge of the clock
 - the D input must be stable some time after the rising edge of the clock



Timing(2)

- The operation of a chip at 1 Hz can be affected by:
 - setup time violations
 - hold time violations
- The operation of a chip at 100 MHz can be affected by:
 - setup time violations
 - hold time violations

Special cores, I/O timing

- In order to multiply the input clock by $5/7$ we can use a
 - PLL ?
 - DLL ?
- In order to achieve identical setup/hold times on all bits of a synchronous 32-bit input bus we must first use the D-flip-flops
 - in the core logic cells ?
 - in the I/O cells ?