Designing with VHDL

Designing with VHDL

- Overview of the Hardware Description Languages (HDL)
- VHDL for synthesis
  - Signals, types, vectors, hierarchy, attributes
  - Combinational circuits – concurrent vs. sequential assignments
  - Examples
  - Mathematical operations, packages
  - Sequential circuits – DFFs and latches
  - More examples
  - State machines
  - Error protection and error correction

Overview HDL – ABEL, AHDL, Verilog

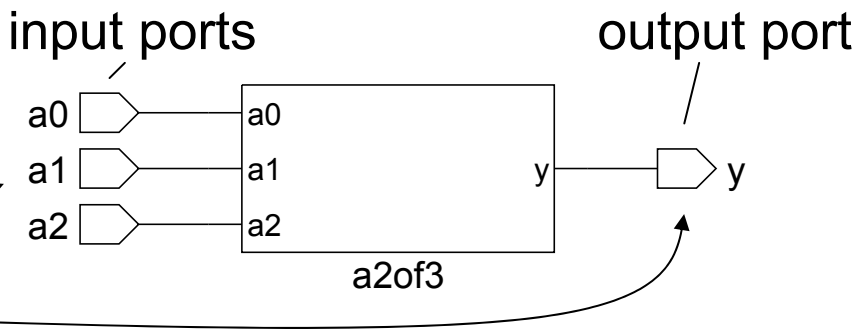
- ABEL
  - Originally developed for SPLDs and still in use for SPLDs
  - Now owned by Xilinx
- AHDL
  - Developed by Altera, still used in Altera library components
  - Syntax similar to Ada
- Verilog HDL
  - Together with VHDL the standard HDL now
  - Syntax similar to C
- Other → SystemC, SystemVerilog, Verilog-AMS

VHDL

- VHDL = VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuit
- Developed on the basis of **`ADA`** with the support of the USA militaries, in order to help when making ***documentation*** of the digital circuits
- The next natural step is to use it for ***simulation*** of digital circuits
- And the last very important step is to use it for ***synthesis*** of digital circuits
- Versions: 1987, 1993, 2000, 2002 and 2008
- Together with **`Verilog`** is the mostly used language for development of digital circuits
- Extensions for simulations of analogue circuits

Structure of an **entity** in VHDL - example

```
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 a 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;
```



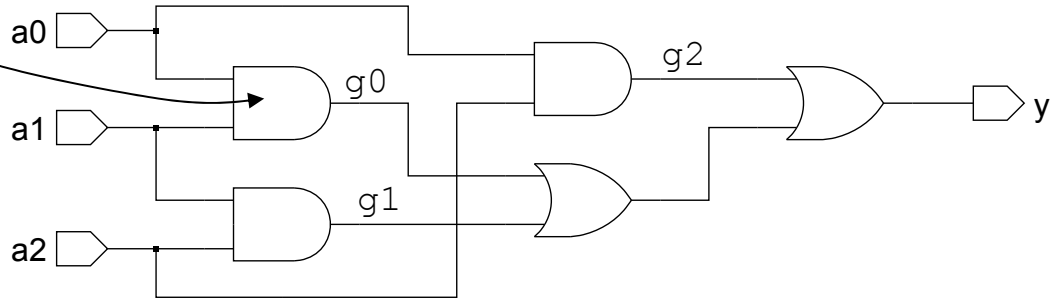
internal signals

VHDL is not case-sensitive!

Recommendation: 1 file – 1 entity

filename = name of the entity

the order of the assignments here is **not** important!



Structure of an **entity** in VHDL

entity
port
in out
inout
buffer
architecture
signal

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```
+ other library declarations, this is the standard minimum

```
entity <entity_name> is
port (
   <port_name>   : <in|out|inout|buffer> <signal_type>;
   ...
   <port_name>   : <in|out|inout|buffer> <signal_type>);
end <entity_name>;

architecture <arch_name> of <entity_name> is

...
signal <internal_signal> : <signal_type>;
...
begin
   -- comment to the end of the line
   ...
end [<arch_name>];
```

port type

+ optional `type` and `constant` declarations

Unlike C and Verilog, VHDL is not case-sensitive!

```
/* block comment
   in VHDL-2008 */
```

Instantiation of sub-blocks

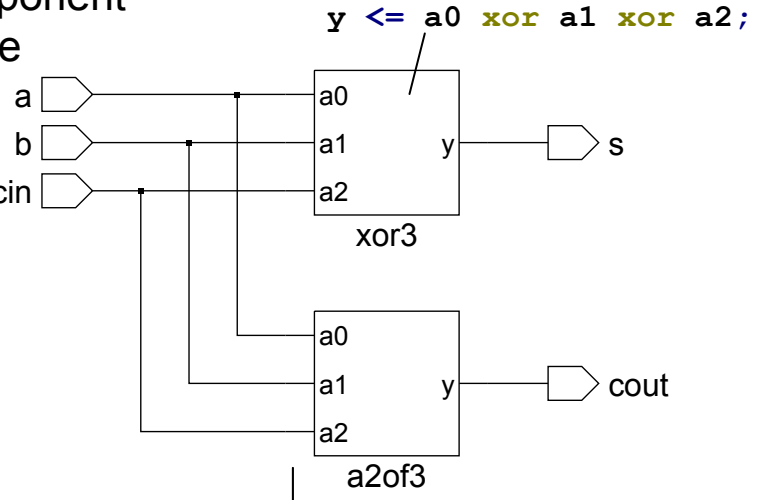
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

entity fadd_top is
port (
    cin     : in  std_logic;
    a       : in  std_logic;
    b       : in  std_logic;
    cout    : out std_logic;
    s       : out std_logic);
end fadd_top;

architecture struct of fadd_top is
-- component declarations
component xor3 is
port (a0 : in  std_logic;
      a1 : in  std_logic;
      a2 : in  std_logic;
      y  : out std_logic);
end component;

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

declare the "pinout" of the used components

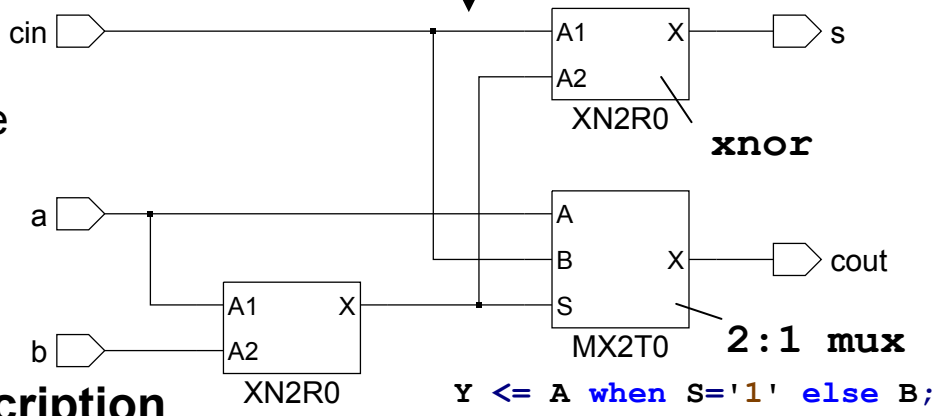
```
begin
sum: xor3
port map(
  a0  => a,
  a1  => b,
  a2  => cin,
  y   => s);

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

component name

label



Synthesis + techn. mapping



known as STRUCTURAL description

Instantiation – port-signal mapping

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

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

swapping of the pairs is not important

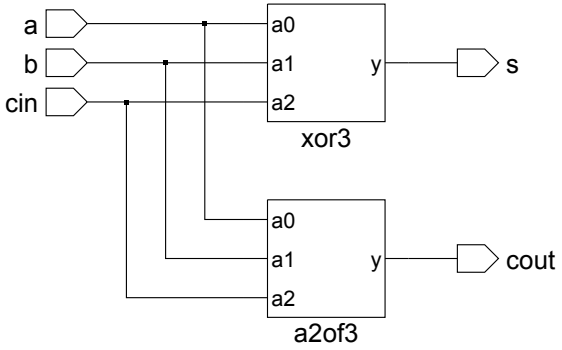
There is a shorter way to connect the ports of the component to the signals, shown to the right. ***Not recommended***, as in this case the mapping relies on the correct order!

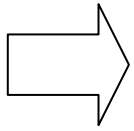
```
carr: a2of3
port map(a, b, cin, cout);
```

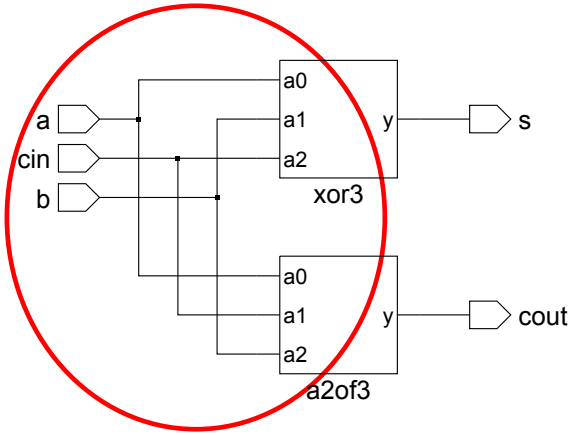
a wrong order of the signals here will change the circuit!!!

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

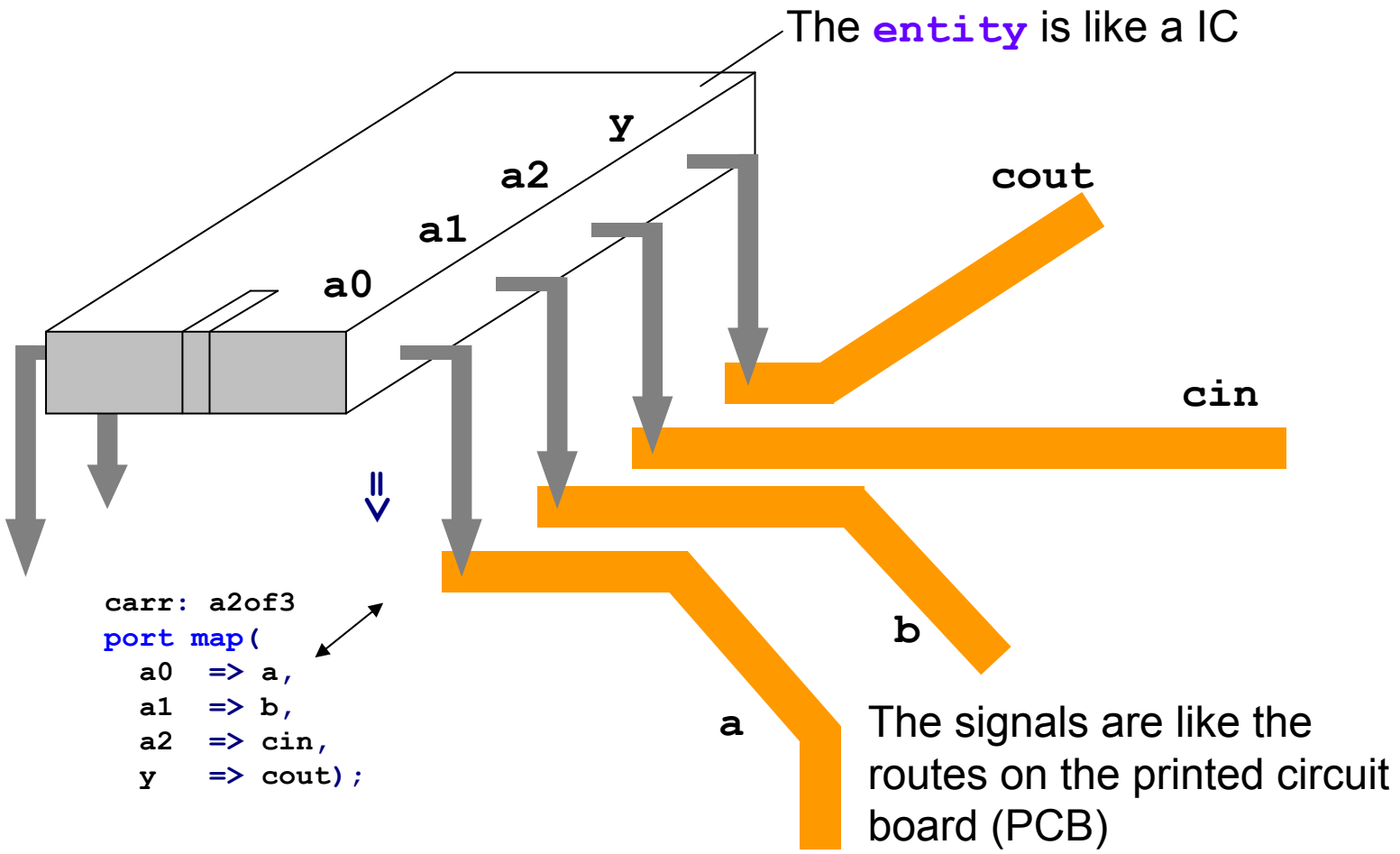
b and **cin** are swapped







Port-signal mapping – how to remember



time
Boolean
Integer
Natural
Positive

Types of data in VHDL(1)

- **time** (**fs, ps, ns, us, ms, sec, min, hr**)
  - 1 ns, 20 ms, 5.2 us
- **real** (**-1e38..+1e38**)
- **integer** (-(2^{31}-1) .. 2^{31}-1) with predefined subtypes **natural** (≥0) and **positive** (>0)

```
signal counter : Integer;
signal timer   : Natural;
```

- **boolean** has two possible values **FALSE** and **TRUE**
  - Not intended for electrical signals!
  - Typically used when checking some conditions, like

```
if a  = b then --  equal
if a /= b then --  not equal
if a >  b then --  larger
if a <  b then --  smaller
if a <= b then --  smaller or equal
if a >= b then --  larger or equal
```

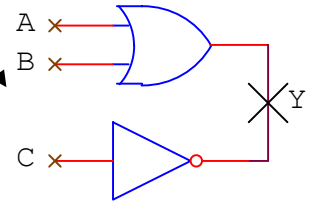
↖the result of the comparison is a **boolean**

For other examples see "Working with vectors" and "Signal attributes"

Types of data in VHDL(2)

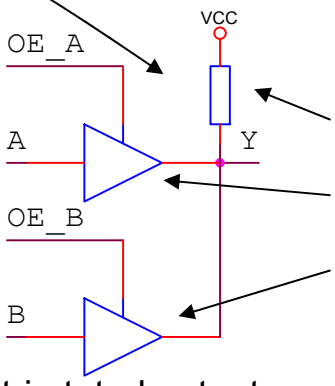
- **bit** has two possible values **'0'** and **'1'**
  - These two values are not enough to model real hardware!
- **std_(u)logic** to the **'0'** and **'1'**, introduced 7 additional values for tri-stated (**'Z'**), unknown (**'X'**), weak 0 (**'L'**), weak 1 (**'H'**), weak unknown (**'W'**), uninitialized (**'U'**) and don't care (**'-'**)

This is allowed only when using **std_logic** but not **std_ulogic** or **bit**!




```
Y <= not C;
Y <= A or B;
```

Example for pull-up (weak 1) and tri-stated outputs, **std_logic** is required

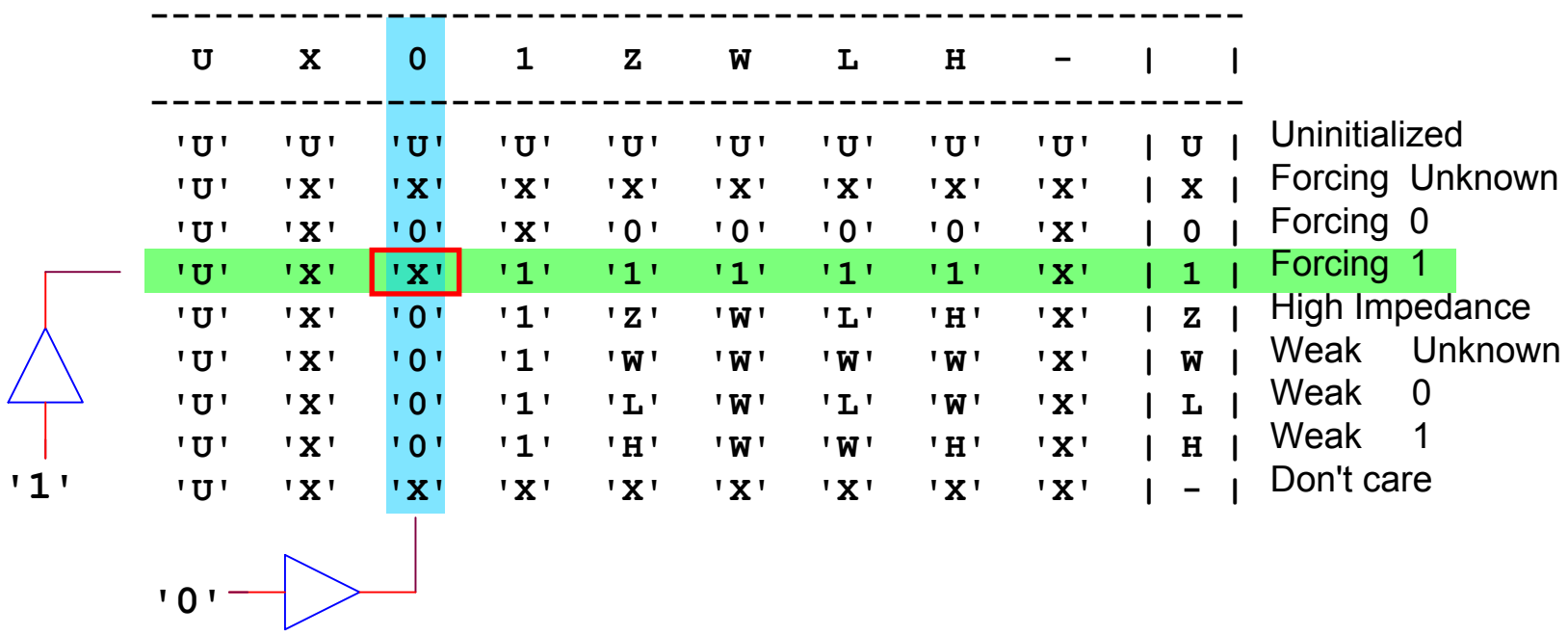


tri-stated output

```
Y <= 'H';
Y <= A when OE_A='1' else 'Z';
Y <= B when OE_B='1' else 'Z';
```

The std_logic type

… is a resolved version of **std_ulogic**, this means there is a method to resolve the conflicts when one signal has multiple drivers



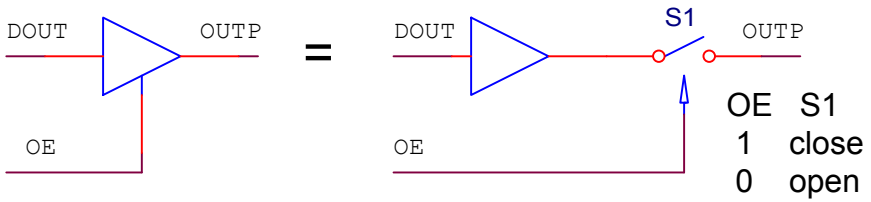
	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

U	Uninitialized
X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don't care

inout
out
'Z'
'X'

Tri-state and bidirectional ports

```
entity triout is
port (outp : out std_logic;
      dout : in  std_logic;
      oe   : in  std_logic);
end triout;
architecture b of triout is
begin
  outp <= dout when oe='1' else 'Z';
end;
```

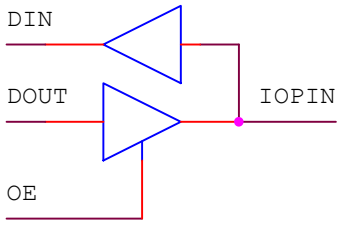


When using tri-stated buses driven by multiple drivers:

- Be sure that only one driver is active at the same time
- Insert turn-around cycles when changing the driver of the line with all drivers turned off
- Internal tri-state lines are typically not supported for FPGAs, some tools convert them to multiplexers

compare!

```
entity bidir is
port (iopin : inout std_logic;
      din   : out   std_logic;
      dout  : in    std_logic;
      oe    : in    std_logic);
end bidir;
...
with iopin select
din   <= '0' when '0'|'L',
         '1' when '1'|'H',
         'X' when others;
with oe select
iopin <= dout when '1',
         'Z'  when '0',
         'X'  when others;
end;
```



Buffer ports

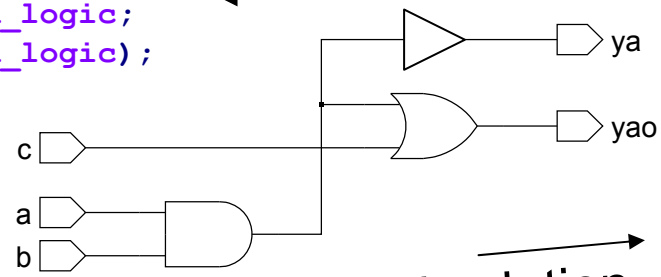
A signal of the mode **out** can not be used back as input in the entity. There are two solutions for this problem, either *using the* **buffer** *mode of the* **entity port**:

```
port (
    a     : in     std_logic;
    b     : in     std_logic;
    c     : in     std_logic;
    ya    : buffer std_logic;
    yao   : out    std_logic);
…
begin
    ya <= a and b;
    yao <= ya or c;
```

buffer is allowed to be read back but **out** not!

buffer ?

Not recommended!

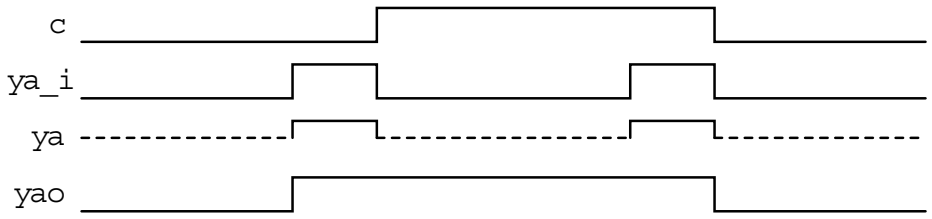


the standard solution

or with an additional **signal**:

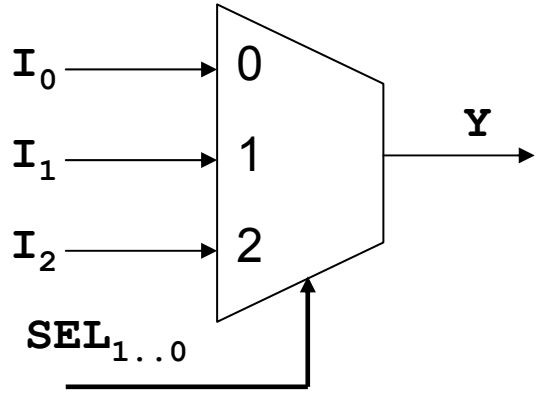
```
port (
  a     : in  std_logic;
  b     : in  std_logic;
  c     : in  std_logic;
  ya    : out std_logic;
  yao   : out std_logic);
…
signal ya_i : std_logic;
begin
  ya_i <= a and b;
  yao <= ya_i or c;
  ya <= ya_i;
```

use additional internal signal

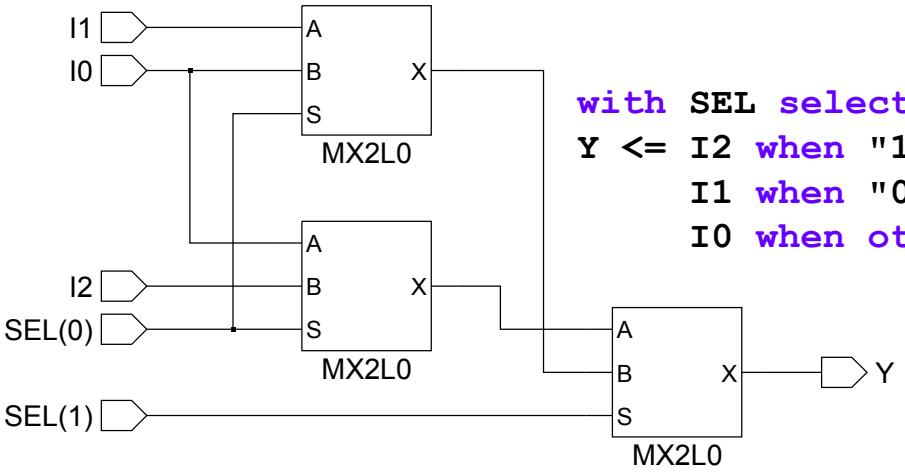


If the port **ya** is connected to another driver (driving '1' in this simulation), the internal signal **ya_i** is not affected!

Example for don‘t care '-' with a multiplexer 3:1



SEL	Y
0	I0
1	I1
2	I2
3	?

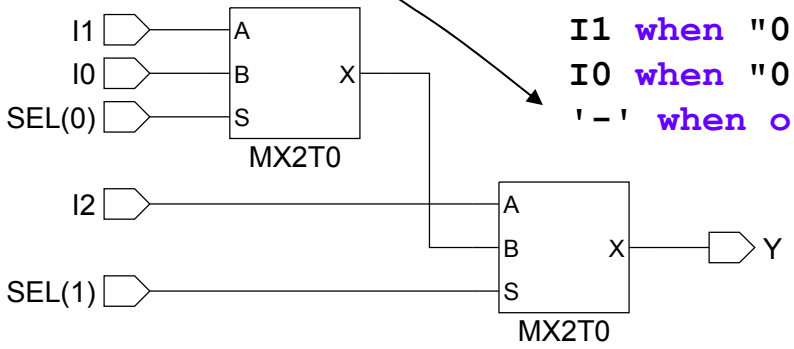



```
with SEL select
Y <= I2 when "10",
     I1 when "01",
     I0 when others;
```

What is the benefit of '-' ?

- automatic choice of the best value
- undefined output when the input SEL is unexpected in the functional simulation

```
with SEL select
Y <= I2 when "10",
     I1 when "01",
     I0 when "00",
     '-' when others;
```



More complex data types

- **Array**
  - predefined in `IEEE.STD_LOGIC_1164`, e.g. `std_logic_vector(3 downto 0);`

```
subtype reg_data is std_logic_vector(31 downto 0);
type mem_array is array(0 to 63) of reg_data;
```

the direction of the index

- **Enumerated**
  - Used mainly to describe state machines

```
type state_type is (idle, run, stop, finish);
```

- **Record**

```
type lvds_t is record a, b : std_logic; end record;
type ni_port_out_t is record
                        ctrl    : lvds_t;
                        clk     : lvds_t;
                        pretrig : lvds_t;
                      end record;
signal niP4 : ni_port_out_t;
signal myclk : lvds_t;
...
myclk.a  <=     clk;
myclk.b  <= not clk;
niP4.clk <= myclk;
```

records in record is allowed

access to the elements of the record like in C and Pascal

Generic

- **Generic** is a parameter of the **entity**, which is known at the compilation time
- The same **entity** can be instantiated many times with different values of the **generic**(s)
- Typically the generics are of type **integer**, **boolean** or **time**

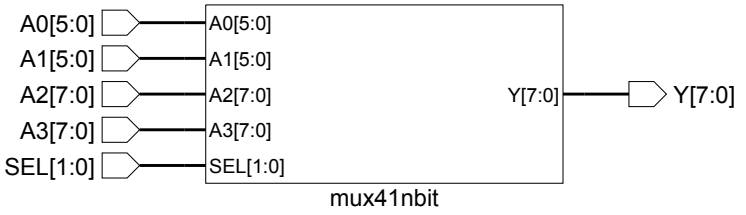
this is only the **default** value, used when no "**generic map**" found (next slide)

```
entity mux21nbit is

generic(N : Integer := 4; tdel : time := 2 ns);

port(A0, A1 : in  std_logic_vector(N-1 downto 0);
     SEL    : in  std_logic;
     Y      : out std_logic_vector(N-1 downto 0));
end entity;
```

The **default** value is the actual value used when the entity is the **top** of the design

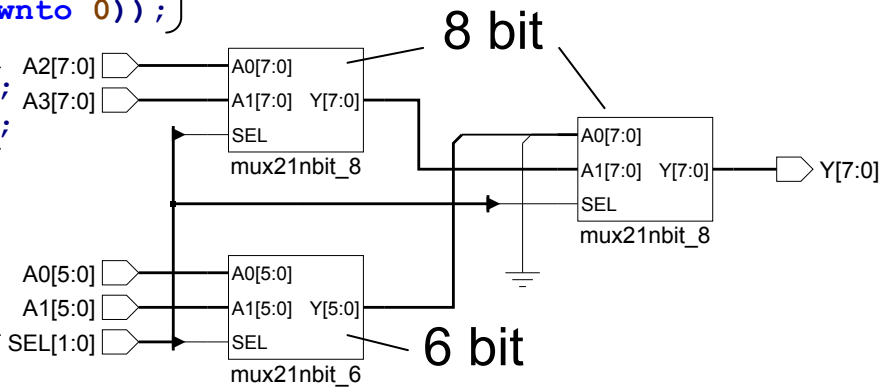
Generic map - example

```
entity mux41 is
port(A0, A1 : in  std_logic_vector(5 downto 0);
     A2, A3 : in  std_logic_vector(7 downto 0);
     SEL    : in  std_logic_vector(1 downto 0);
     Y      : out std_logic_vector(7 downto 0));
end mux41;
```




```
architecture struct of mux41 is
component mux21nbit is
generic(N : Integer := 4; tdel : time := 2 ns);
port(A0, A1 : in  std_logic_vector(N-1 downto 0);
     SEL    : in  std_logic;
     Y      : out std_logic_vector(N-1 downto 0));
end component;
signal y01 : std_logic_vector(5 downto 0);
signal y23 : std_logic_vector(7 downto 0);
begin
m01: mux21nbit
generic map(N => 6)
port map(A0  => A0, A1  => A1,
         SEL => SEL(0), Y => y01);
m23: mux21nbit
generic map(N => 8)
port map(A0  => A2, A1  => A3,
         SEL => SEL(0), Y => y23);
my: mux21nbit
generic map(N => 8)
port map(A0  => "00" & y01, A1  => y23, SEL => SEL(1), Y => Y);
```

Component declaration



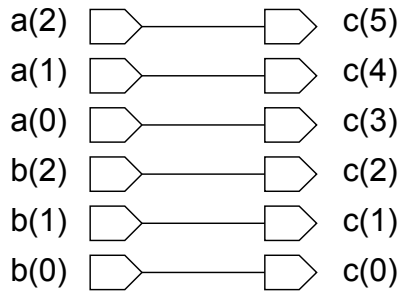
Overwrite the default value of the **generic** N

extend **y01**

Working with vectors - aggregates

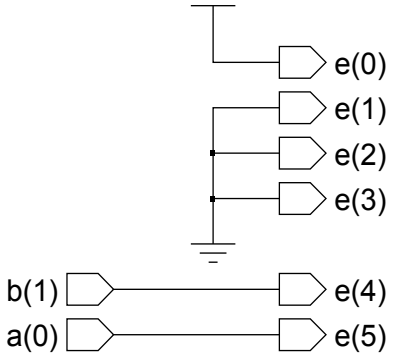
```
a : in  std_logic_vector(2 downto 0);
b : in  std_logic_vector(2 downto 0);
c : out std_logic_vector(5 downto 0);
d : out std_logic_vector(5 downto 0);
e : out std_logic_vector(5 downto 0);
f : out std_logic_vector(2 downto 0);
```

```
c <= a & b;
```

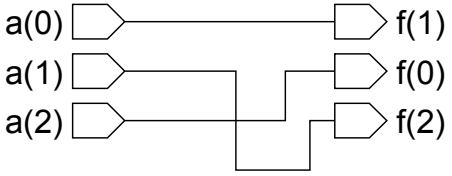


leftmost in **e** index

```
e <= (a(0), b(1), 0 => '1', others => '0');
```

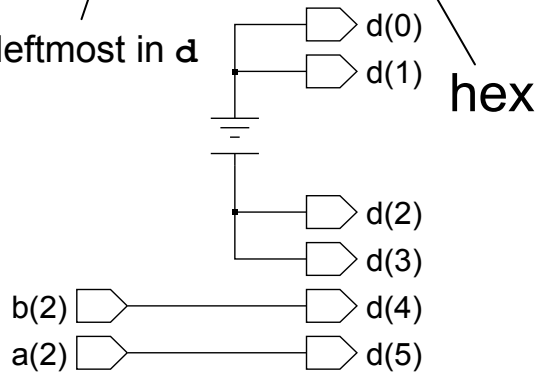



```
(f(0), f(2), f(1)) <= a;
```




```
d <= a(2) & b(2) & X"C";
```

leftmost in **d** hex



Working with vectors - multidimensional

- Two-dimensional vectors are typically not supported by the most software tools
- One simple way is to declare a new type as vector of vectors:

```
subtype my_data is std_logic_vector(15 downto 0);
type my_array is array(0 to 7) of my_data;

OR simply

type my_array is array(0 to 7) of
                        std_logic_vector(15 downto 0);
```

generic
generate
downto
to

Working with vectors – attributes

```
generic (N : Natural := 4);
port (
  a : in  std_logic_vector(2 to N+1);
  c : out std_logic_vector(N-1 downto 0);
  d : out std_logic_vector(N-1 downto 0) );
…
gn: for i in 0 to c'length-1 generate
        c(c'low + i) <= a(a'low + i);
    end generate;
    d <= a;
```

use **generic** to set the size

a(2) → c(0) → d(3)
a(3) → c(1) → d(2)
a(4) → c(2) → d(1)
a(5) → c(3) → d(0)

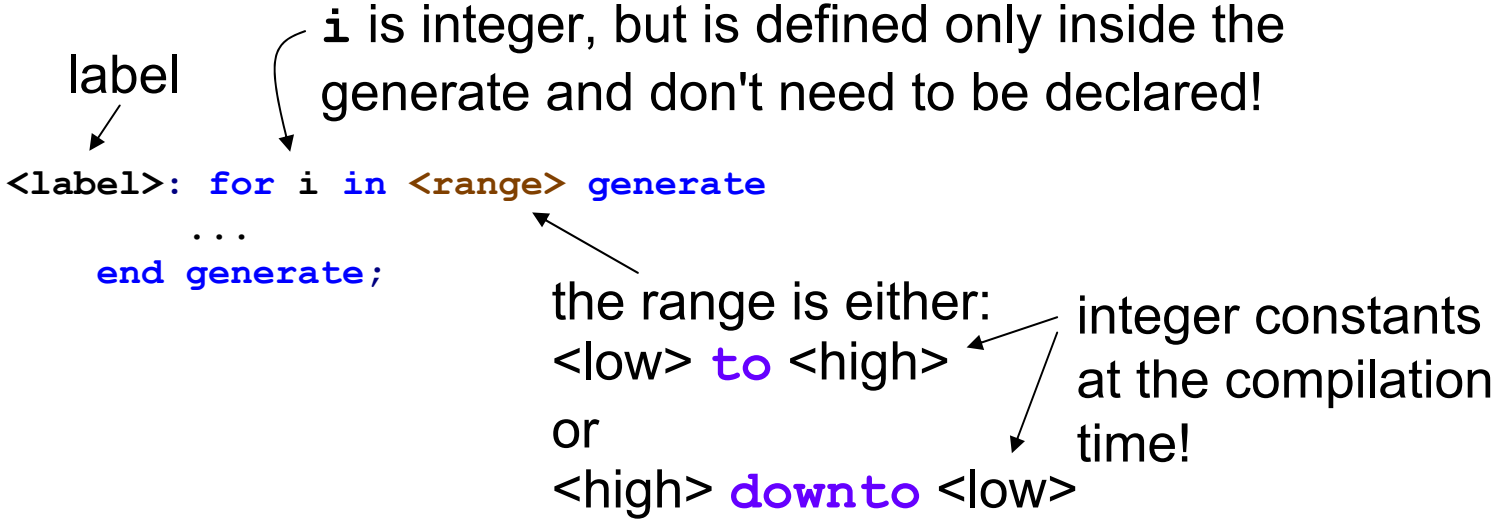
Be careful when mixing arrays with different directions (**to** ↔ **downto**)!

Use the predefined attributes to access different parameters of the array

a'length, c'length	N
a'low, a'left	2
a'high, a'right	N+1
c'low, c'right	0
c'left, c'high	N-1
a'ascending	TRUE
c'ascending	FALSE
a'range	2 to N+1
c'range	N-1 downto 0

for … generate

…can be used to repeat some assignments or instantiations



The restrictions are the same as for

```
if .. generate .. end generate
```

Used in several examples later

Conditional code – `if … generate`

In many programming languages (e.g. in C with **`#define`** … **`#ifdef`** … **`#endif`**) one can compile some parts of the source code depending on some condition. In VHDL one can use for the same purpose a **`boolean generic`** parameter and **`if`** … **`generate`** … **`end generate.`**

This is useful to configure the entity, instead of creating many variants as different entities.

This construct can not be used so freely as the **`#ifdef`** in C.

The usage is restricted to the **`architecture`** part of the **`entity`**, but outside processes. So one can include or exclude complete concurrent assignments or processes or entity instantiations.

```
...
generic(opipe : Boolean := true);
...
op1: if opipe generate
oreg: dff_array
generic map(N => datout'length)
port map(
     clk => clk,
     d   => dat_m,
     q   => datout);
end generate;
op0: if not opipe generate
     datout <= dat_m;
end generate;
```

label

label

`else generate` ⟷ `end generate;`

VHDL-2008

Output with/without register, depending on the **`generic`** parameter **`opipe`**

VHDL-2008: **`else generate`**, **`elsif generate`** , **`case <const> generate`**

Constants

- Can be declared in the architecture part together with the signals or in the packages

```
constant <constant_name> : <constant_type> := <value>;

constant Nbits  : Integer := 8;
constant Nwords : Integer := 6;

subtype my_word is std_logic_vector(Nbits-1 downto 0);

type my_array is array(0 to Nwords-1) of my_word;

constant all1 : my_word := (others => '1');
constant all0 : my_word := (others => '0');

constant arr_ini : my_array := (all0, all1, X"01", X"12",
                                others => "10101111");

constant Tco    : time := 5 ns;
constant Tsetup : time := 2 ns;
constant Thold  : time := 1 ns;
```

Put the sizes of the vectors in constants

hex

0x00	0
0xFF	
0x01	
0x12	
0xAF	
0xAF	5

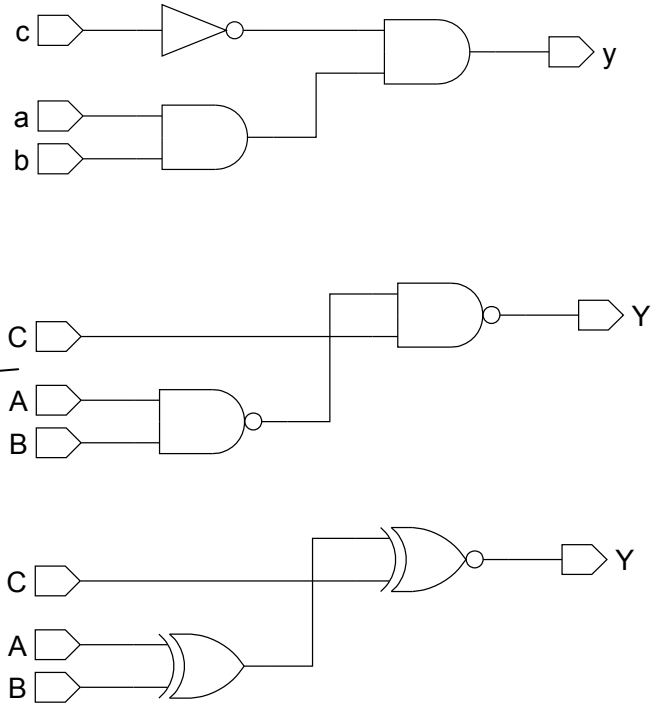
Constants of type **time**

Combinational circuits in VHDL(1)

or
nor
and
nand
xor
xnor
not

- Using simple assignments and Boolean expressions:

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



Use brackets when more than 2 operands and different operations!

Combinational circuits in VHDL(2)

`when … else`
`with … select`
`process`
`if … then …`

- Using Boolean equation + a condition:

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

conditional signal assignment

- Like a table, giving the output for all possible inputs:

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

selected signal assignment

- Using **`process`** – sequential code, like in a normal programming language

```
process(a, b, c)
   begin
      y <= '0';
      if a = '1' or b = '1' then y <= '1'; end if;
      if c = '1' then y <= '0'; end if;
   end process;
```

sensitivity list

The last value assigned will be stored in **`y`**

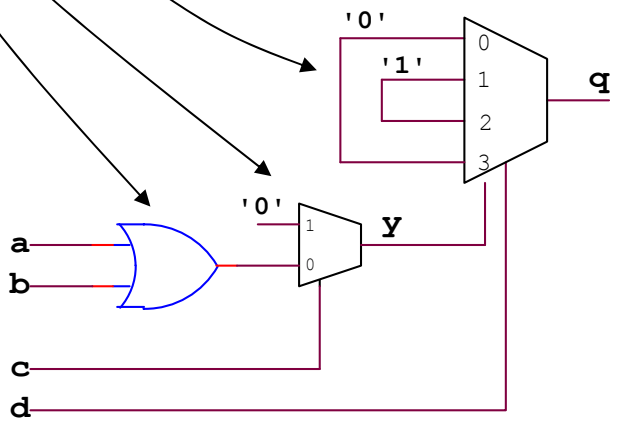
Caution: in all cases the synthesis generates logic working in parallel! Only the compiler/simulator "executes" sequentially the process to "understand" what do we require from it!

Concurrent assignments

- In the real hardware all gates work in parallel
- In VHDL we can use statements which are executed in parallel (concurrent in time), like

```
a_or_b <= a or b;
y <= a_or_b when c ='0' else '0';
with d & y select
   q <= '1' when "10"| "01",
        '0' when others;
```

The order of the assignments in the VHDL code here is NOT important!



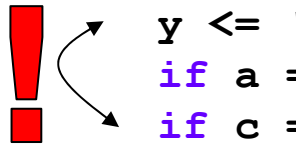
Normally we DON'T assign more than ONCE a value to a signal in the entity

A change in any signal propagates to all other signals depending on it, the assignments are continuously active!

known as DATAFLOW description

Sequential assignments for combinational logic

- We can use assignments which are executed sequentially in a **process**



```
process(a, b, c)
 begin
   y <= '0';
   if a = '1' or b = '1' then y <= '1'; end if;
   if c = '1' then y <= '0'; end if;
end process;
```

The **process** doesn't run continuously! It will be started only after change (**event**) on any signal in its **sensitivity list**

known as BEHAVIOUR description

After an event in **a**, **b** or **c**:
1) new value **'0'** is scheduled for **y**, but **y** still has its old value
2) if **a** or **b** are **'1'**, the new scheduled value of **y** is changed to **'1'**
3) if **c** is **'1'**, the scheduled value of **y** is changed to **'0'**
4) at the end of the **process** the scheduled value is assigned to **y**
5) as **y** is not on the sensitivity list, the **process** is suspended until the next event on **a**, **b** or **c**, and **y** preserves its new value

All this happens within the same simulation *time*!

The order of the assignments in the `process`

… is very important!

```
process(a, b, c)
  begin
1  y <= '0';
2  if a = '1' or b = '1' then y <= '1'; end if;
3  if c = '1' then y <= '0'; end if;
  end process;
```

```
process(all) -- VHDL-2008
```

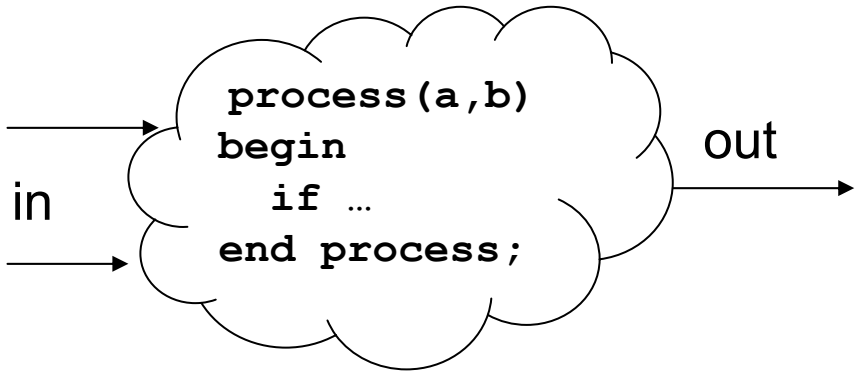
The 6 possible permutations in the **`process`** are equivalent to the following concurrent assignments

```
1-2-3
2-3-1, 3-2-1, 2-1-3
1-3-2, 3-1-2
```

```
y <= (a or b) and not c;
y <= '0';
y <= a or b;
```

Sequential assignments for combinational logic synthesis

- Think about the **process** as a way to describe the reaction of the combination circuit on events on its inputs

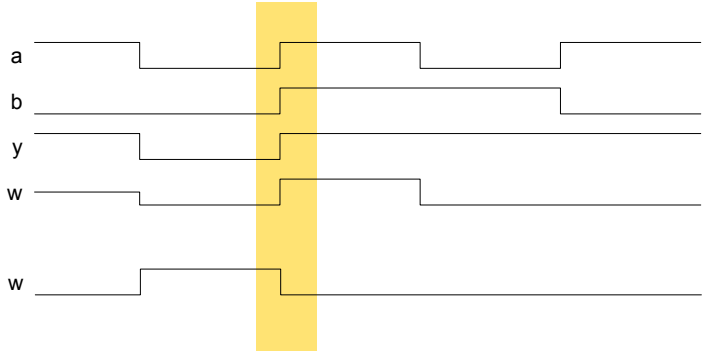


The compiler just calculates the truth table of the circuit in order to synthesise the logic!

The real hardware consists only of GATES!
The outputs of the combinational circuit should depend only on the **present** values of the inputs but **not** on the **past** → this implies that all outputs of the **process** **must** get some value assigned after **any** activation of the **process**!

Simulation deltas

```
process(a, b)
begin
    w <= not y;
    y <= a or b;
end process;
```




```
process(a, b, y)
begin
    w <= not y;
    y <= a or b;
end process;
```

ns	delta	a	b	y	w
0	0	1	0	U	U
0	1	1	0	1	U
20	1	0	0	1	U
20	2	0	0	0	0
40	1	1	1	0	0
40	2	1	1	1	1
60	1	0	1	1	1
60	2	0	1	1	0
80	1	1	0	1	0

```
process
begin
    a <= '1'; b <= '0';
    wait for 20 ns;
    a <= '0'; b <= '0';
    wait for 20 ns;
    a <= '1'; b <= '1';
    wait for 20 ns;
    a <= '0'; b <= '1';
    wait for 20 ns;
    a <= '1'; b <= '0';
    wait;
end process;
```

ns	delta	a	b	y	w
0	0	1	0	U	U
0	1	1	0	1	U
0	2	1	0	1	0
20	1	0	0	1	0
20	2	0	0	0	0
20	3	0	0	0	1
40	1	1	1	0	1
40	2	1	1	1	1
40	3	1	1	1	0
60	1	0	1	1	0
80	1	1	0	1	0

Gating of a vector

```
...
generic (N : Natural := 4);
port (
  a : in  std_logic_vector(N-1 downto 0);
  g : in  std_logic;
  y : out std_logic_vector(N-1 downto 0) );
end for_gen;
architecture ... of for_gen is
signal tmp : std_logic_vector(a'range);
begin
```

1
```
y <= a and g;
```

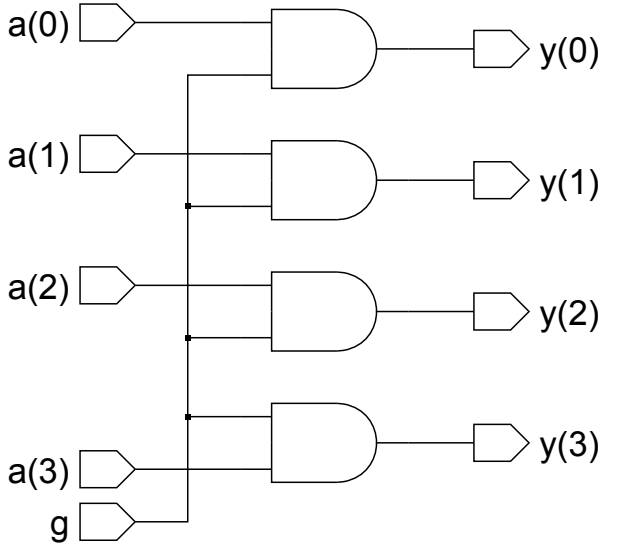
VHDL-2008

2
```
tmp <= (others => g);
y <= a and tmp;
```

attribute

many possible codes

and between two **std_logic_vectors** of the same size



3
```
y <= a when g='1' else (others => '0');
```

4
```
gn: for i in a'range generate
      y(i) <= a(i) and g;
    end generate;
```

label

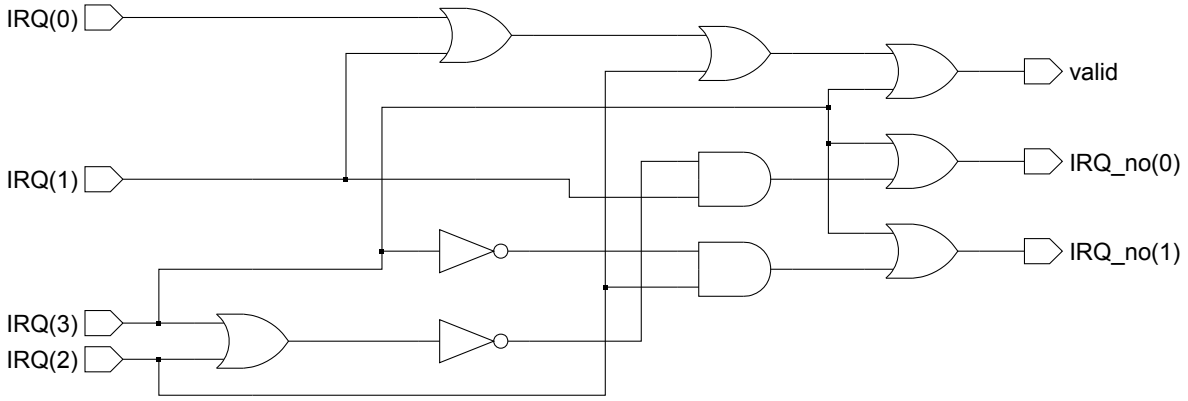
5
```
process(a,g)
begin
    if g='1' then y <= a;
    else y <= (others => '0'); end if;
end process;
```

Priority encoder

```
irq_no <= "11" when IRQ(3) = '1' else
          "10" when IRQ(2) = '1' else
          "01" when IRQ(1) = '1' else
          "00" when IRQ(0) = '1' else
          "--";
valid <= IRQ(0) or IRQ(1) or IRQ(2) or IRQ(3);
```

1

1-st method
(dataflow style)

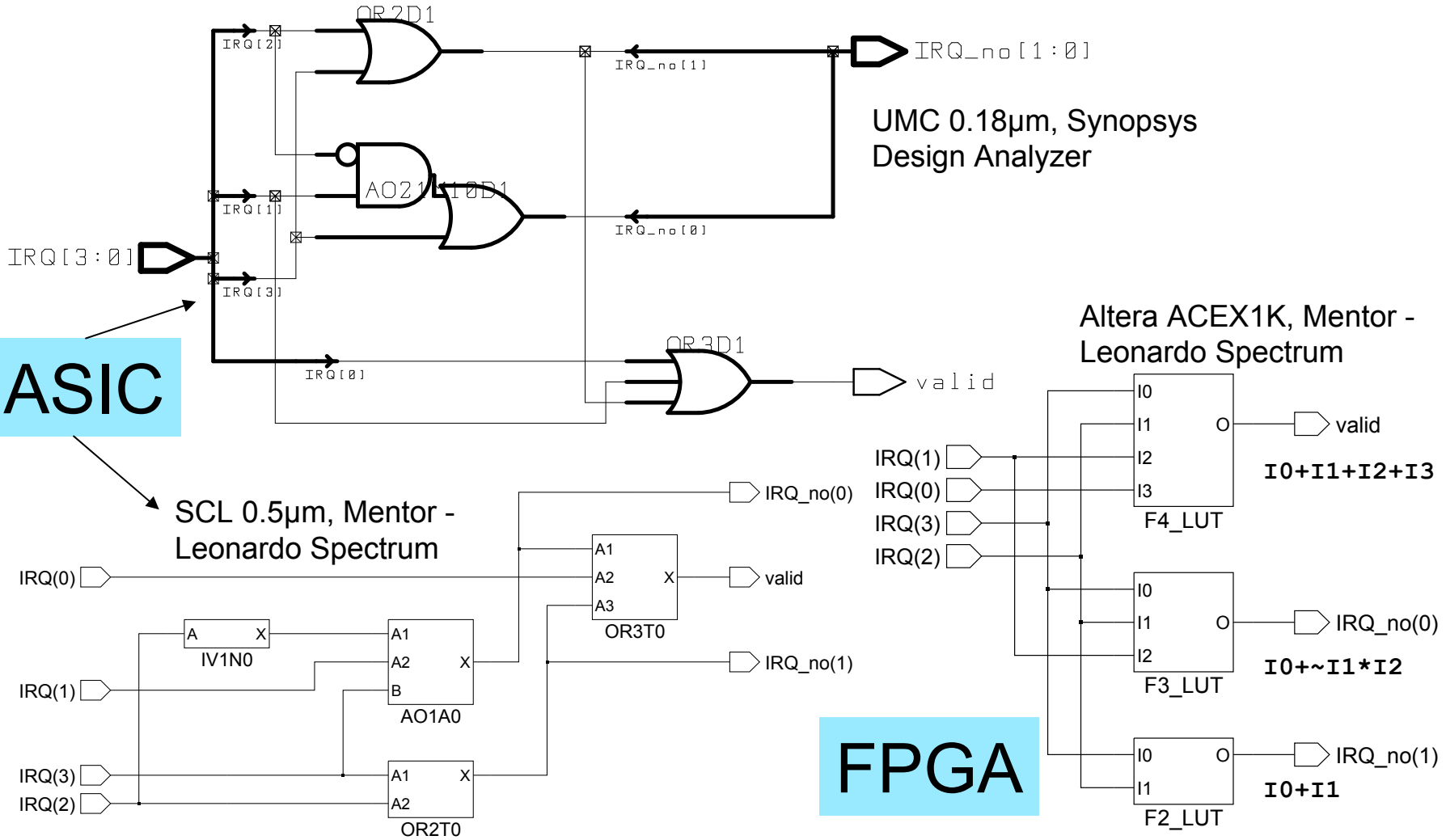



```
pri: process(IRQ)
     begin
      valid  <= '1';
      irq_no <= "--";
      if     (IRQ(3) = '1') then irq_no <= "11";
      elsif  (IRQ(2) = '1') then irq_no <= "10";
      elsif  (IRQ(1) = '1') then irq_no <= "01";
      elsif  (IRQ(0) = '1') then irq_no <= "00";
      else valid <= '0';
      end if;
     end process;
```

2

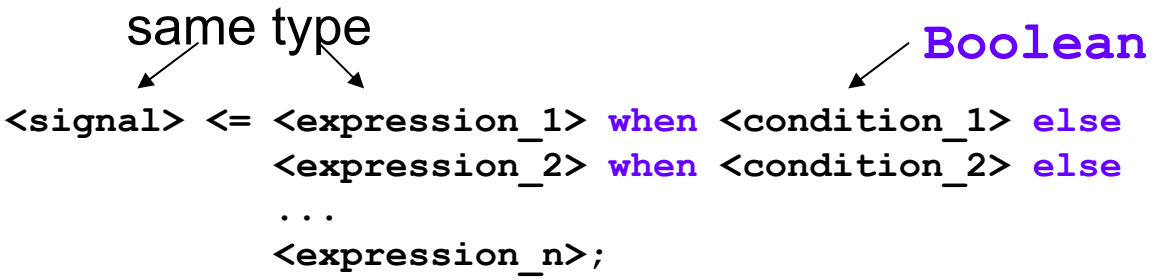
2-nd method,
using a **process**
(behaviour style)

Priority encoder – synthesis for 3 different technologies



Priority logic constructs

- Concurrent assignments – **when** … **else** …



- In a **process** this is equivalent to the following
  if … **then** … **elsif** ... **else**

put in the sensitivity list all signals read in the process!

```
process(...)
  begin
    if    <condition_1> then signal <= <expression_1>;
    elsif <condition_2> then signal <= <expression_2>;
    ...
    else  <signal> <= <expression_n>;
    end if;
  end process;
```

if-then-else can be used only in a **process**! It can be used for more than an assignment to a single signal

Multiplexer

Two equivalent descriptions, here because of the different length of **status, cfg1, cfg2, cfg3**, the **process** description is nicer

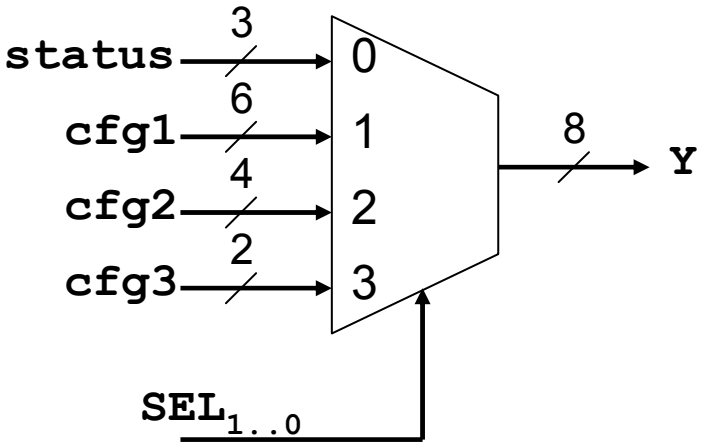
1

```
with SEL select
Y <= "00000"  & status when "00", -- 3 bits
     "00"     & cfg1   when "01", -- 6 bits
     "0000"   & cfg2   when "10", -- 4 bits
     "000000" & cfg3   when "11", -- 2 bits
     (others => 'X')   when others;
```

2

```
process(status, cfg1, cfg2, cfg3, SEL)
begin
    Y <= (others => '0');
    case SEL is
    when "00" => Y(status'range) <= status;
    when "01" => Y(cfg1'range  ) <= cfg1;
    when "10" => Y(cfg2'range  ) <= cfg2;
    when "11" => Y(cfg3'range  ) <= cfg3;
    when others => Y <= (others => 'X');
    end case;
end process;
```

1-st method (dataflow style)



2-nd method, using a **process** (behaviour style)

Demultiplexer

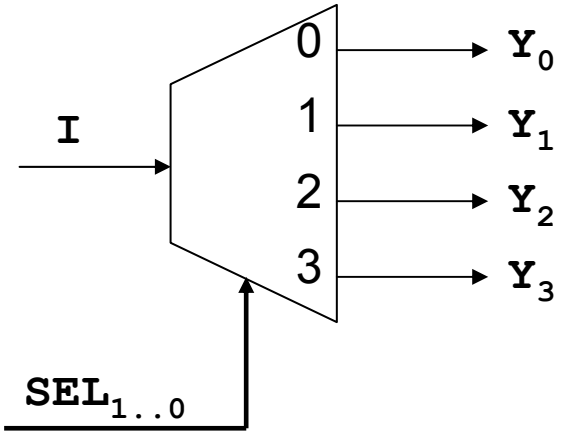
1

```
with SEL select
  Y <= I & "000"        when "11",
       '0' & I & "00"   when "10",
       "00" & I & '0'   when "01",
       "000" & I        when "00",
       (others => 'X') when others;
```

1-st and 2-nd method:
- not scalable
+ in case of undefined **SEL** input, all outputs are **'X'**

2

```
process(I, SEL)
begin
    Y <= (others => '0');
    case SEL is
    when "00" => Y(0) <= I;
    when "01" => Y(1) <= I;
    when "10" => Y(2) <= I;
    when "11" => Y(3) <= I;
    when others => Y <= (others => 'X');
    end case;
end process;
```



3

```
process(I, SEL)
begin
    Y <= (others => '0');
    Y(conv_integer(SEL)) <= I;
end process;
```

3-rd method
+ compact, scalable
- in case of undefined **SEL** input, **Y(0)** will be activated!

Non-priority selection logic

- Concurrent assignments – **with** … **select** … **when**

```
with <signal_s> select
     <signal> <= <expression_1> when <case_1>,
                 <expression_2> when <case_2>,
                 ...
                 <expression_n> when others;
```

constants of the same type as **<signal_s>**

- In a **process** this is equivalent to the following:

case can be used only in a **process**! It can be used for more than an assignment to a single signal

```
process(<signal_s>,...)
begin
  case <signal_s> is
    when <case_1> => <signal> <= <expression_1>;
    when <case_2> => <signal> <= <expression_2>;
    ...
    when others   => <signal> <= <expression_n>;
  end case;
end process;
```

In both cases **all** possible values of the **<signal_s>** must be specified (**<case_1>**, **<case_2>**…) exactly once, either explicitly or only some of them + the rest using **when others**!

Type conversions

- VHDL is strictly typed language
- Even mixing of **bit** and **std_logic** is not directly possible, but normally this is not necessary
- An array of **bit** (or **std_logic**) can not be directly used as integer number, but it is frequently used to hold integer numbers
- A conversion function (own or from some library) should be used

```
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
...
signal data11bits : std_logic_vector(10 downto 0);
signal tsti : Integer range 0 to 2047;
...
data11bits <= conv_std_logic_vector(2046,11); --> "11111111110"
tsti <= conv_integer(data11bits); --> 2046
-- The same constant can be loaded in data11bits with:
data11bits <= (0 => '0', others => '1');
```

```
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
```

integer value

number of bits

In order to convert correctly **std_logic_vector** to **Integer** use descending index in the **std_logic_vector** and 0 for the rightmost index!

The integer type

- The usage of **Integer** to hold signals is generally **not recommended** for the same reason as the usage of **bit** and **bit_vector**!

  When converting **std_logic_vector** to **Integer** all values different from '0' and '1' are lost, all undefined values are converted to the nice value '0'!

- Except in simulations, **always specify** the **range** of the **integer**, otherwise 32 bit logic will be synthesised!

```
subtype my_uint5bit  is Integer range 0 to 31;
subtype my_uint0to25 is Integer range 0 to 25;
subtype my_uint3to25 is Integer range 3 to 25;
subtype my_int_3to25  is Integer range  -3 to 25;
subtype my_int_32to25 is Integer range -32 to 25;
subtype my_int_33to25 is Integer range -33 to 25;
```

5 bit

6 bit

7 bit

- Note that an upper limit different from 2^N-1 and lower limit different from 0 and -2^N are useful **only for simulations** (to get error when out of range)
- For synthesis the limits are used **only** to get the number of the bits
- Lower limit > 0 is **ignored** for synthesis (is the same as 0)

Mathematical operations with **integer**s

```
signal byte1, byte2, byte3, byte4, byte5, byte6 : Integer range 0 to 16#FF#;
signal sint1, sint2, sint3, sint4, sint5, sint6 : Integer range -128 to 127;
signal word1, word2, word3 : Integer range 0 to 16#FFFF#;
signal intg : Integer range -2**15 to 2**15-1;
begin
  byte1 <=  20;
  sint1 <= -20;
  byte2 <= byte1 / 2;      -- =   10
  byte3 <= byte1 mod 3;    -- =    2
  byte4 <= byte1 rem 3;    -- =    2
  byte5 <= byte1 + byte2;  -- =   30
  byte6 <= byte1 - byte2;  -- =   10
  sint2 <= sint1 / 2;      -- =  -10
  sint3 <= sint1 mod 3;    -- =    1
  sint4 <= sint1 rem 3;    -- =   -2
  sint5 <= sint1 / 3;      -- =   -6
  sint6 <= sint1 + byte2;  -- =  -10
  word1 <= byte1**2;       -- =  400
  word2 <= sint1**2;       -- =  400
  word3 <= byte1*byte2;    -- =  200
  intg  <= byte1*sint1;    -- = -400
```

for positive numbers **mod = rem**

for negative numbers **mod ≠ rem**

Note that multiplication is **expensive** in hardware, division, mod and rem are generally **not** supported for synthesis (except for divider 2^N)

Mathematical operations with **std_logic_vectors**

- Using appropriate library it is possible to mix different types in mathematical operations and to apply mathematical operations (+ or -) to non-integer objects

```
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
...
signal data11b, data_inc : std_logic_vector(10 downto 0);
...
data_inc <= data11b + 1;
```

If `data11b` is `"11111111111" (2047)`, the result will be 0!

- The same is possible with the multiplication, but be careful, the multipliers are large! Use only for power of 2!
- For synthesis the division is supported only for power of 2, in this case it is just a shift (arithmetical or logical?)
- For every technology there are libraries with optimized modules for mathematical operations

The arithmetic packages(1)

- If you have only signed or only unsigned in the **entity** then you can use arithmetic operations with **std_logic_vector** by including the packages:

```
USE IEEE.STD_LOGIC_ARITH.all;
```

interpret **std_logic_vector** in arithmetic operations as **signed**

```
USE IEEE.STD_LOGIC_SIGNED.all;
```

only one of them!

```
USE IEEE.STD_LOGIC_UNSIGNED.all;
```

interpret **std_logic_vector** in arithmetic operations as **unsigned**

```
USE IEEE.Numeric_Std_Signed.all;
USE IEEE.Numeric_Std_Unsigned.all;
```

VHDL-2008

The arithmetic packages(2)

- The better way is to use only:

```
USE IEEE.Numeric_Std.all;
```

- Declare as **signed** or **unsigned** all **std_logic_vector**s in the design, used in arithmetic operations :

```
signal uns : unsigned(7 downto 0);
signal sgn : signed(7 downto 0);
```

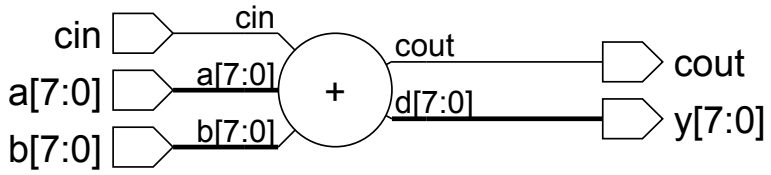
- Note that the definition of **signed** and **unsigned** is exactly the same as of the **std_logic_vector**, but they are different and cannot be freely mixed (see the adder example)

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;
```

There are two possibilities:

1. inferring (the synthesis tools recognize the adder and instantiate it from a library)
2. instantiating a ready adder component



Adder with `std_logic_arith`

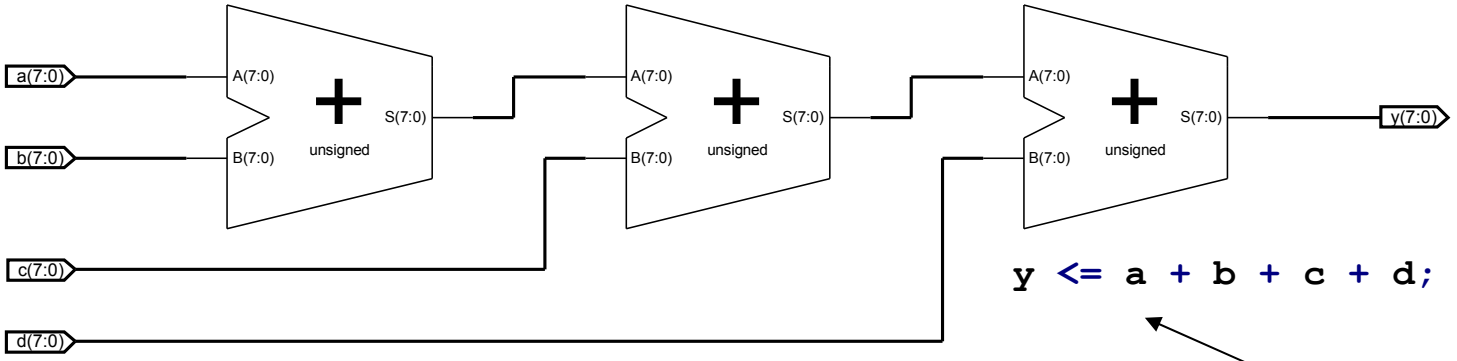
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.all;
entity adder is
generic (N : Natural := 8);
port (cin  : in  std_logic;
      a    : in  std_logic_vector(N-1 downto 0);
      b    : in  std_logic_vector(N-1 downto 0);
      cout : out std_logic;
      y    : out std_logic_vector(N-1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

Look at the source code of the packages **`std_logic_arith`**, **`std_logic_signed`** and **`std_logic_unsigned`** to learn more!

convert the **`std_logic_vector`** to **`unsigned`**

convert back from **`unsigned`** to **`std_logic_vector`**

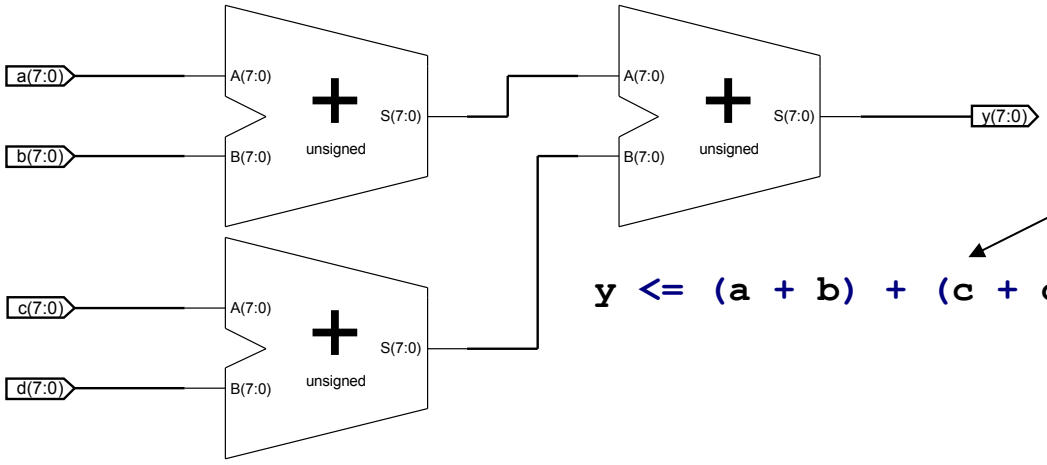
Adders and parentheses




```
y <= a + b + c + d;
```

```
a : in  std_logic_vector(7 downto 0);
b : in  std_logic_vector(7 downto 0);
c : in  std_logic_vector(7 downto 0);
d : in  std_logic_vector(7 downto 0);
y : out std_logic_vector(7 downto 0) );
```

This implementation doesn't have the best timing, if all inputs are valid simultaneously




```
y <= (a + b) + (c + d);
```

Using parentheses one can force the structure to another tree of adders with better timing

Package description

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

package <package_name> is

constant <constant_name> : <type> := <value>;

subtype <vec_name> is std_logic_vector(<range>);
subtype <int_name> is Integer range <range>;
type ...;
...
function <func_name>( <var_name> : <var_type>;
                ...) return <ret_type>;
...
end <package_name>;

package body <package_name> is

-- local constant, type, subtype declarations

-- function and procedure definitions

end <package_name>;
```

The package can be used for declarations common to the project

constant, **type**, **subtype**, **function**, **procedure** declarations

not visible outside

Function

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

package <package_name> is
...
function <func_name>( <var1_name> : <var1_type>;
                      ...
                      <varN_name> : <varN_type>)
          return <ret_type>;
...
end <package_name>;

package body <package_name> is
function <func_name>(
  <var_name> : <var_type;
     ...) return <ret_type> is
variable <var_name> :  <var_type;
begin
...
return <ret_value>;
end <func_name>;
...
end <package_name>;
```

The function can be used for some calculations

input variables

the **functions** can use only **variables** but no **signals**!

use here everything what is typical for a **process** like **case**, **if**; timing statements are not allowed here!

the return value, of the <ret_type>

package
variable
function
loop
high
length

Package and function example

```
library ieee;
use ieee.std_logic_1164.all;
package mypack is
constant ten : std_logic_vector(3 downto 0) := "1010";
subtype my_data is std_logic_vector(10 downto 0);
function se( src : std_logic_vector;
             ndst : Integer) return std_logic_vector;
end mypack;

package body mypack is

function se( src : std_logic_vector;
             ndst : Integer) return
std_logic_vector is
variable tmp :
std_logic_vector(ndst-1 downto 0);
begin
  if src'length <= ndst then
    for i in src'range loop
        tmp(i):= src(i);
    end loop;
  for i in src'high+1 to ndst-1 loop
        tmp(i):= src(src'high);
    end loop;
  else
    for i in 0 to ndst-1 loop
        tmp(i):= src(i);
    end loop;
  end if;
  return tmp;
end se;
end mypack;
```

constant, **type**, **function** definitions

attribute

attribute

note the different symbol:
:= for variables
<= for signals

Operator overloading(1)

- VHDL is strictly typed language, so a function with the same name can be defined for different types of the arguments and of the result, the compiler automatically selects the proper one
- The operators like +, - can be overloaded with functions, here is a part of the **`std_logic_arith`** package:

```
…
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
…
```

Operator overloading(2)

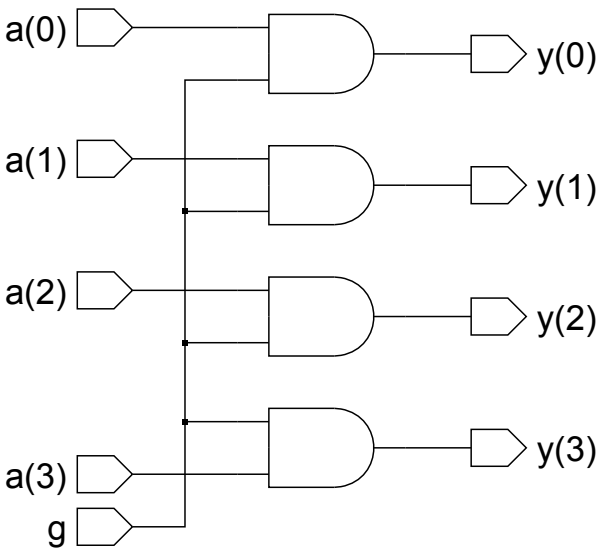
```
…
entity gate_vect is
generic (N : Natural := 4);
port (a : in  std_logic_vector(N-1 downto 0);
      g : in  std_logic;
      y : out std_logic_vector(N-1 downto 0) );
end gate_vect;
architecture ...
function "and" (L: std_logic; R: std_logic_vector)
    return std_logic_vector is
variable tmp : std_logic_vector(R'range);
begin
    for i in R'range loop
        tmp(i) := R(i) and L;
    end loop;
    return tmp;
end;
function "and" (L: std_logic_vector; R: std_logic)
     return std_logic_vector is
begin
    return R and L;
end;
begin
    y <= g and a;
    y <= a and g;
...
```

In VHDL-2008 already there

unconstraint

left and right operand

both possible



Such overloading functions can be put in a package, but be careful, your code will be not very standard - difficult to understand by other designers!

`Library` and `package` declarations

```
library library_name;

use library_name.package_name.all;

use library_name.otherpackage_name.all;

library work;

use work.my_package.all;
```

declare the `library` name

one `library` can contain many different `package`s

typically one uses the whole `package`, which stays in `all`

the `library` work is by default always accessible, therefore this declaration is not necessary

typically here come the own `package`s with `constant`s, `type`s, `function`s and `procedure`s used in the whole project

Package usage example

The **library** IEEE and the package STD_LOGIC_1164 are typically always used

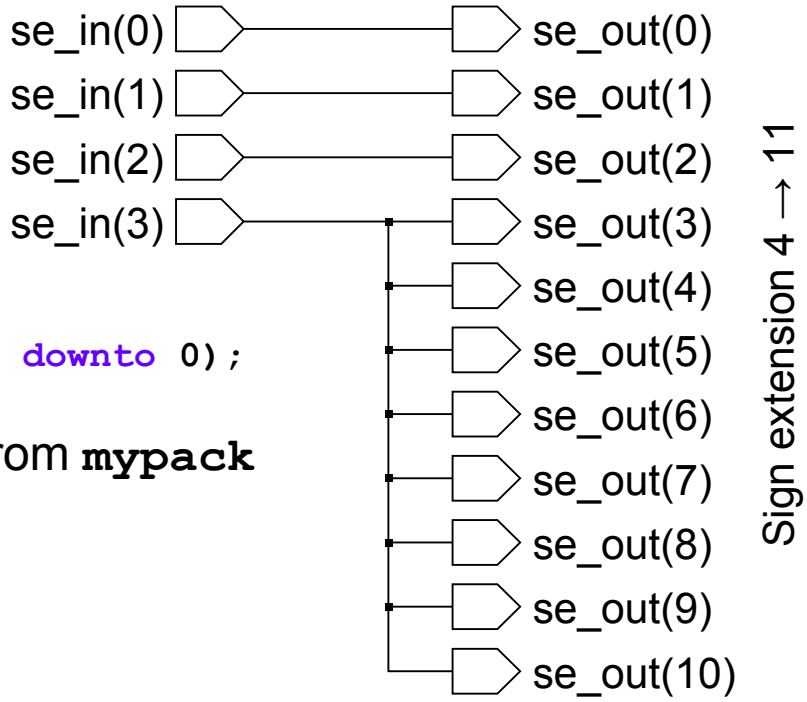
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

use work.mypack.all;

entity test_pack is
generic (Ni : Integer := 4);
port(se_in  : in  std_logic_vector(Ni-1 downto 0);
     se_out : out my_data );
end test_pack;

architecture a of test_pack is
begin
    se_out <= se(se_in, se_out'length);
end;
```

package

type from **mypack**

function from **mypack**

attribute



Variables

- Variable in a function
  - stores intermediate results of the calculations, exactly like in a normal computer program
- Variable in a process:
  - local for the process
  - holds its value between the activations of the process
  - the value assigned to it can be immediately used, like in a normal computer program

Example: `signal` vs. `variable`

```
signal sa : std_logic := 'X';
signal si : Integer range -7 to 7;
begin
process
variable va : std_logic;
variable vi : Integer range -7 to 7;
begin                    --   sa        si        va      vi
                         -- act sch   act sch
                         -- 'X'       -7        'U'     -7
  sa <= '1';             -- 'X' '1'    -7        'U'     -7
  va := sa;              -- 'X' '1'    -7        'X'     -7
  si <= 0;               -- 'X' '1'    -7   0    'X'     -7
  vi := 0;               -- 'X' '1'    -7   0    'X'      0
  wait for 10 ns;        -- '1'        0         'X'      0
  va := '0';             -- '1'        0         '0'      0
  si <= si + 1;          -- '1'        0    1    '0'      0
  vi := vi + 1;          -- '1'        0    1    '0'      1
  sa <= va;              -- '1' '0'    0    1    '0'      1
  si <= si + 1;          -- '1' '0'    0    1    '0'      1
  vi := vi + 1;          -- '1' '0'    0    1    '0'      2
  va := 'H';             -- '1' '0'    0    1    'H'      2
  si <= si + 1;          -- '1' '0'    0    1    'H'      2
  vi := vi + 1;          -- '1' '0'    0    1    'H'      3
  sa <= va;              -- '1' 'H'    0    1    'H'      3
  si <= si + 1;          -- '1' 'H'    0    1    'H'      3
  vi := vi + 1;          -- '1' 'H'    0    1    'H'      4
  wait for 10 ns;        -- 'H'        1         'H'      4
```

act – actual value
sch – scheduled value

for the signals only!

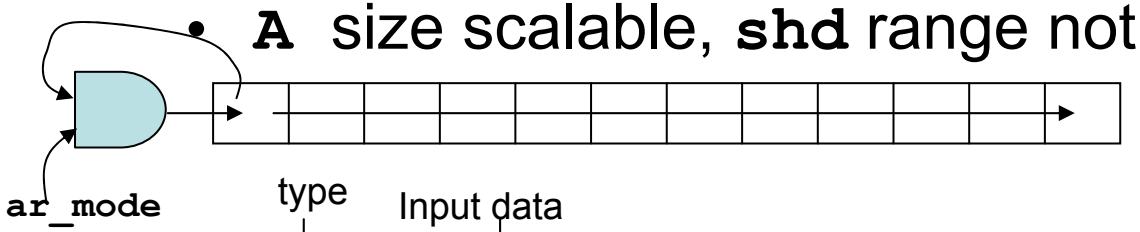
the state after the execution of each line

the changes are marked with ●

for a `signal`: each new assignment overwrites the scheduled value, `wait` or `end process` updates the actual value

Barrel shifter right

- Used to divide by power of 2
- Depending on the data type, the most significant bit filled with 0 or preserved
- First method, very simple description
  - **A** size scalable, **shd** range not scalable




```
m <= ar_mode and A(A'high); -- the new MSB
with shd select
  Y <= A                          when "000",
       m & A(A'high downto 1)     when "001",
       m & m & A(A'high downto 2) when "010",
       m & m & m & A(A'high downto 3) when "011",
       m & m & m & m & A(A'high downto 4) when "100",
       m & m & m & m & m & A(A'high downto 5) when "101",
       m & m & m & m & m & m & A(A'high downto 6) when "110",
       m & m & m & m & m & m & m & A(A'high downto 7) when "111",
       (others => 'X') when others;
```

output

shift **shd**

Area
1544 µm² (UMC 0.18 µm)
50 LUT4 (Altera ACEX 1k)
47 LUT4 (Xilinx Spartan 2E)

Barrel shifter – other methods

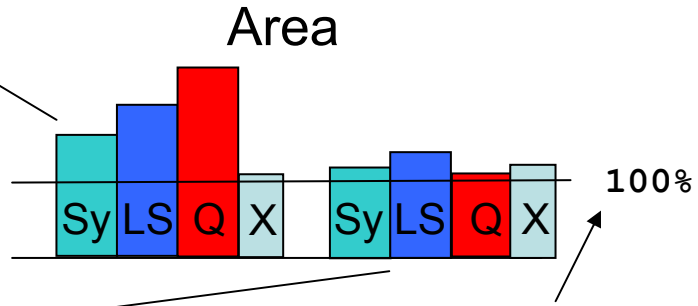
- Two attempts to write it more flexible

```
process(A, shd, m)
variable s : Integer;
variable tmp : std_logic_vector(Y'range);
begin
    s := conv_integer(shd);
    tmp := A;
    for i in 0 to 7 loop
      if (i < s) then tmp := m & tmp(A'high downto 1); end if;
    end loop;
    Y <= tmp;
end process;
```

t_D +35%

The variables are updated immediately!

copy the result to a signal, the **variable**s are visible **only** locally in the **process**!



= the simple solution from the previous page

```
process(A, shd, m)
variable s : Integer;
begin
    s := conv_integer(shd);
    for i in Y'range loop
        if (i > (Y'high - s)) then Y(i) <= m;
        else Y(i) <= A(i+s); end if;
    end loop;
end process;
```

t_D +15%

Sy	– Synposys, UMC 0.18µm
LS	– Leonardo Spectrum, SLC 0.5 µm
Q	– Quartus, Altera ACEX 1K
X	– ISE, Xilinx

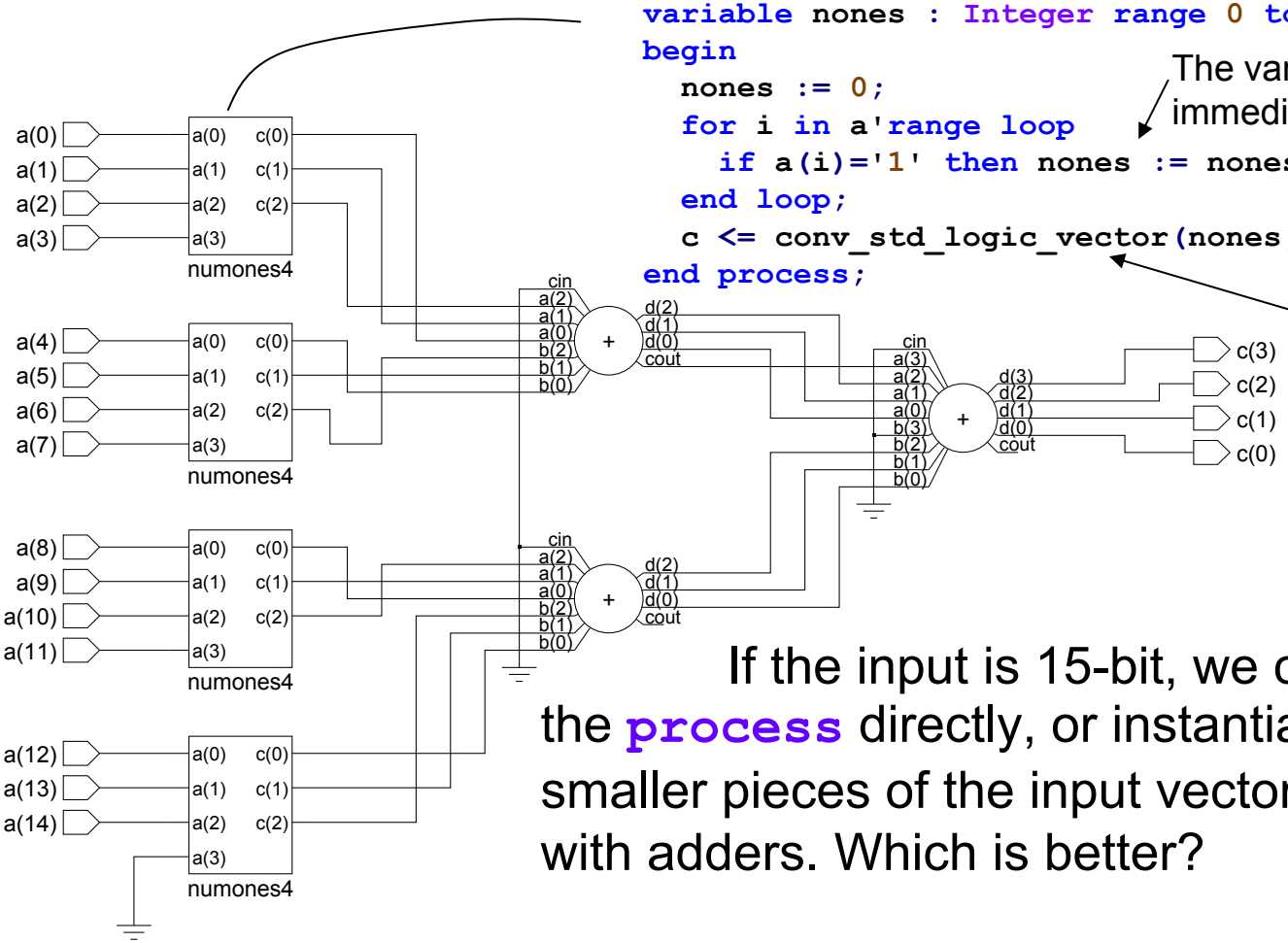
Counting ones

Let's try to count the number of `'1'` in a N-bit vector. The process shown below can do this:

```
process(a)
variable nones : Integer range 0 to a'length;
begin
  nones := 0;
  for i in a'range loop
    if a(i)='1' then nones := nones + 1; end if;
  end loop;
  c <= conv_std_logic_vector(nones, c'length);
end process;
```

The variables are updated immediately!

copy the result to a signal



If the input is 15-bit, we can either use the **`process`** directly, or instantiate it 4 times for smaller pieces of the input vector and do the rest with adders. Which is better?

Recommendations

- Forget about how you write C programs!
- Here, the compiler synthesizes logic according to your code, working in parallel
- Write as simple as possible, without using complex constructs
- Do not write any line of code, without having some imagination of what kind of hardware will come out
- It is nice to write more universal and flexible, but not at the cost of design size and speed

The choice is yours!

- Never make anything simple and efficient when a way can be found to make it complex and wonderful (Murphy)

OR

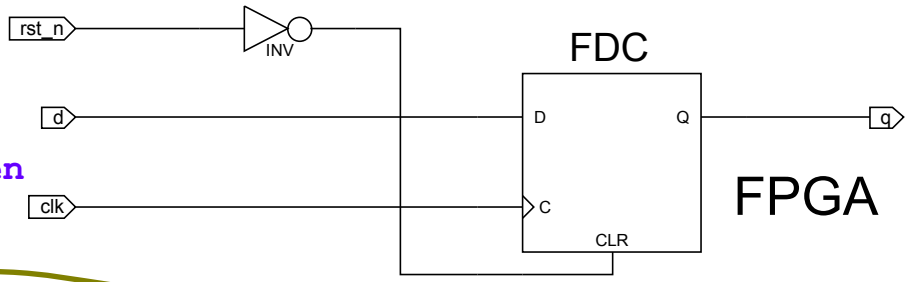
- Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away (Antoine de Saint-Exupéry)

Sequential circuits in VHDL - DFF

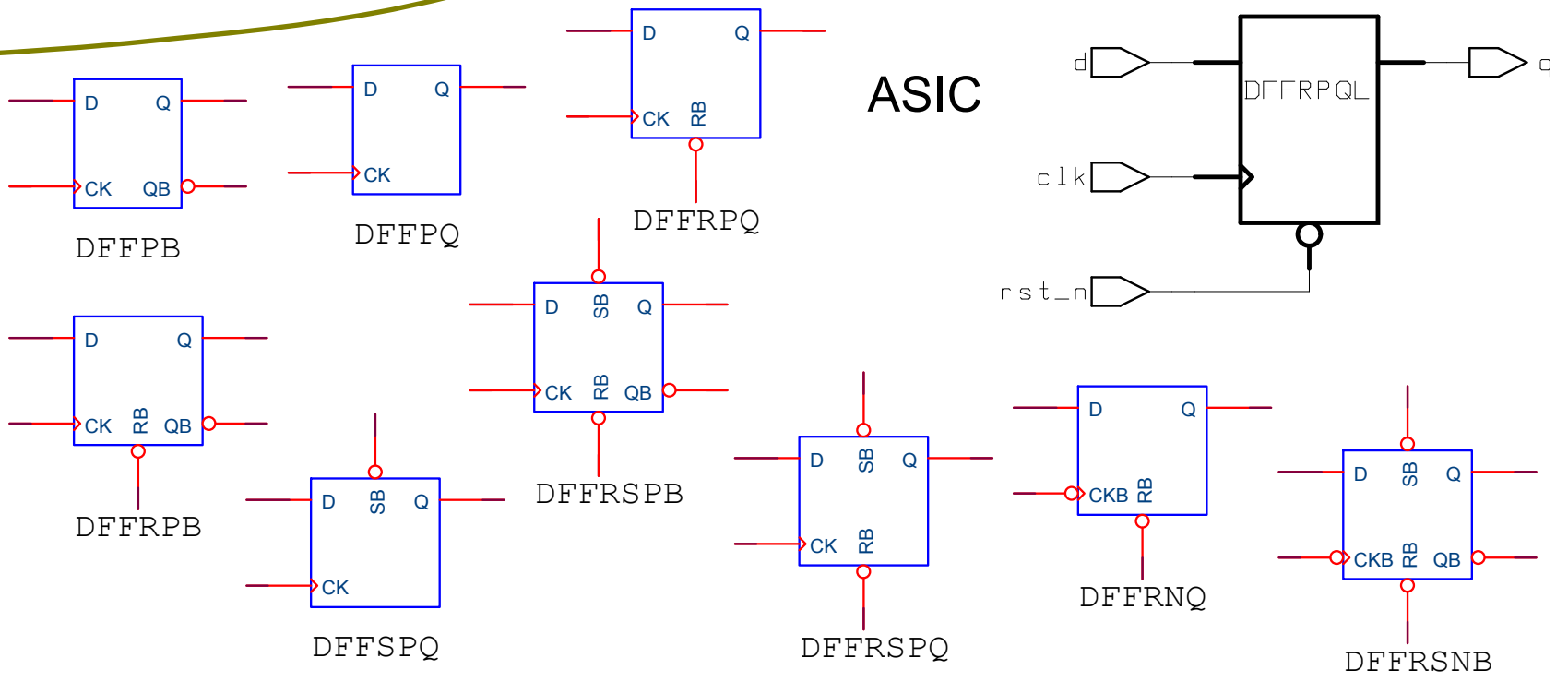
```
process(clk, rst_n)
begin
    if rst_n = '0' then q <= '0';
    elsif clk'event and clk='1' then
        q  <= d;
    end if;
end process;
```

attribute

0



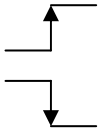
ASIC



Signal attributes

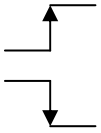
- '**event** – true if an event occurred in the last simulation cycle
- For the very frequent conditions of edge sensitive flip-flops one can use the functions:

```
clk'event and clk='1'  ➔   rising_edge(clk)
clk'event and clk='0'  ➔   falling_edge(clk)
```



- '**last_value** – actually the functions above use this attribute to exclude all clock transitions different from '0' → '1' (like 'x' → '1') or '1' → '0' (like 'U' → '0'):

```
clk'event and clk='1' and clk'last_value='0'
clk'event and clk='0' and clk'last_value='1'
```



- more signal attributes are available for simulation

Edge triggered `process` with one asynchronous set or reset

Unlike in a combinational process, list **here** only the ONE clock signal and the ONE reset signal, and no other signals read in the process!

```
process(<clock_signal>, <set_reset_signal>)
begin
    if <set_reset_signal> = <'0'|'1'> then
        ...
        asynchronous assignments of constants!
        ...
    elsif xxx_edge(<clock_signal>) then
        ...
        synchronous assignments
        ...
    end if;
end process;
```

Assigning values from other signals here is **not recommended** (see later **DFF with asynchronous load**)**!**

All output signals of the **`process`** must be here (see later **Shift register with bad reset**)

only one **`elsif`** without further **`else`** checking for edge

Unlike in the combinational case, the output signals of the **`process`** don't need to get values assigned in all cases – the circuit **has memory!**

Edge triggered `process` with synchronous set or reset

Unlike in a combinational process, list **here** only the ONE clock signal and **no other** signals read in the process!

only one `if` checking for edge

```
process(<clock_signal>)
begin
   if xxx_edge(<clock_signal>) then
      if <set_reset_signal> = <'0'|'1'> then
        ...
        set/reset assignments
        ...
      else
        ...
        synchronous assignments
        ...
      end if;
   end if;
end process;
```

All output signals must be here (see later **Shift register with bad reset**)

Unlike in the combinational process, the output signals do not need to get a value assigned in all cases – the circuit **has memory!**

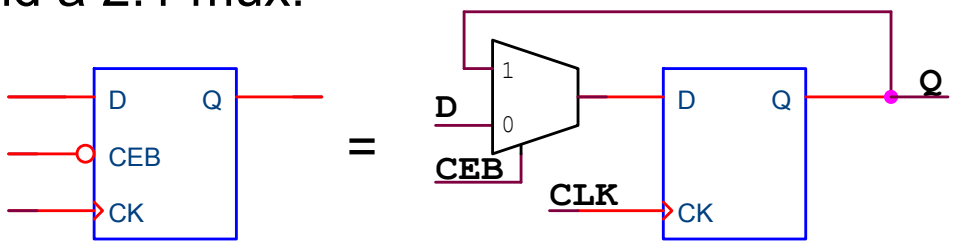
no `else`

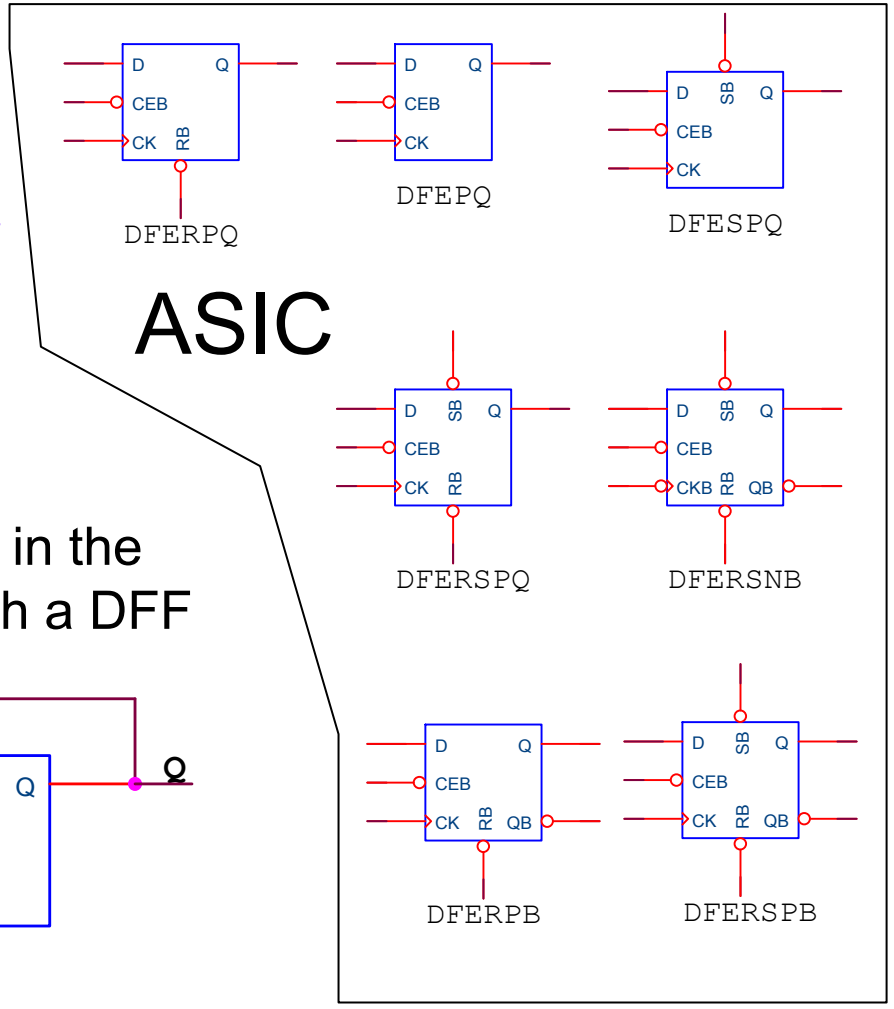
Sequential circuits in VHDL — DFFE(1)

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

'0'

If DFF with enable is not available in the technology, it can be emulated with a DFF and a 2:1 mux:





DFF with enable(2)

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

Altera Quartus 8.1
Error (10397): VHDL Event Expression error at my_dffe.vhd(57): can't form clock edge from S'EVENT by combining it with an expression that depends on a signal besides S

Leonardo Spectrum 2006b
…my_dffe.vhd",line 57: Error, clock expression should contain only one signal.

ISE 9.2 – no errors/warnings

! Not recommended descriptions !

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

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

If missing

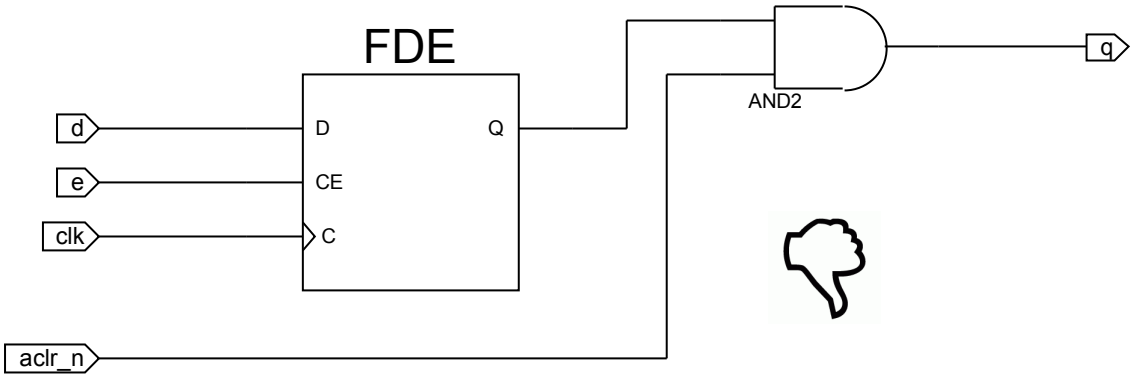
Altera Quartus 8.1:
Warning (10492): VHDL Process Statement warning at my_dffe.vhd(34): signal "en_n" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Leonardo Spectrum:
"my_dffe.vhd",line 45: Warning, en_n should be declared on the sensitivity list of the process.

Xilinx ISE 9.2:
WARNING:Xst:819 - "my_dffe.vhd" line 30: The following signals are missing in the process sensitivity list:

A strange DFFE with "asynchronous clear"

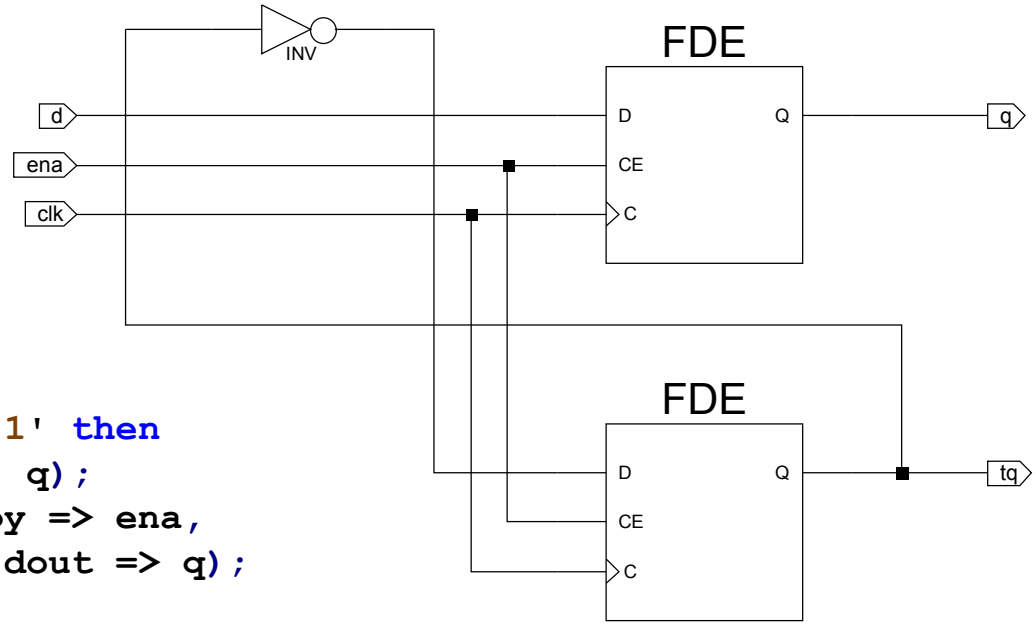
```
process(clk, aclr_n)
begin
 if aclr_n = '0' then
  q <= '0';
 else
  q <= q_i;
 end if;
 if clk'event and clk='1' then
  if e = '1' then
   q_i <= d;
  else
   q_i <= q_i;
  end if;
 end if;
end process;
```



DFFE and TFF with procedure and process

```
procedure myreg(
    signal din  : in    std_logic;
    signal cpy  : in    std_logic;
    signal tgl  : inout std_logic;
    signal dout : out   std_logic) is
begin
    if cpy = '1' then
        dout <= din;
        tgl  <= not tgl;
    end if;
end;
signal tqi : std_logic;
begin
process(clk)
begin
    if clk'event and clk='1' then
        myreg(d, ena, tqi, q);
or      myreg(din => d, cpy => ena,
              tgl => tqi, dout => q);
    end if;
end process;
tq <= tqi;
```

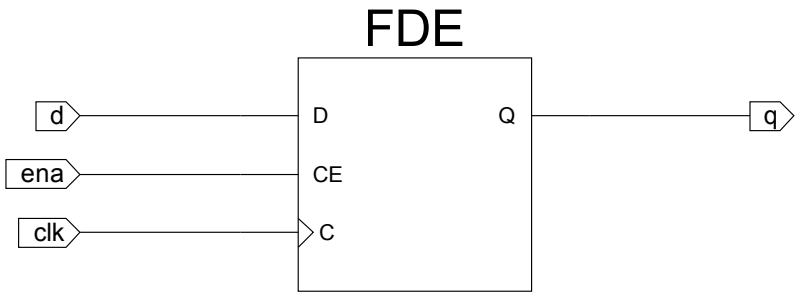
similar to the ports of an entity



call the procedure, like instantiation of a component

DFFE and MUX with procedure

```
procedure mydff(
    signal din  : in  std_logic;
    signal clk  : in  std_logic;
    signal ena  : in  std_logic;
    signal q    : out std_logic) is
begin
    if rising_edge(clk) then
        if ena='1' then q <= din; end if;
    end if;
end;
...
    mydff(din => d, ena => ena, clk => clk, q => q);
```

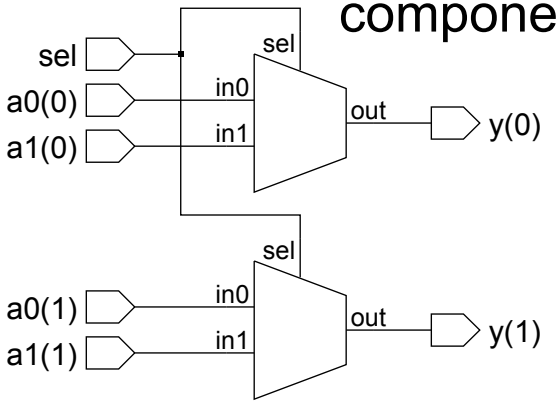


It is much better to put such code in entities and to instantiate as components!

---

unconstraint

```
procedure my21mux(
 signal a0  : in  std_logic_vector;
 signal a1  : in  std_logic_vector;
 signal sel : in  std_logic;
 signal y   : out std_logic_vector) is
begin
    if sel = '1' then y <= a1;
                 else y <= a0; end if;
end;
```

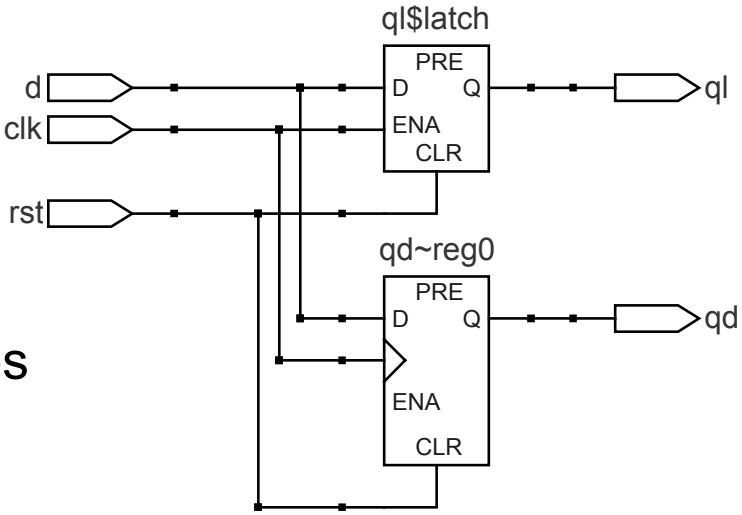


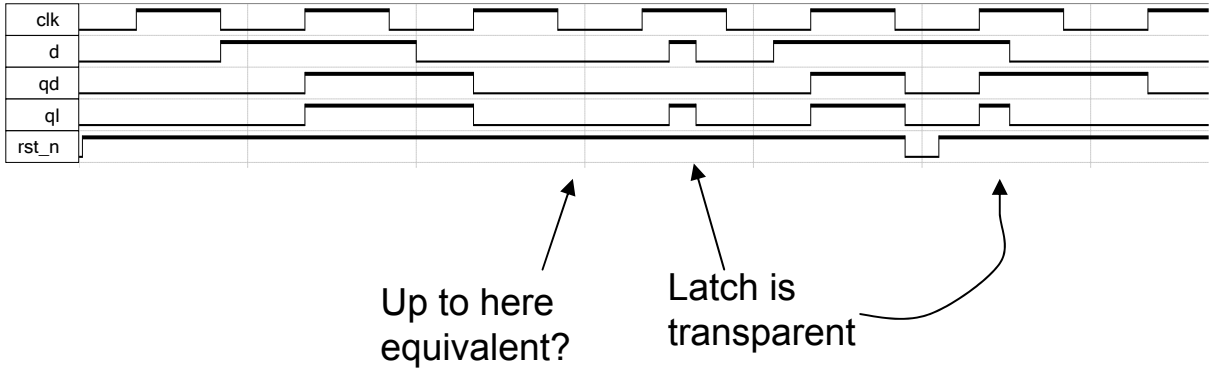
DFF or Latch (1) ?

```
process(clk, rst)
begin
    if rst = '1' then
        qd <= '0';
    elsif clk'event and clk='1' then
        qd <= d;
    end if;
end process;
```

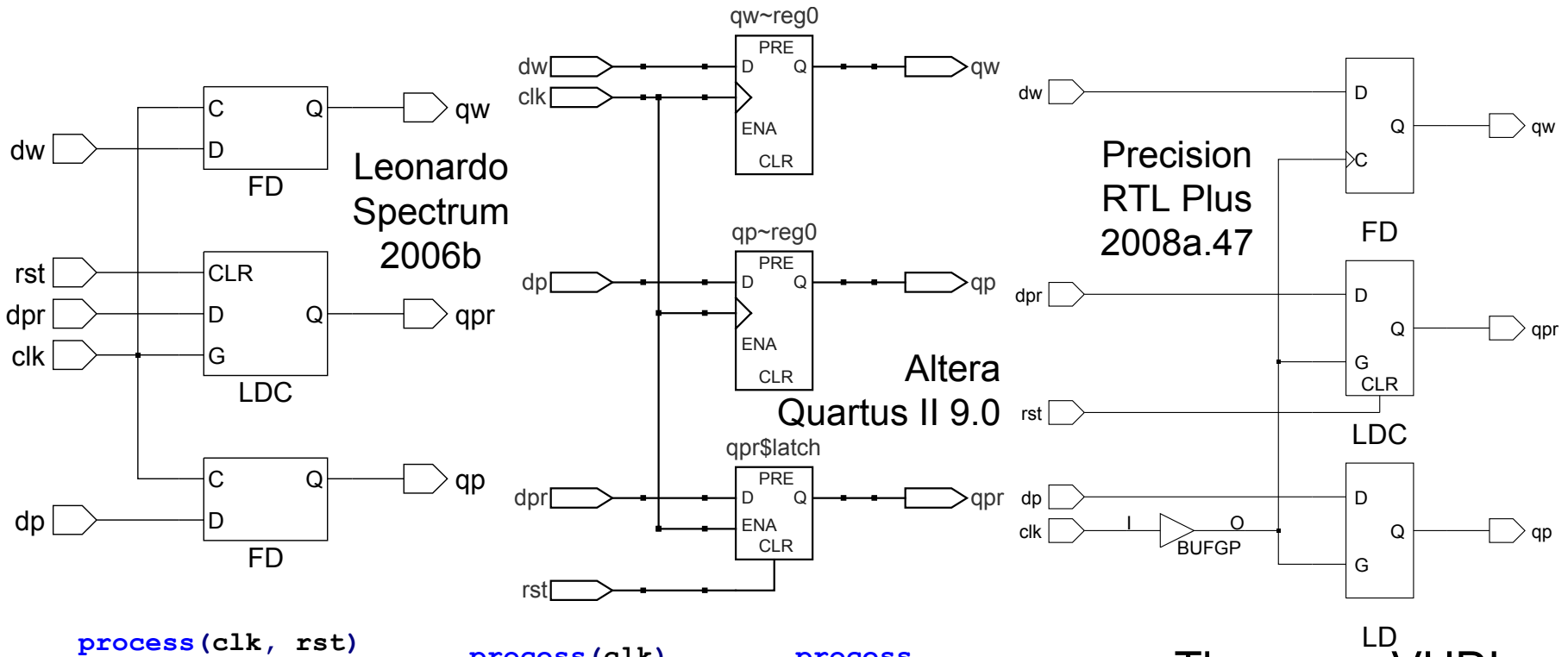
the two differences

```
process(clk, d, rst)
begin
  if rst = '1' then
     ql <= '0';
  elsif clk='1' then
     ql <= d;
  end if;
end process;
```





DFF or Latch (2) ?




```
process(clk, rst)
begin
  if rst = '1' then
    qpr <= '0';
  elsif clk='1' then
    qpr <= dpr;
 end if;
end process;
```

LATCH

```
process(clk)
begin
  if clk='1' then
    qp <= dp;
  end if;
end process;
```

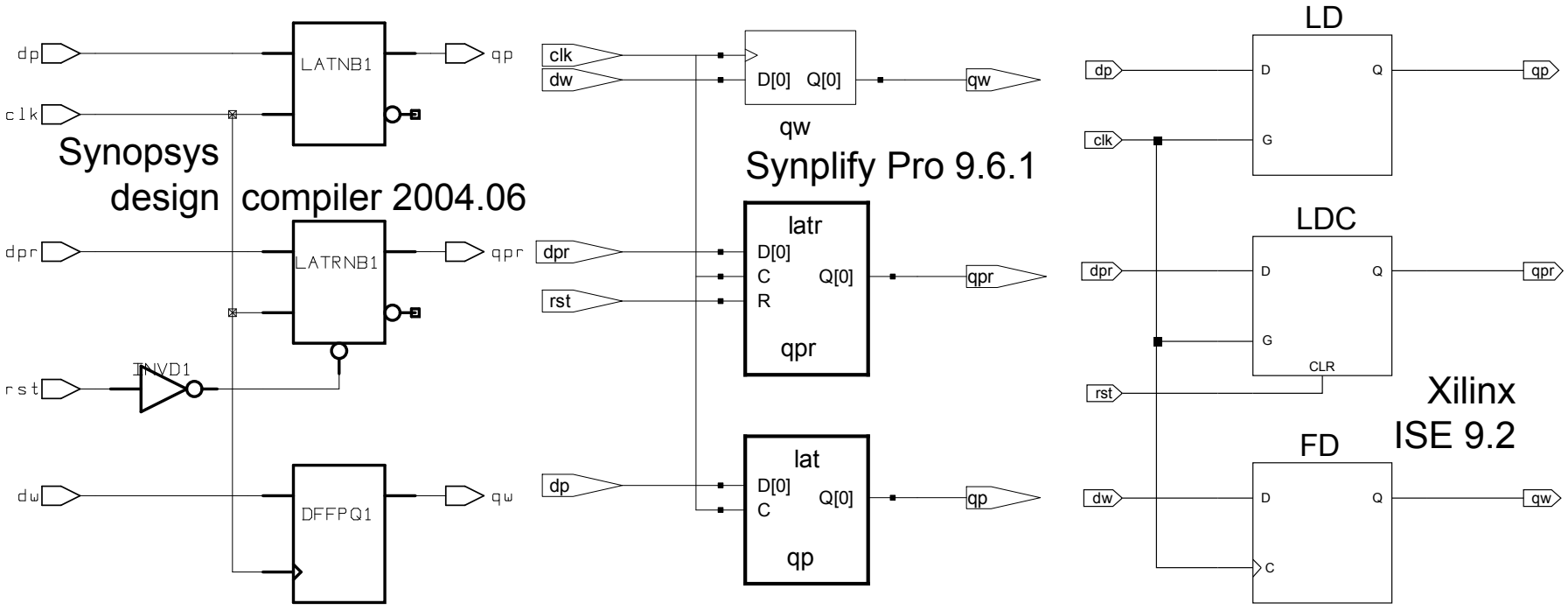
?

```
process
begin
  wait until clk='1';
    qw <= dw;
end process;
```

DFF

The same VHDL code synthesized with 6 different tools and simulated (next slides)

DFF or Latch (3) ?




```
process(clk, rst)
begin
  if rst = '1' then
    qpr <= '0';
  elsif clk='1' then
    qpr <= dpr;
 end if;
end process;
```

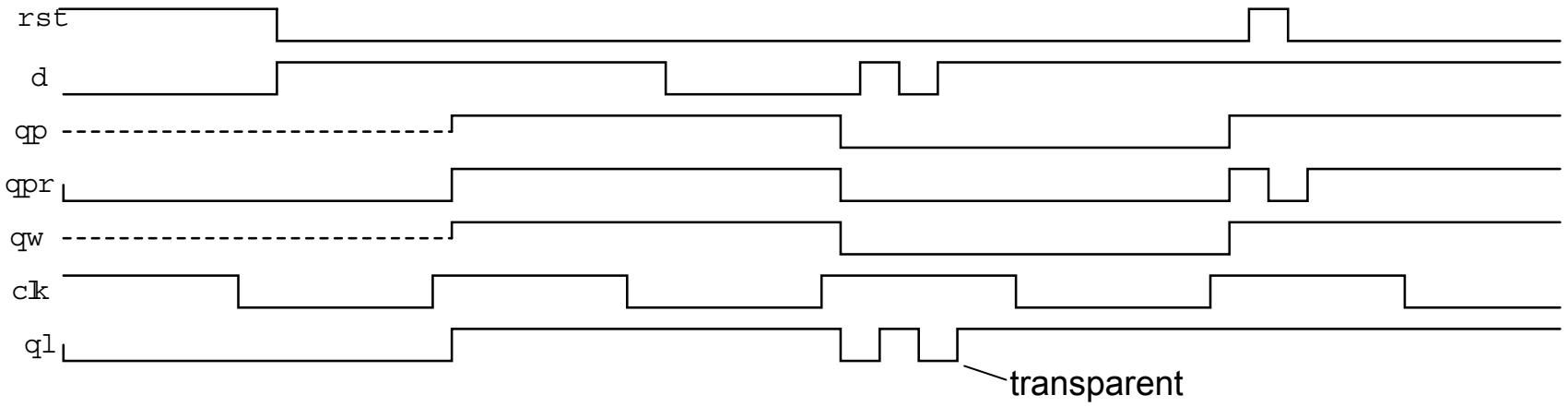
LATCH
(with warnings)

```
process(clk)
begin
  if clk='1' then
    qp <= dp;
  end if;
end process;
```

```
process
begin
  wait until clk='1';
    qw <= dw;
end process;
```

DFF

DFF or Latch (4) ?




```
process(clk, rst)
begin
  if rst = '1' then
    qpr <= '0';
  elsif clk='1' then
    qpr <= d;
 end if;
end process;
```

```
process(clk)
begin
  if clk='1' then
    qp <= d;
  end if;
end process;
```

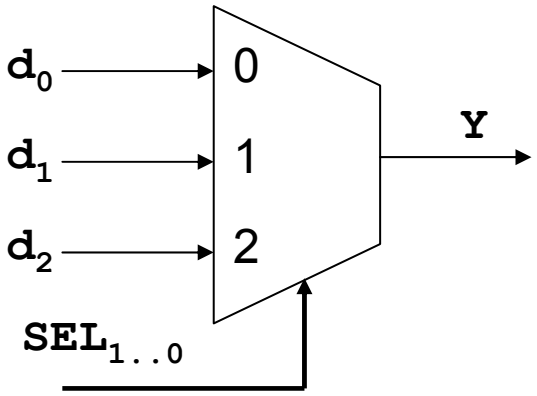
```
process
begin
  wait until clk='1';
    qw <= d;
end process;
```

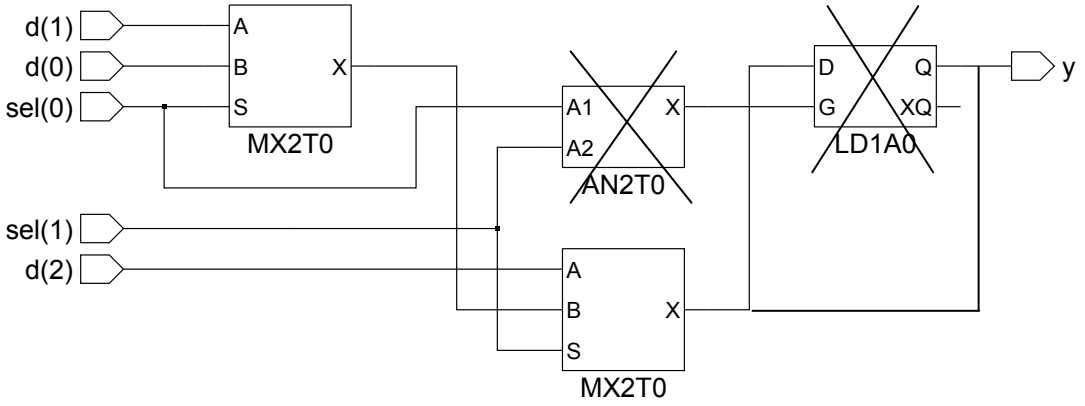
```
process(clk, d)
begin
  if clk='1' then
    ql <= d;
  end if;
end process;
```

Not recommended!

As expected, in the **functional simulation** (ModelSim and Aldec AHDL) all **3 processes** behave as **edge triggered DFFs**! But 6 different synthesis tools found 1 or 2 latches! The fourth process on this page is a **latch** and is used for reference in the waveform.

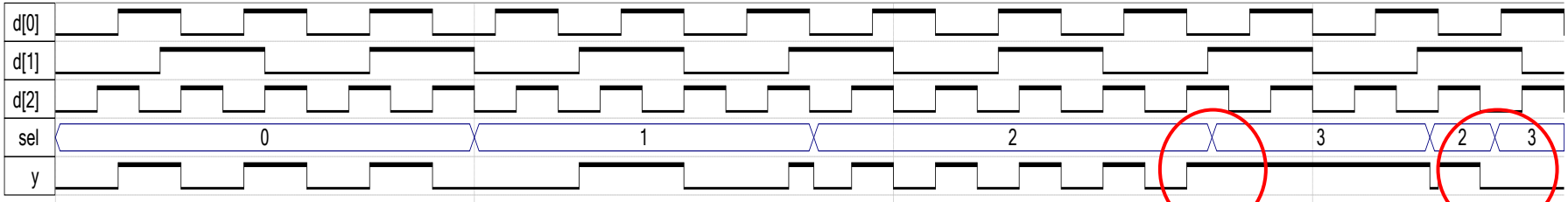
Unwanted Latch






```
process(sel, d)
begin
    if    sel = "00" then y <= d(0);
    elsif sel = "01" then y <= d(1);
    elsif sel = "10" then y <= d(2);
    else                  y <= '-';
    end if;
end process;
```

If the output is not specified for all possible inputs, Latches are inferred!



y preserves its last value when **sel** goes to **"11"**

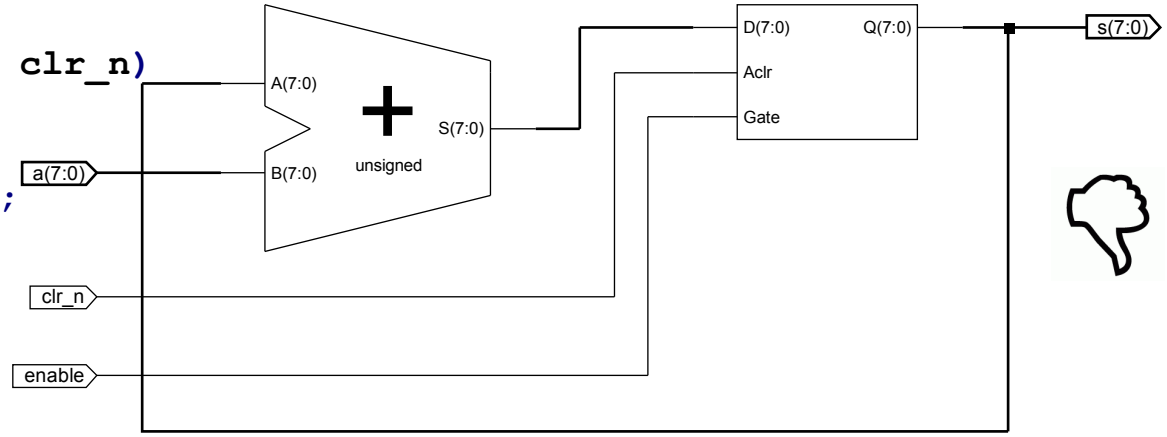
How to avoid unwanted Latches?

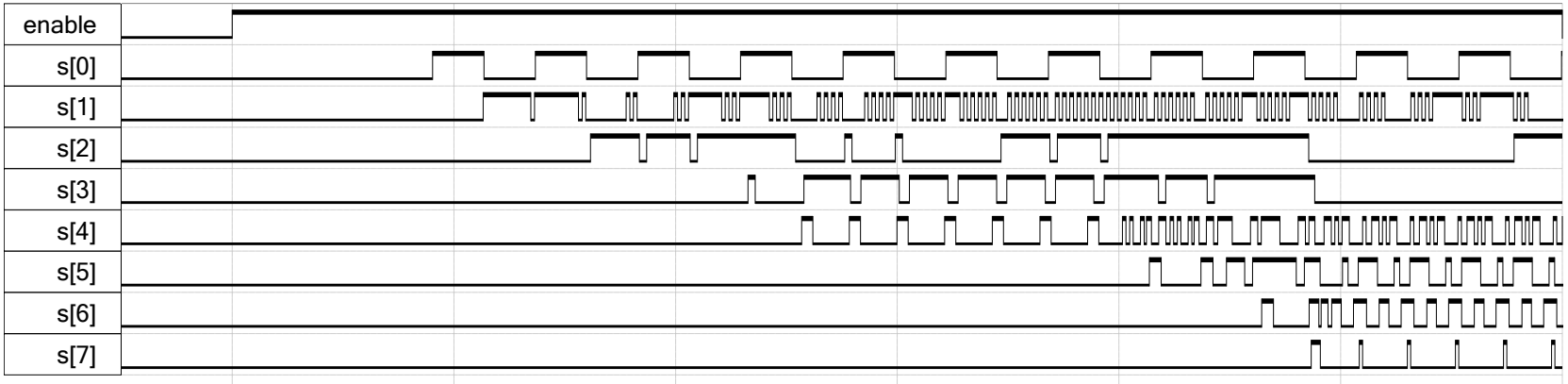
- Anyway using latches is not recommended, except in some special cases
- Unwanted latches increase the size of the circuit, make the timing analysis more difficult (static timing analysis)
- All compilers assert warnings when finding a latch
  - The problem is, sometimes the warnings are so many and it is hard to read all of them
- In order to avoid latches in combinational circuits:
  - initialize the output signal at the beginning of the process, `'-'` is in many cases a good choice
  - use **`case`** instead of **`if-then-else`**, when you have non-priority choice, but this doesn‘t automatically prevent latches
- The synthesis tools don't interpret correctly the sensitivity lists, use only widely accepted descriptions of DFFs!

Combinational loop(1)

Latches are inferred when a feedback is detected by the synthesis tool. Here the problem is not only the latch itself, the feedback brings the circuit to oscillations.

```
process(a, sum, enable, clr_n)
begin
 if clr_n = '0' then
  sum <=(others => '0');
 elsif enable='1' then
  sum <= sum + a;
 end if;
end process;
s <= sum;
```

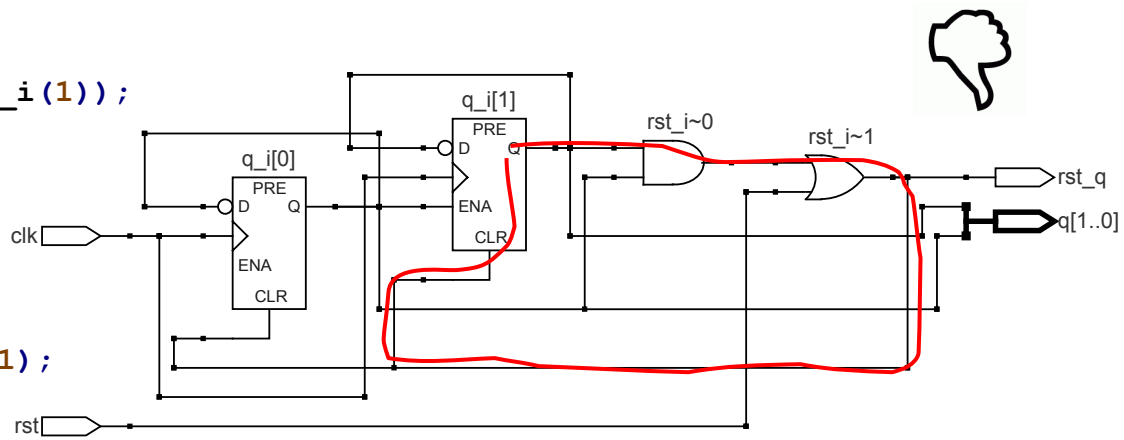




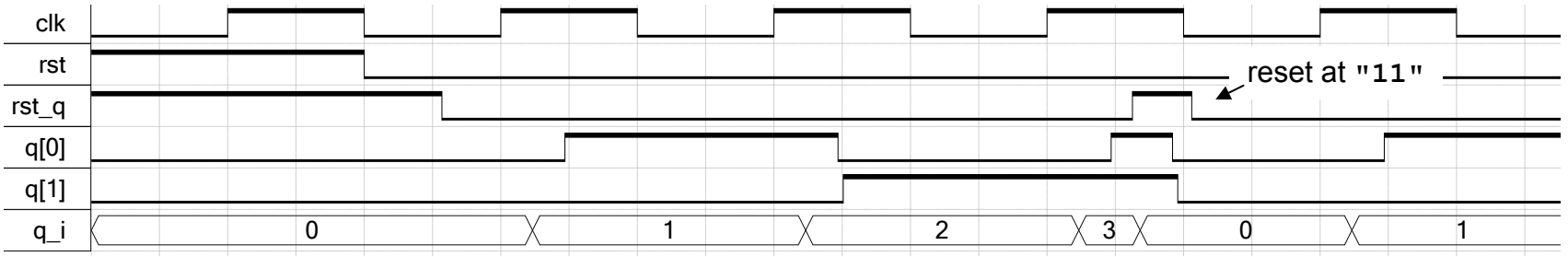
Combinational loop(2)

This should be a counter modulo 3. Its states are `"00"`, `"01"`, `"10"`. The counter enters for short time the state `"11"`, but then a reset pulse is generated.

```
rst_i <= rst or (q_i(0) and q_i(1));
process(clk, rst_i)
begin
  if rst_i = '1' then
    q_i <= "00";
  elsif rising_edge(clk) then
    q_i(0) <= not q_i(0);
    q_i(1) <= q_i(0) xor q_i(1);
  end if;
end process;
q <= q_i;
rst_q <= rst_i;
```



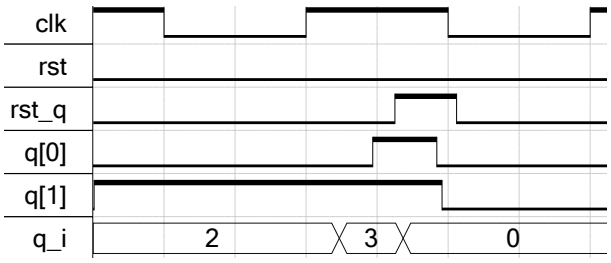
…more about this example in "Timing simulations"



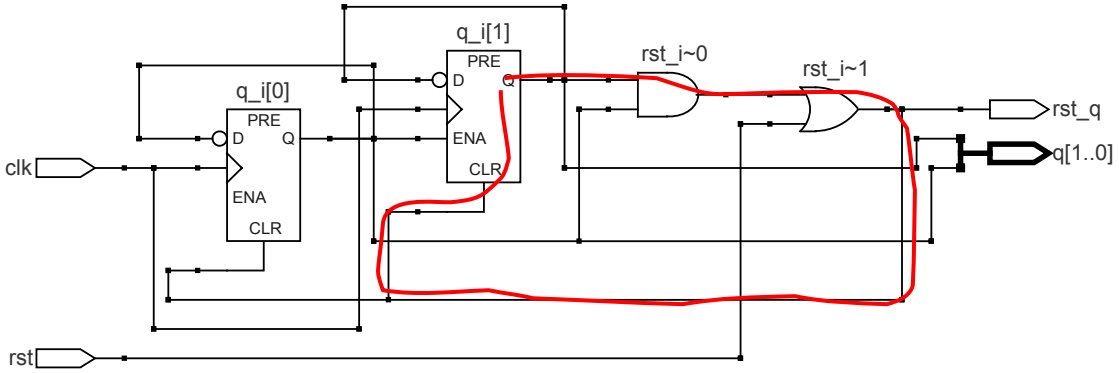
Combinational loop(3)

There are several problems with this circuit:

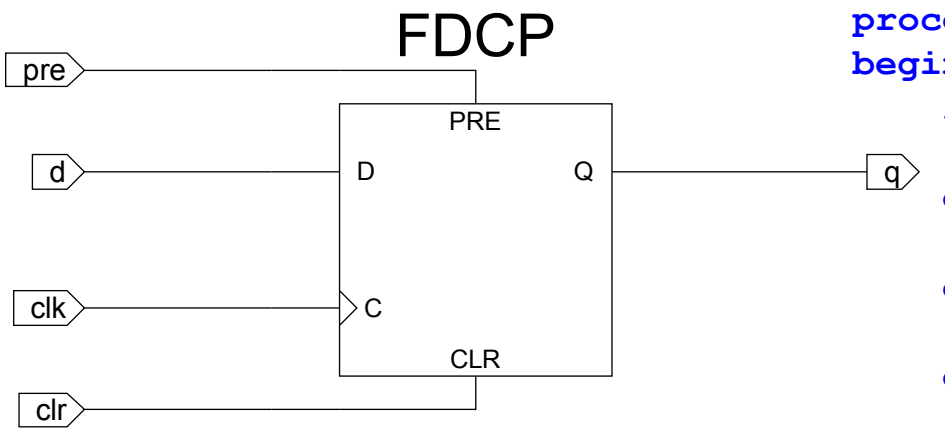
- There are two combinational loops - each starts from a DFF output, goes through the AND and OR gates and closes to the reset port of the same DFF
- The duration of the reset pulse is probably on the lower limit, it could happen (depending on the supply voltage and temperature), that only one of the DFFs is cleared successfully
- It could happen, that after `"00"` and `"01"` when going to `"10"`, both DFFs are for short time high, which leads to a reset pulse with very short duration
- The timing analysis is hard to be done
- The combinational logic has less than the usual complete period after an asynchronous reset, setup time violation is possible



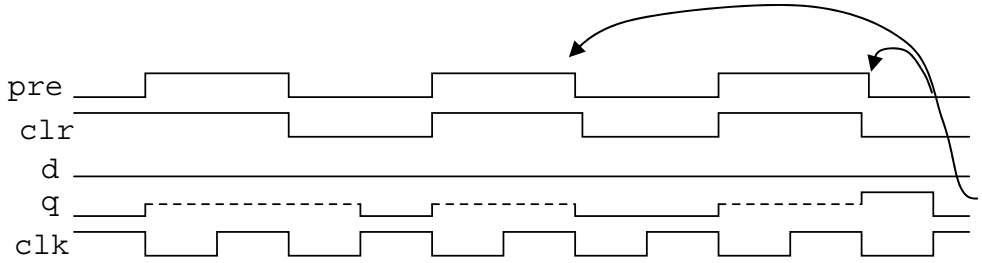
…more for this example in "Timing simulations"



DFF with asynchronous set and reset




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



Not a good idea to use both **`pre`** and **`clr`**:

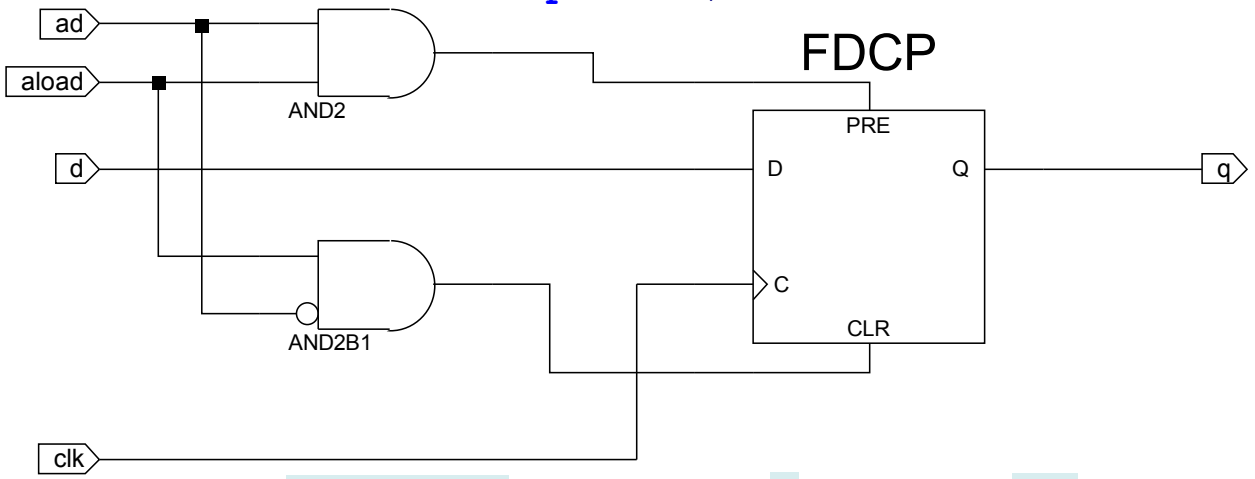
- small additional delays and glitches can make the behaviour non-deterministic

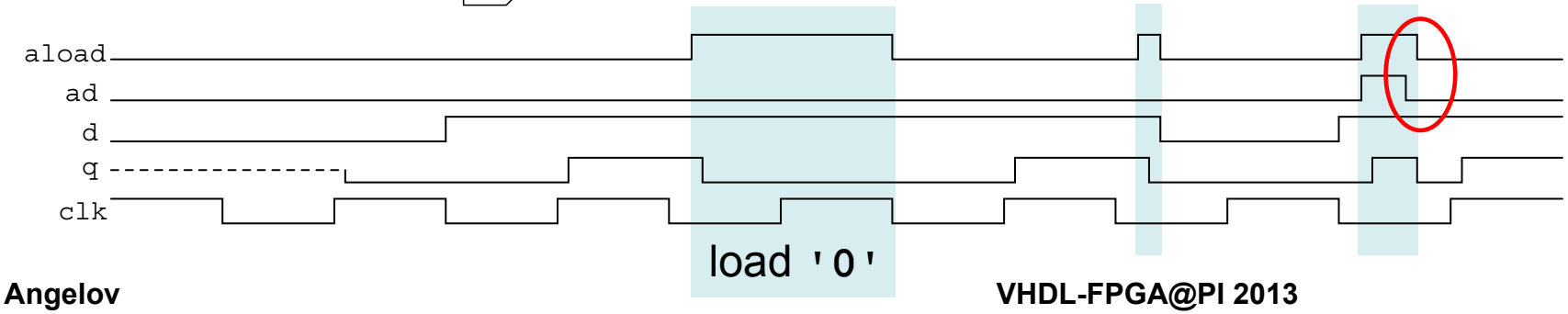
DFF with asynchronous load

Not a good idea:
- additional logic resources
- difficult to fulfil timing conditions (**ad** must be valid when **aload** goes inactive)

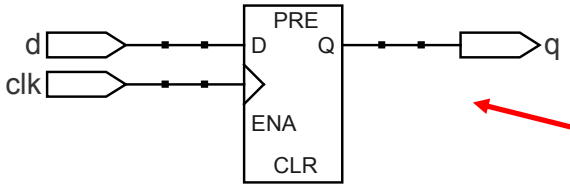
```
process(clk, aload, ad)
begin
    if aload = '1' then
        q <= ad;
    elsif clk'event and clk='1' then
        q <= d;
    end if;
end process;
```

When taking only **ad** and **aload** signals into account, this is a pure latch





Shift register with variables

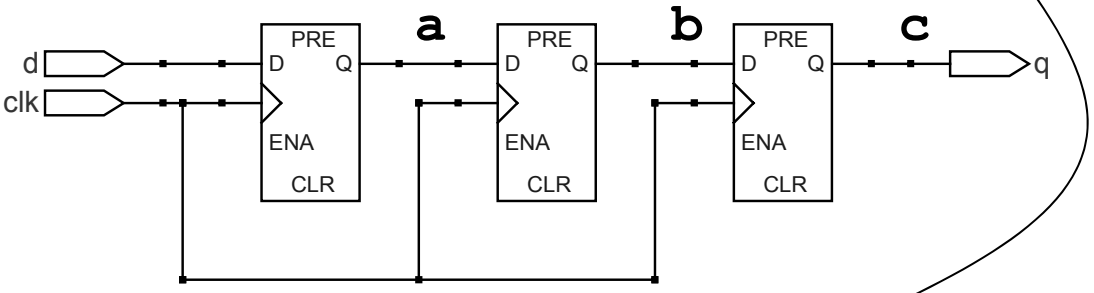



```
process(clk)
variable a,b,c : std_logic;
begin
    if clk'event and clk='1' then
        a := d;          c := b;
        b := a;          b := a;
        c := b;          a := d;
    end if;
    q <= c;
end process;
```

wrong order!

correct!

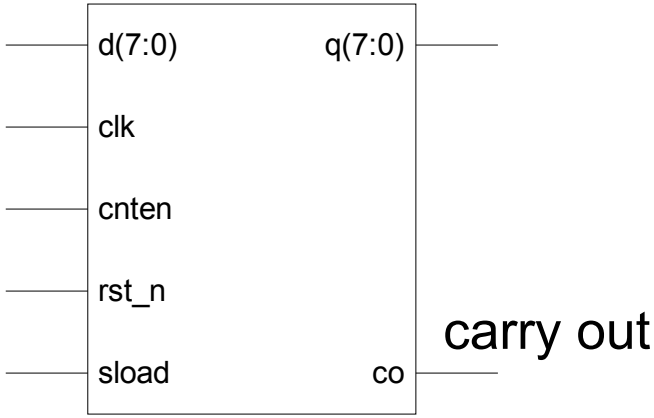
For a **variable**, the VHDL code in the **process** is interpreted exactly like in a normal computer program, each assignment is immediate! For a **signal** the last value assigned is copied to it at the **end** of the **process**!

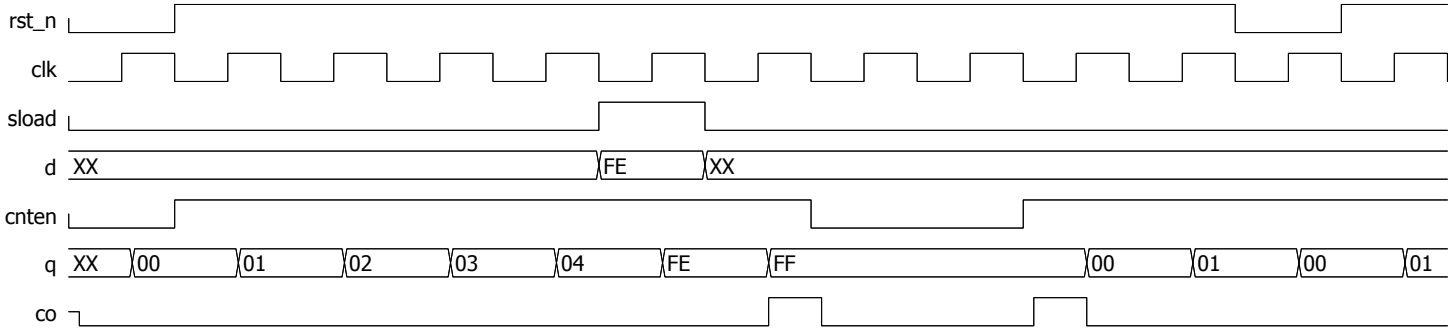


If using **signal**s instead of **variable**s, the order of the three lines is **not** important! Avoid using **variable**s for real signals!

A simple counter with synchronous reset & load

```
signal q_i    : std_logic_vector(q'range);
constant full : std_logic_vector(q'range) := (others => '1');
begin
process(clk)
begin
  if clk'event and clk='1' then
    if rst_n = '0' then
      q_i <= (others => '0');
    elsif sload = '1' then
      q_i <= d;
    elsif cnten = '1' then
      q_i <= q_i + 1;
    end if;
  end if;
end process;
q  <= q_i;
co <= '1' when q_i = full and sload = '0' and cnten = '1' else '0';
```





Counter with signal/variable and std_logic_vector(1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity counter_var is
generic (N : Natural := 8);
port (
    clk    : in  std_logic;
    rst_n  : in  std_logic;
    cnten  : in  std_logic;
    qss    : out std_logic_vector(N-1 downto 0);
    qsv    : out std_logic_vector(N-1 downto 0));
end counter_var;
architecture a of counter_var is
signal  cntss    : std_logic_vector(qss'range);
begin
 -- using std_logic signal
    process(clk, rst_n)
    begin
        if rst_n = '0' then
            cntss <= (others => '0');
        elsif clk'event and clk='1' then
          if cnten = '1' then
            cntss <= cntss + 1;
          end if;
        end if;
    end process;
    qss <= cntss;
```

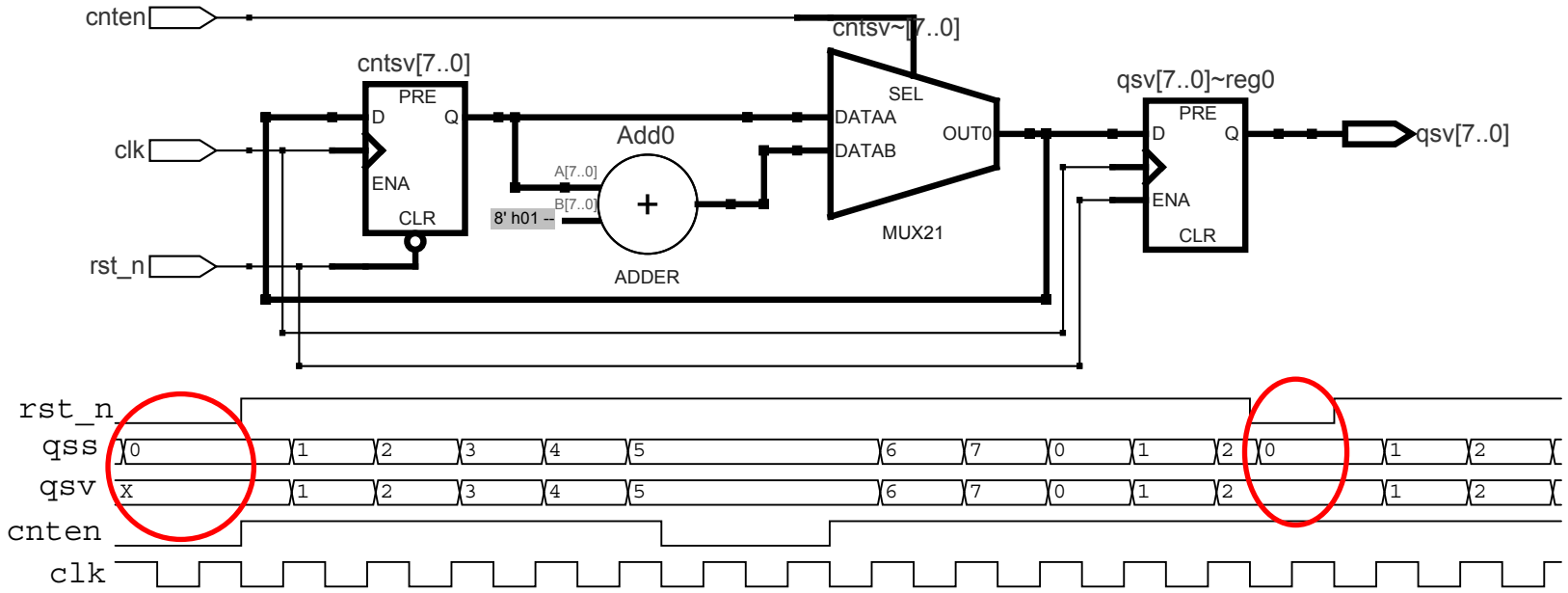
Overflow here is ignored!

```
-- using std_logic variable
    process(clk, rst_n)
variable cntsv : std_logic_vector(qsv'range);
begin
    if rst_n = '0' then
        cntsv := (others => '0');
    elsif clk'event and clk='1' then
      if cnten = '1' then
        cntsv := cntsv + 1;
      end if;
    end if;
    qsv <= cntsv;
end process;
```

?

Both descriptions are equivalent.
In the case of **variable** be careful where to copy the value of the **variable** to the **signal**!

Counter with signal/variable and std_logic_vector(2)



Wrong position of **this** line! The result is twice more DFFs and strange behaviour!

```
process(clk, rst_n)
  variable cntsv : std_logic_vector(qsv'range);
  begin
    if rst_n = '0' then
      cntsv := (others => '0');
    elsif clk'event and clk='1' then
      if cnten = '1' then cntsv := cntsv + 1; end if;
      qsv <= cntsv;
  end if;
end process;
```

Counter with signal/variable and Integer(1) 👎

```
entity counter_var is
generic (N : Natural := 8);
port (clk    : in  std_logic;
      rst_n  : in  std_logic;
      cnten  : in  std_logic;
      qis    : out std_logic_vector(N-1 downto 0);
      qiv    : out std_logic_vector(N-1 downto 0) );
end counter_var;
...
subtype counter_int is Integer range 0 to 2**N-1;
```

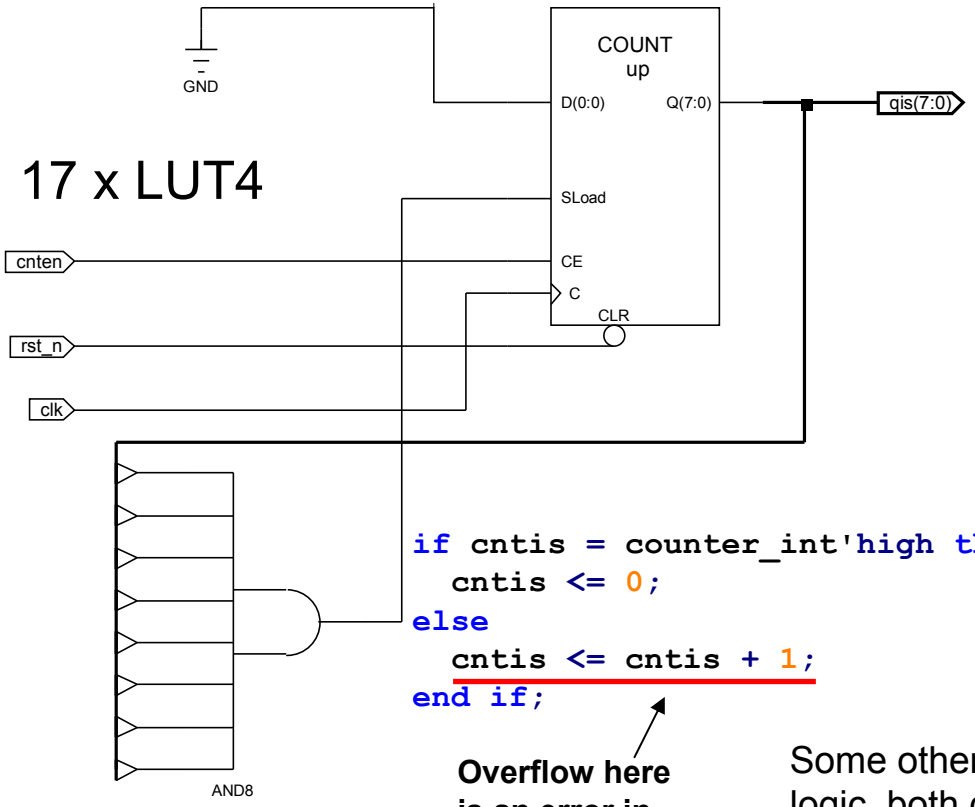
```
signal  cntis : counter_int;
...
process(clk, rst_n)
begin
    if rst_n = '0' then
        cntis <= 0;
    elsif clk'event and clk='1' then
      if cnten = '1' then
        if cntis = counter_int'high then
            cntis <= 0;
        else
            cntis <= cntis + 1;
        end if;
      end if;
    end if;
end process;
qis <= conv_std_logic_vector(cntis, N);
```

Why ?
(next slide)

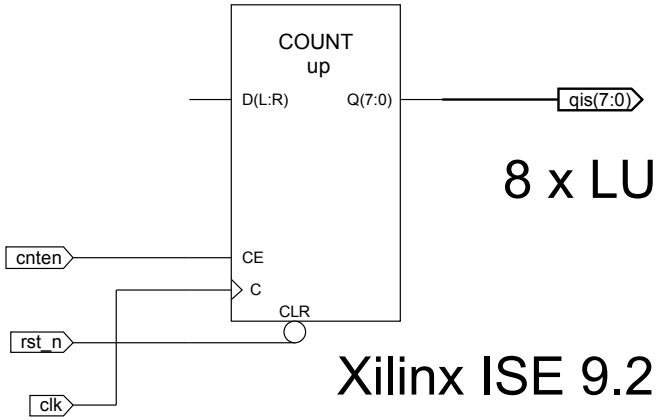
```
-- using integer variable
    process(clk, rst_n)
    variable cntiv : counter_int;
    begin
        if rst_n = '0' then
            cntiv := 0;
        elsif clk'event and clk='1' then
          if cnten = '1' then
            if cntiv = counter_int'high then
                cntiv := 0;
            else
                cntiv <= cntiv + 1;
            end if;
          end if;
        end if;
        qiv <= conv_std_logic_vector(cntiv, N);
    end process;
```

Counter with signal/variable and Integer(2)

Writing simply `cntis <= cntis + 1;` leads to overflow error in the simulation



17 x LUT4



8 x LUT4

Xilinx ISE 9.2

```
if cntis = counter_int'high then
  cntis <= 0;
else
  cntis <= cntis + 1;
end if;
```

Overflow here is an error in simulation!

```
cntis <= (cntis + 1) mod 2**N;
```

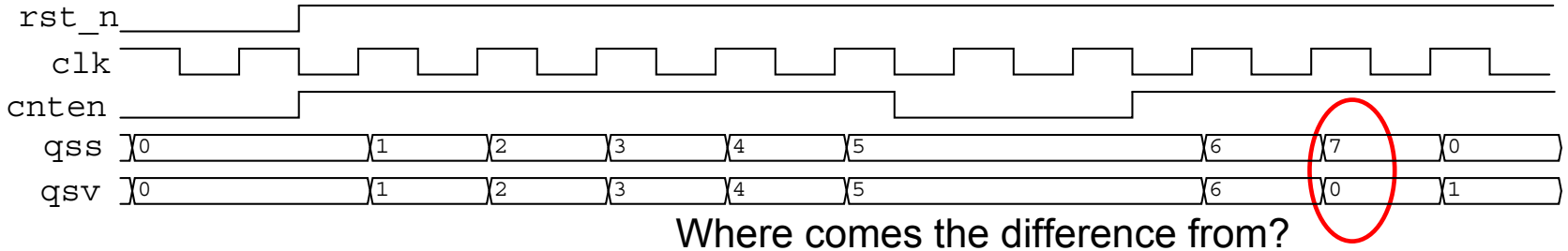
the better solution

Some other synthesis tools do not introduce additional logic, both descriptions are equivalent

Counter with signal/variable

```
signal cntss   : std_logic_vector(qss'range);
constant cntfull : std_logic_vector(qss'range) := (others => '1');
begin
process(clk, rst_n)

begin
  if rst_n = '0' then
    cntss <= (others => '0');
  elsif clk'event and clk='1' then
    if cnten = '1' then
      cntss <= cntss + 1;
      if cntss = cntfull then
        cntss <= (others => '0');
      end if;
    end if;
  end if;
end process;
qss <= cntss;
```

```
process(clk, rst_n)
variable cntsv : std_logic_vector(qsv'range);
begin
  if rst_n = '0' then
    cntsv := (others => '0');
  elsif clk'event and clk='1' then
    if cnten = '1' then
      cntsv := cntsv + 1;
      if cntsv = cntfull then
        cntsv := (others => '0');
      end if;
    end if;
  end if;
  qsv <= cntsv;
end process;
```

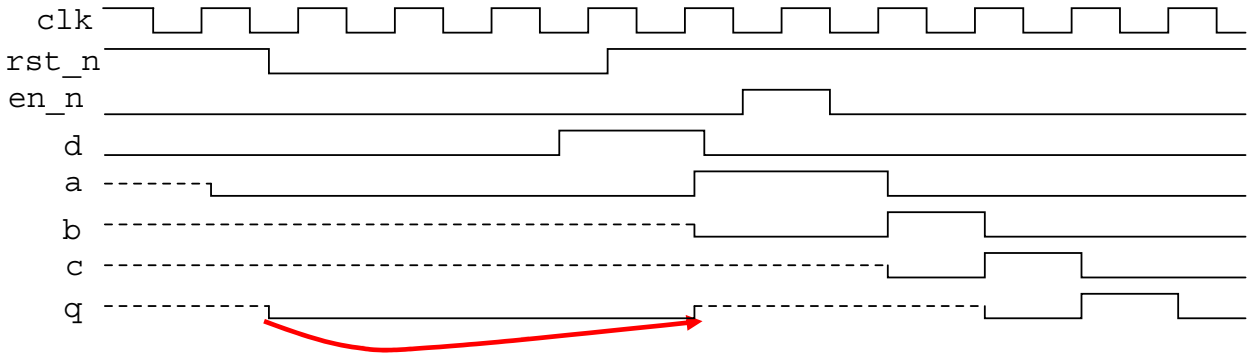


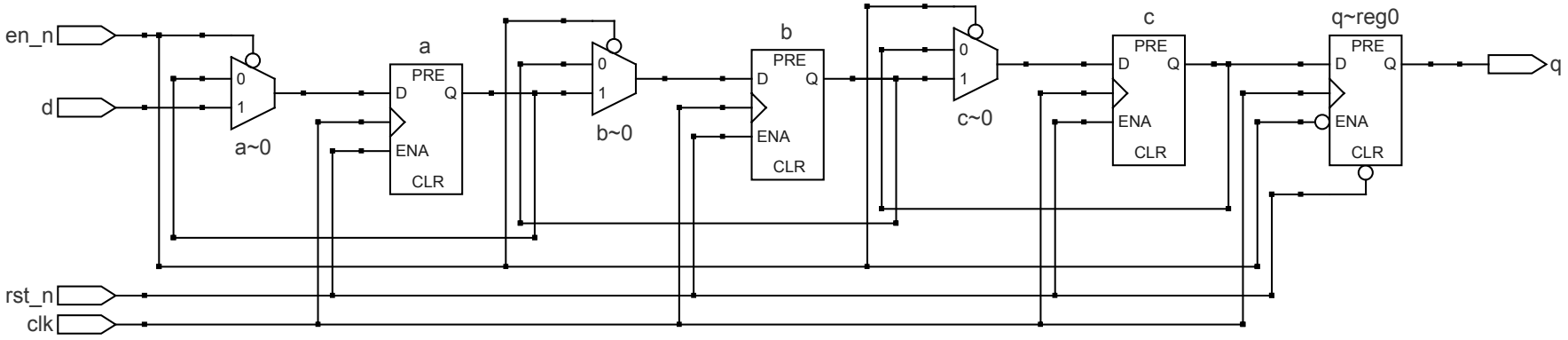
Where comes the difference from?

Shift register with bad reset

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

If any register is missing in the reset part of the **process**, additional logic is inserted and the reset is impossible!



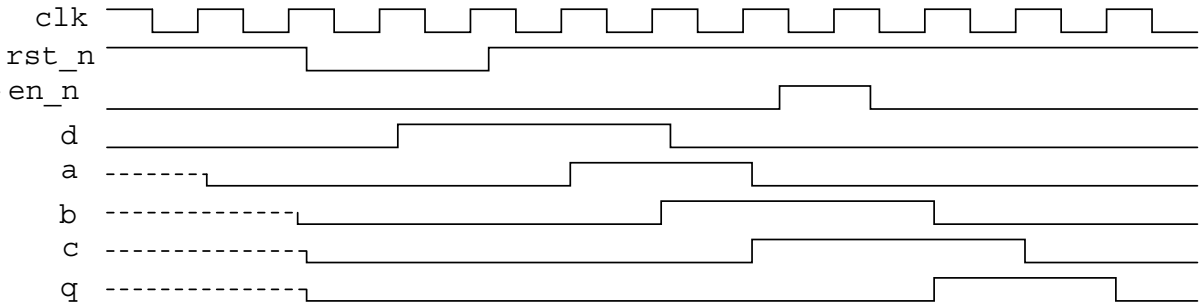


Shift register – correct reset

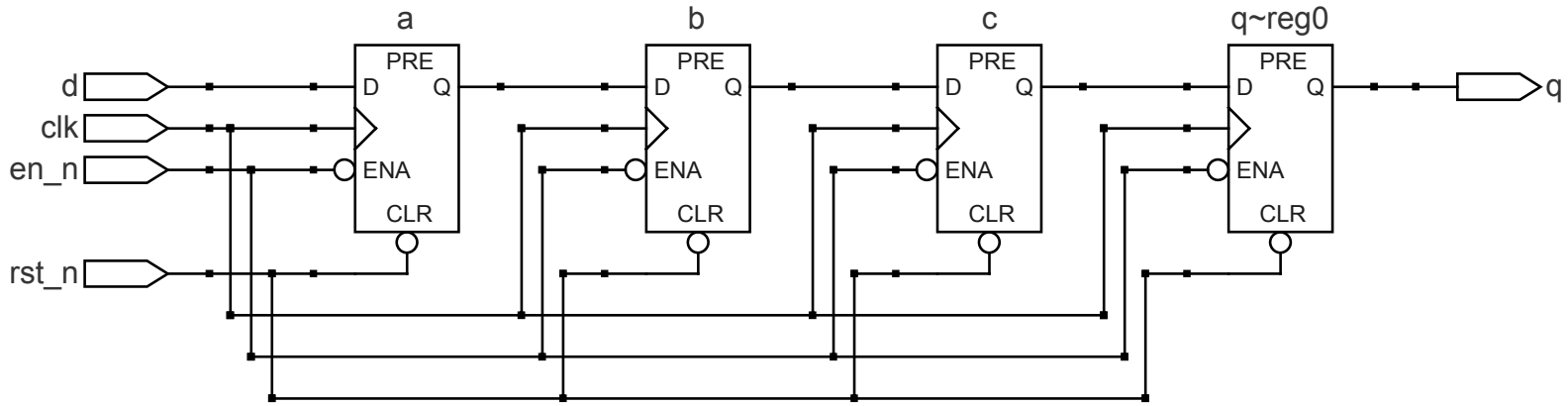
```
process(clk, rst_n)
begin
  if rst_n = '0' then
    q <= '0'; a <= '0'; b <= '0'; c <= '0';
  elsif clk'event and clk='1' then
    if en_n = '0' then
      a <= d;
      b <= a;
      c <= b;
      q <= c;
    end if;
  end if;
end process;
```

either all or no reset!

The order is not important

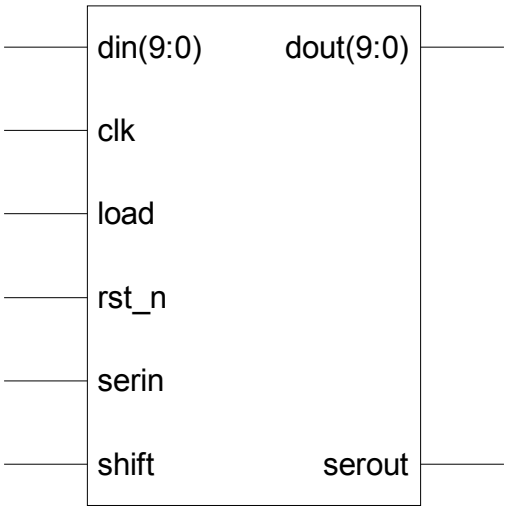


What will happen if we use latches instead of DFFs?



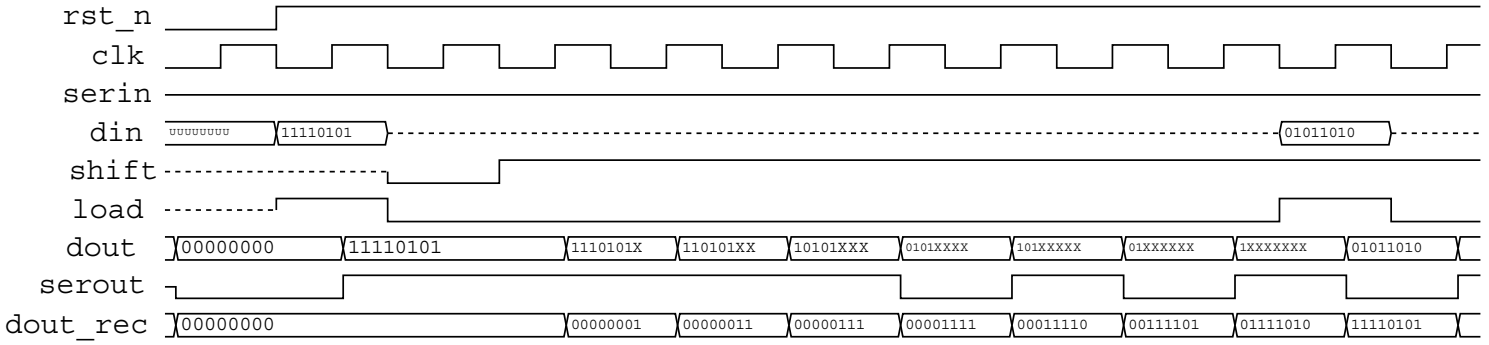
Shift register with parallel load

```
entity shift_reg_full is
generic (N : Natural := 10);
port(
...
signal q : std_logic_vector(N-1 downto 0);
begin
process(clk, rst_n)
begin
 if rst_n = '0' then q <= (others => '0');
 elsif clk'event and clk='1' then
  if load = '1' then q <= din;
  elsif shift='1' then
   q <= q(N-2 downto 0) & serin;
  end if;
 end if;
end process;
```



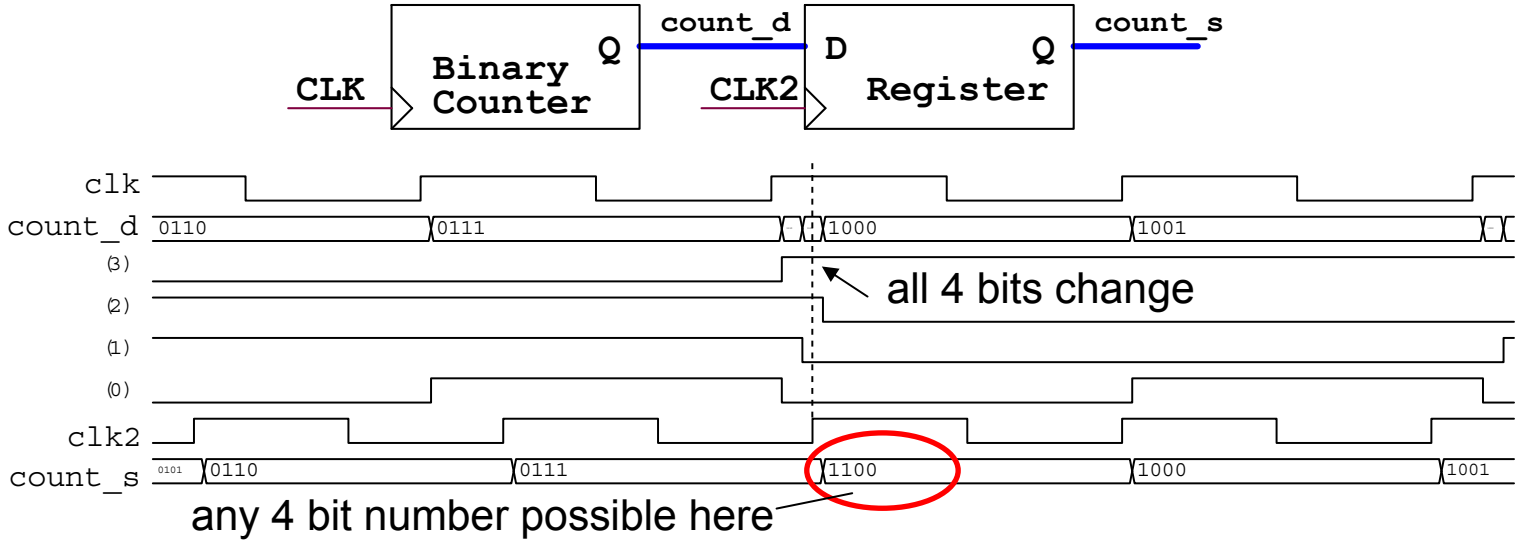

```
serout <= q(N-1);
dout   <= q;
end;
```

Two modules, one sends serially (**serout**) to the other



Gray code and Gray counters(1)

- In case of a N-bit binary counter the number of bits changed at each clock vary from 1 to N, e.g. **0111 → 1000 → 1001 … 1111 → 0000**
- This might be a problem when transferring the counts to another clock domain due to the slight differences of the timing of the different bits



Gray code and Gray counters(2)

- Hamming distance between two N-bit words is defined as the number of differences in all bit positions, e.g.

  001**0**1 and

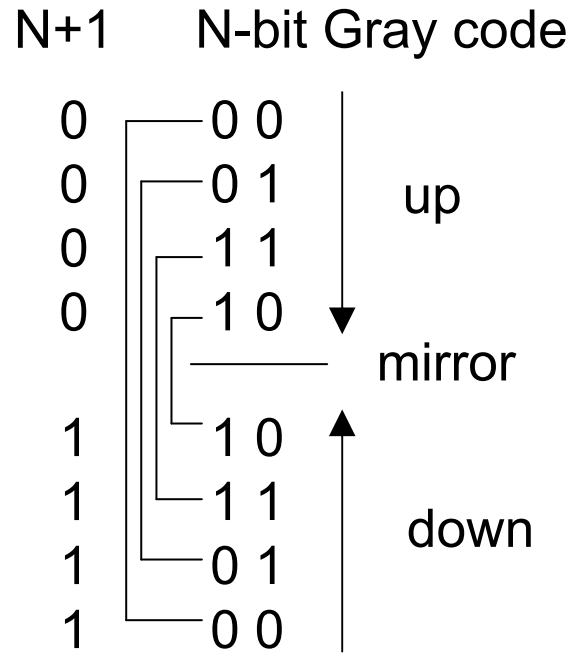
  101**1**1

  have Hamming distance of **2**
- In the binary coding the hamming distance between two adjacent codes can be from 1 to N
- In the Gray coding the hamming distance between two adjacent codes is always 1
- Unfortunately the arithmetic operations in Gray code are not easy, but
- The conversion from binary to Gray and back is not so difficult

Gray code and Gray counters(3)

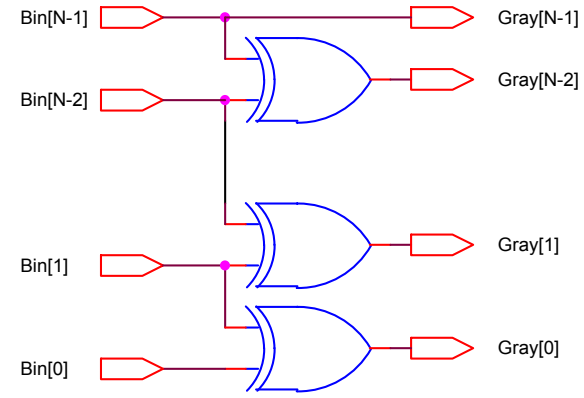
Binary	distance	Gray
0 0 0 0		0 0 0 0
0 0 0 1	> 1	0 0 0 1
0 0 1 0	> 2	0 0 1 1
0 0 1 1	> 1	0 0 1 0
0 1 0 0	> 3	0 1 1 0
0 1 0 1	> 1	0 1 1 1
0 1 1 0	> 2	0 1 0 1
0 1 1 1	> 1	0 1 0 0
1 0 0 0	> 4	1 1 0 0
1 0 0 1	> 1	1 1 0 1
1 0 1 0	> 2	1 1 1 1
1 0 1 1	> 1	1 1 1 0
1 1 0 0	> 3	1 0 1 0
1 1 0 1	> 1	1 0 1 1
1 1 1 0	> 2	1 0 0 1
1 1 1 1	> 1	1 0 0 0
0 0 0 0	> 4	0 0 0 0

How to generate the code?



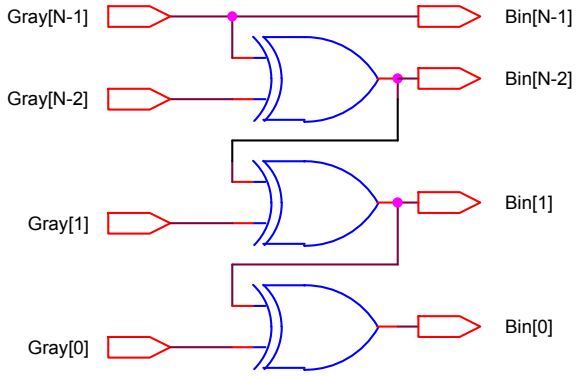
Typical example for recursion
→ good for software, but not for hardware implementation

Gray code and Gray counters(4)



Binary → Gray

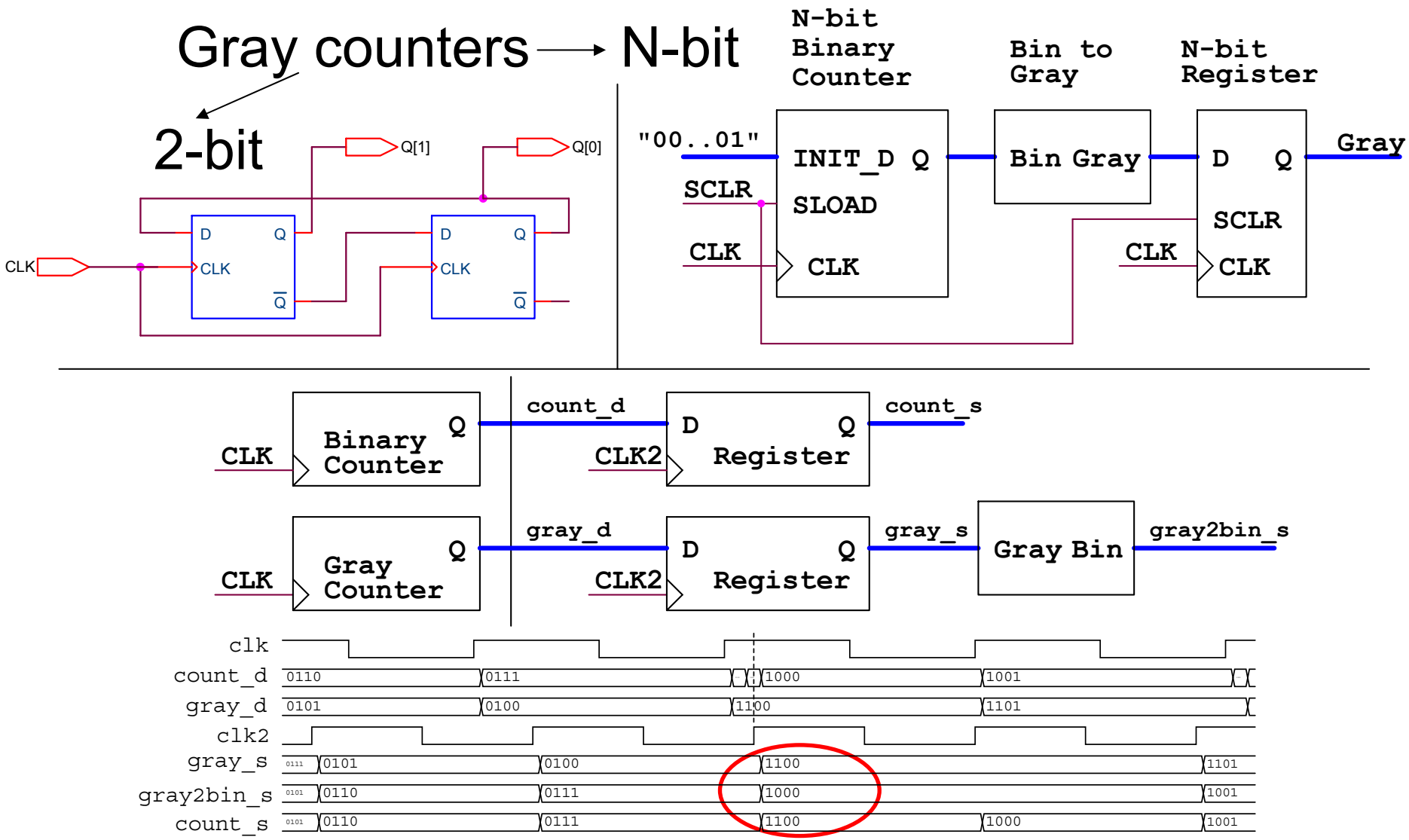
```
...
gray <= '0' & bin(N-1 downto 1) xor bin;
...
```



Gray → binary

```
...
signal bin_i : std_logic_vector(gray'range);
begin
    bin_i(N-1) <= gray(N-1);
binc: for i in N-2 downto 0 generate
    bin_i(i) <= gray(i) xor bin_i(i+1);
end generate;
    bin <= bin_i;
...
```

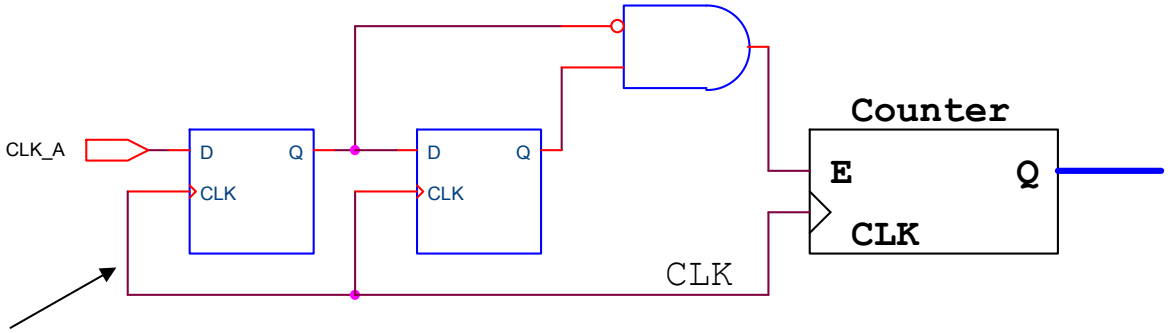
Gray code and Gray counters(5)



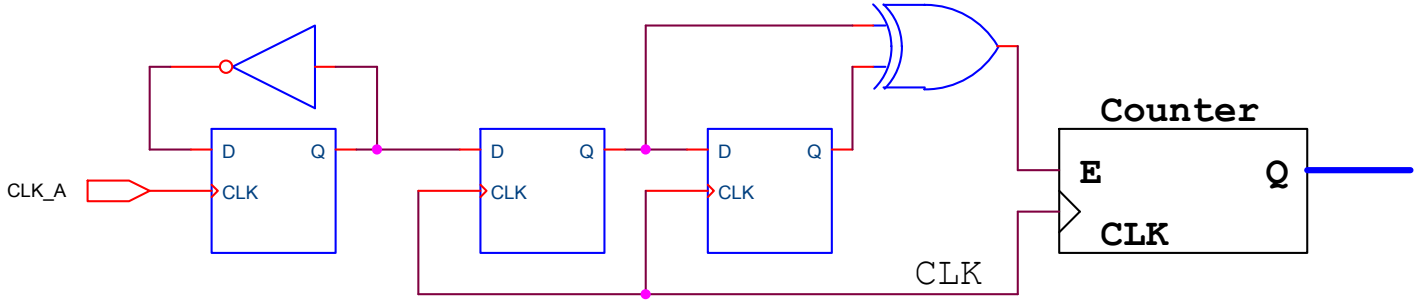
Counting asynchronous events(1)

- It is not very nice to have tens of clocks in a single chip
  - Many clock domains (trees)
  - The counter & register outputs are not synchronous to the system clock and can not be used directly
- It might be better to run the counters with the system clock and to enable each counter properly
- Provided the input frequency is not higher than the system clock, there is a very simple solution

Counting asynchronous events(2)



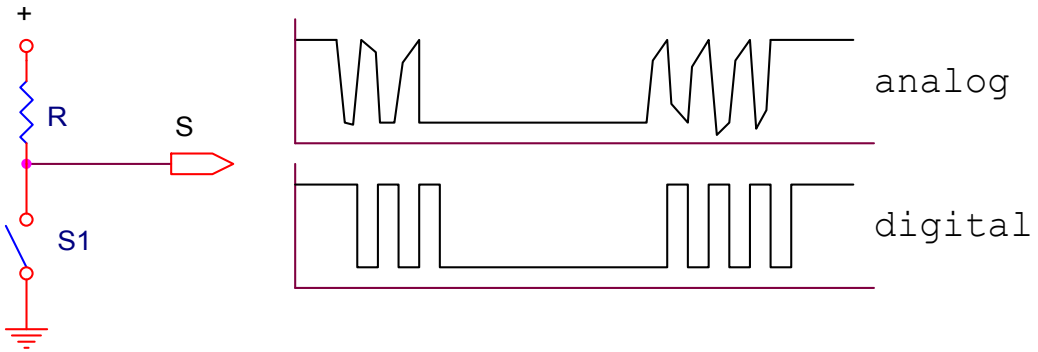
Edge detection – the pulse duration on the **CLK_A** input must be larger than the period of the **CLK**, even if the frequency of **CLK_A** is very low



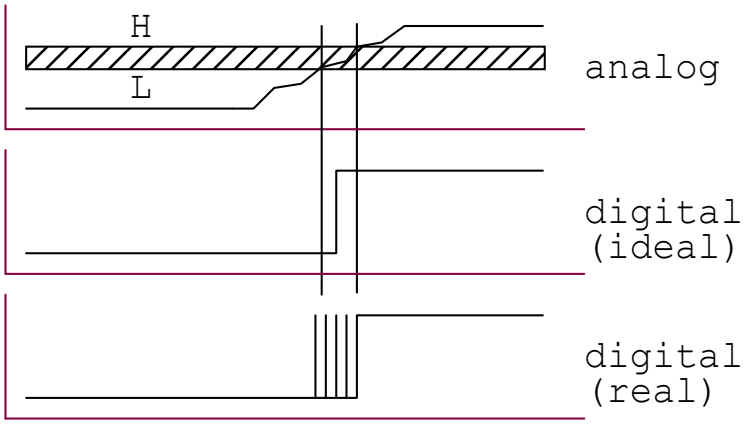
Here the only requirement is that the input frequency is below the system clock **CLK**

Filter for noisy or slow signals(1)

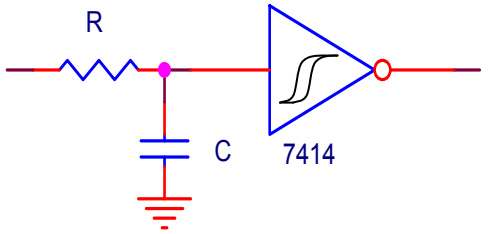
- Some signals come from mechanical buttons and have many transitions



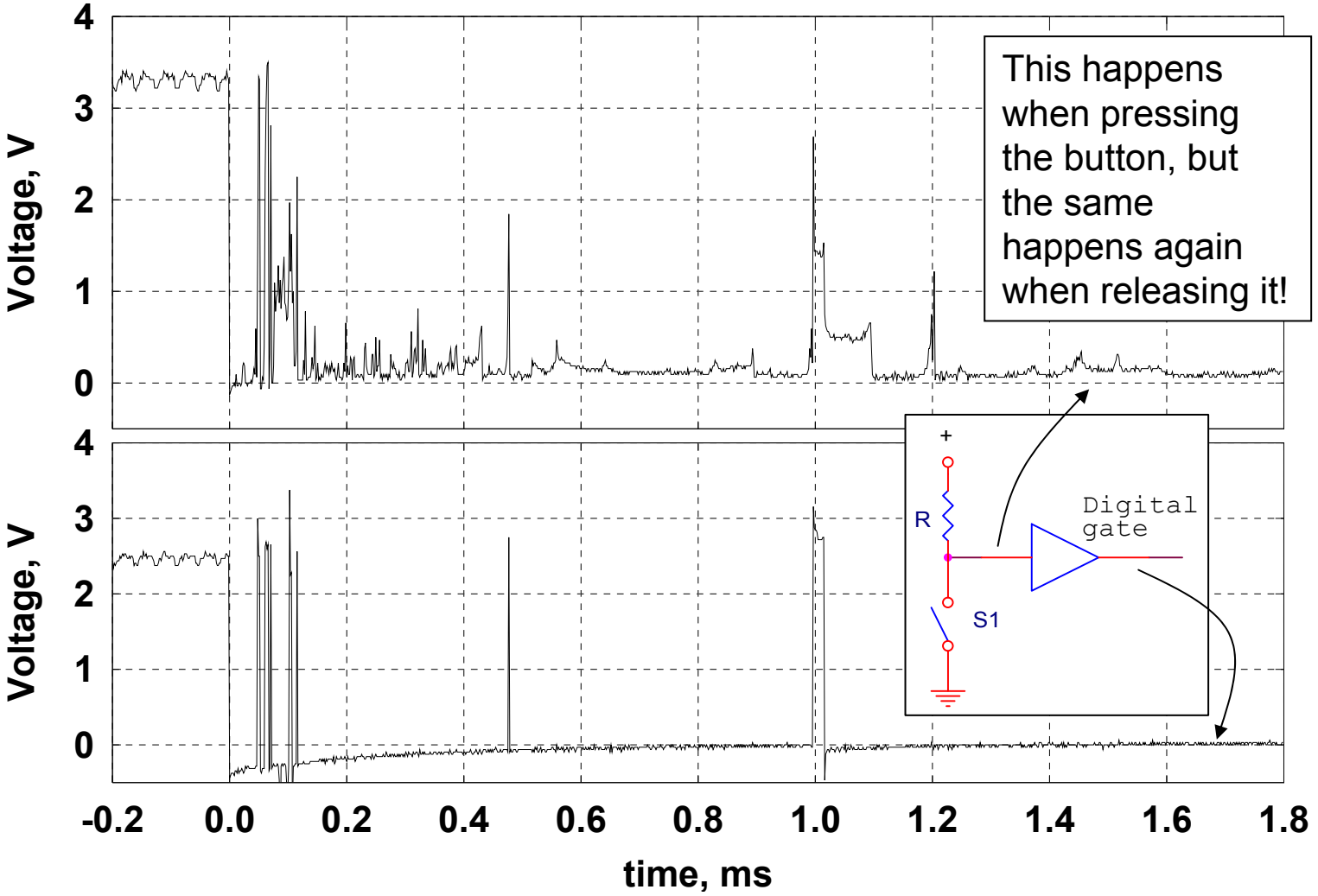
- or are very slow, compared to the typical digital signals



One possible solution: low pass filter + Schmidt trigger with hysteresis



Filter for noisy or slow signals(2)



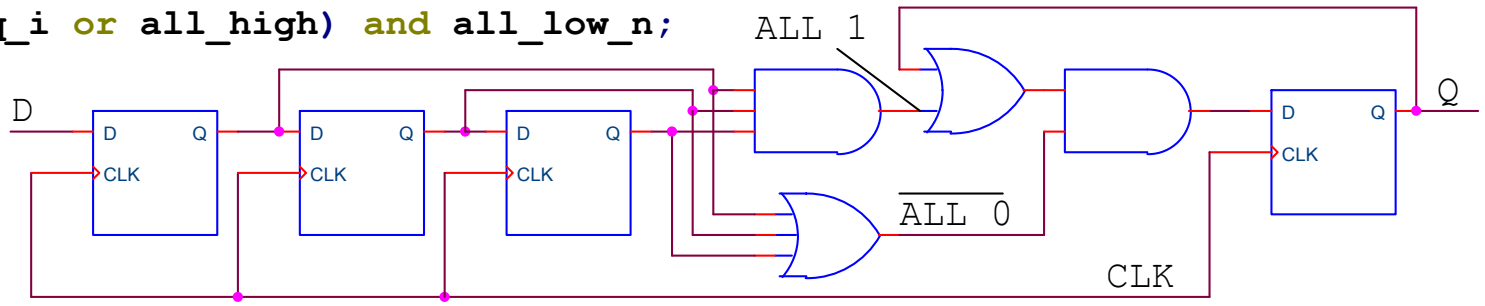
Filter for noisy or slow signals(3)

```
entity filt_short is
generic (N : Integer := 3);
port(clk  : in    std_logic;
     d    : in    std_logic;
     q    : out   std_logic);
end filt_short;
architecture a of filt_short is
signal   samples  : std_logic_vector(N-1 downto 0);
constant samples1 : std_logic_vector(N-1 downto 0) := (others => '1');
constant samples0 : std_logic_vector(N-1 downto 0) := (others => '0');
signal all_high, all_low_n, q_i : std_logic;
begin
  all_high  <= '1' when samples = samples1 else '0';
  all_low_n <= '0' when samples = samples0 else '1';
  process(clk)
  begin
    if clk'event and clk='1' then
      samples <= samples(N-2 downto 0) & d;
      q_i <= (q_i or all_high) and all_low_n;
    end if;
  end process;
  q <= q_i;
end;
```

Using a shift register

The input is shifted in, the output is changed only if all bits in the shift register are equal

The clock frequency must be low enough, for mechanical buttons in the order of 100Hz or below

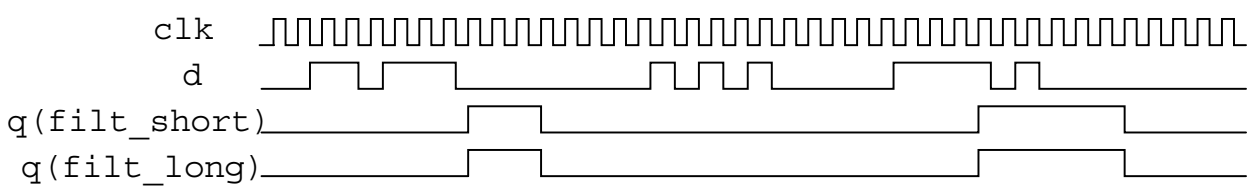


Filter for noisy or slow signals(4)

```
entity filt_long is
generic (N : Integer := 3);
port(clk   : in    std_logic;
     d     : in    std_logic;
     q     : out   std_logic);
end filt_long;
architecture a of filt_long is
signal counts  : Integer range 0 to N-2;
signal samples : std_logic_vector(1 downto 0);
begin
  process(clk)
  begin
    if clk'event and clk='1' then
      samples <= samples(0) & d;
      if (samples(0) xor samples(1)) = '1' then -- different
        counts <= N-2;
      else
        if counts = 0 then q <= samples(1);
                      else counts <= counts - 1; -- count down
        end if;
      end if;
    end if;
  end process;
end;
```

Using a counter

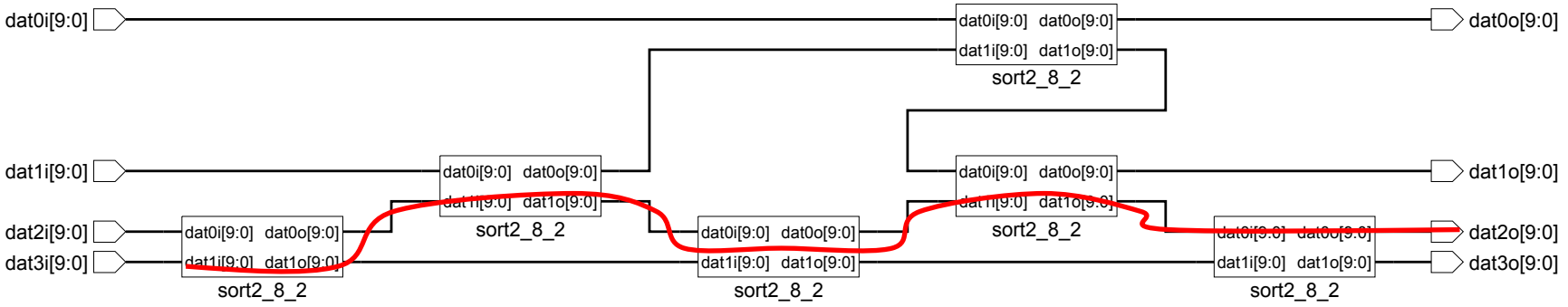
The input signal **d** must remain unchanged for **N clk** cycles in order to be copied to the output **q,** otherwise the counter will be reloaded



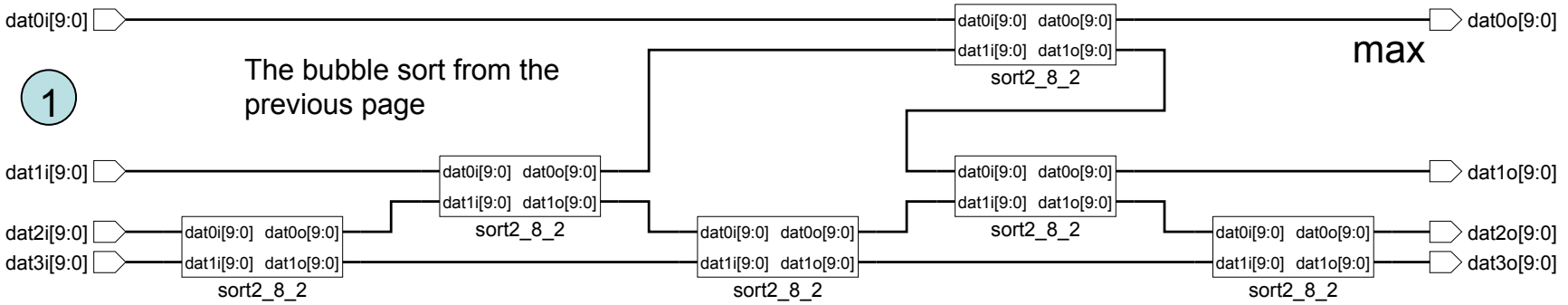
Sort4 – bubble sort

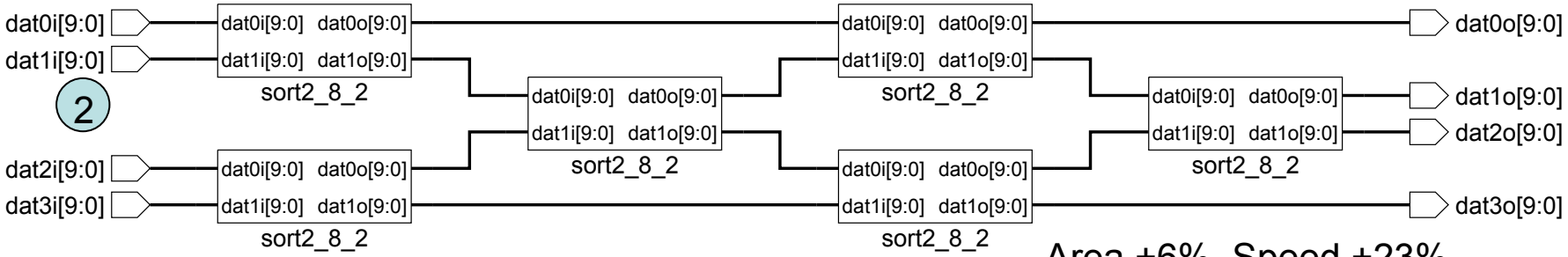
Lets try to sort 4 numbers in hardware. Maybe the simplest way is to first build a sorter for 2 numbers. Then combine many such blocks to realize something like a bubble sort.

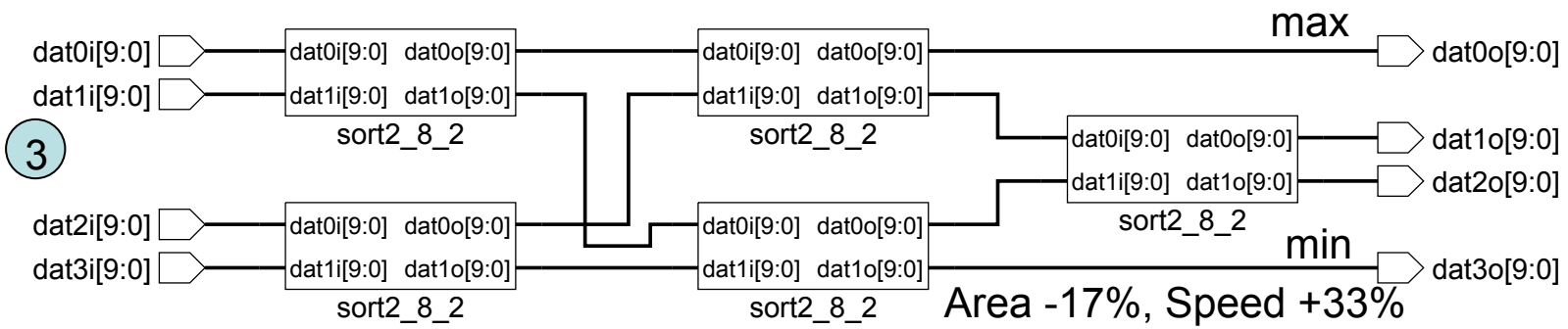
```
entity sort2 is
generic(Nbits : Natural := 8; Nidx  : Natural := 2);
port(
    dat0i : in  std_logic_vector(Nbits+Nidx-1 downto 0);
    dat1i : in  std_logic_vector(Nbits+Nidx-1 downto 0);
    dat0o : out std_logic_vector(Nbits+Nidx-1 downto 0);
    dat1o : out std_logic_vector(Nbits+Nidx-1 downto 0));
end sort2;
architecture a of sort2 is
signal d0gtd1 : std_logic;
signal dat0cmp, dat1cmp : std_logic_vector(Nbits-1 downto 0);
begin
    dat0cmp <= dat0i(Nbits+Nidx-1 downto Nidx);
    dat1cmp <= dat1i(Nbits+Nidx-1 downto Nidx);
    d0gtd1  <= '1' when dat0cmp > dat1cmp else '0';
    dat0o   <= dat0i when d0gtd1 = '1' else dat1i;
    dat1o   <= dat1i when d0gtd1 = '1' else dat0i;
end;
```



Sort4 – some better architectures

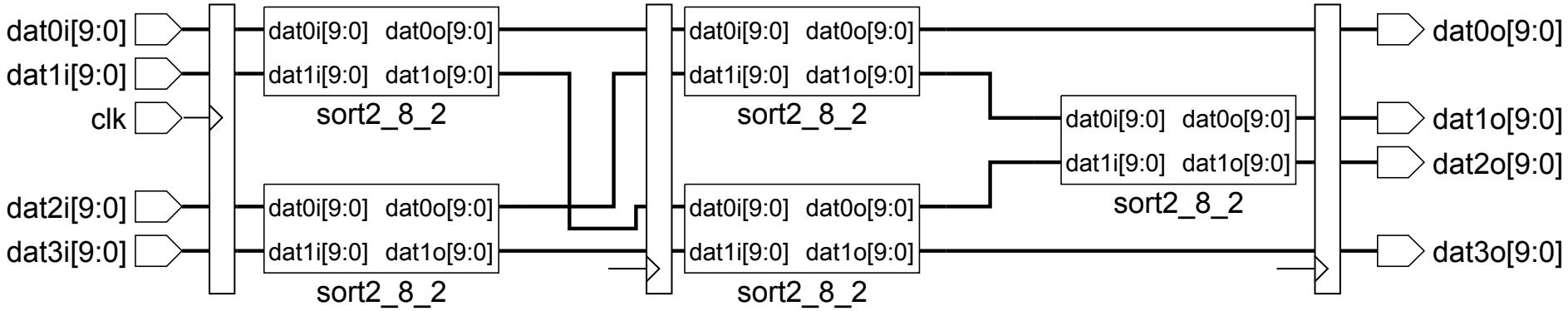




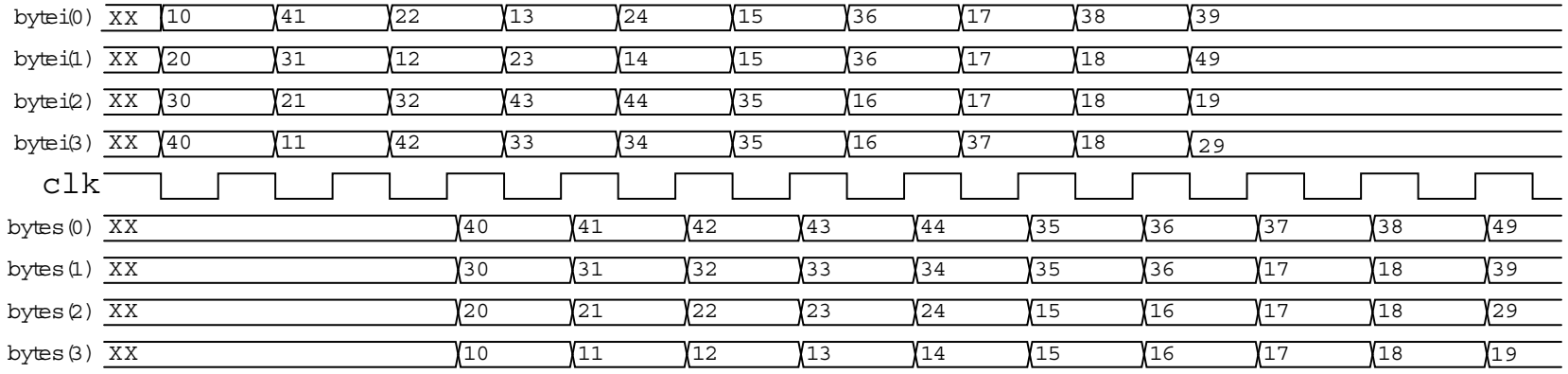


Sort4 - pipelining

In order to estimate easier the speed, we can first put registers at the input and at the output

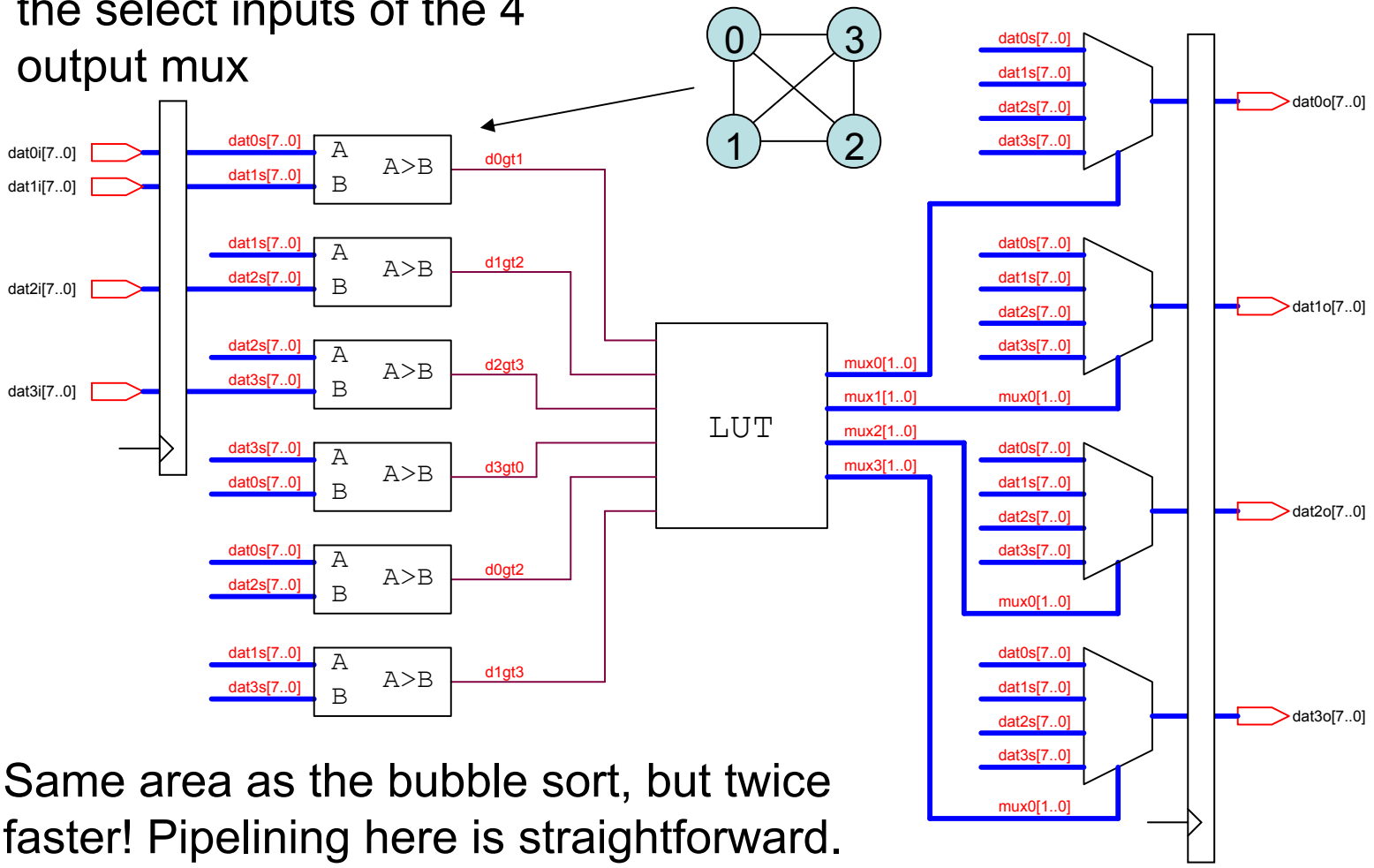


The next natural step would be to put one register in the middle.
We pay with latency, but we get the design running faster (+56%).



Sort4 – the best solution(1)

Compare each pair, put the result (6 bits) into a LUT and calculate the select inputs of the 4 output mux



Same area as the bubble sort, but twice faster! Pipelining here is straightforward.

Sort4 – the best solution(2)

The LUT can be realized as a **process** with a big **case** (in total 24 cases, the rest are don't care). The generation of the VHDL **case** code can be done by a simple C++ program.

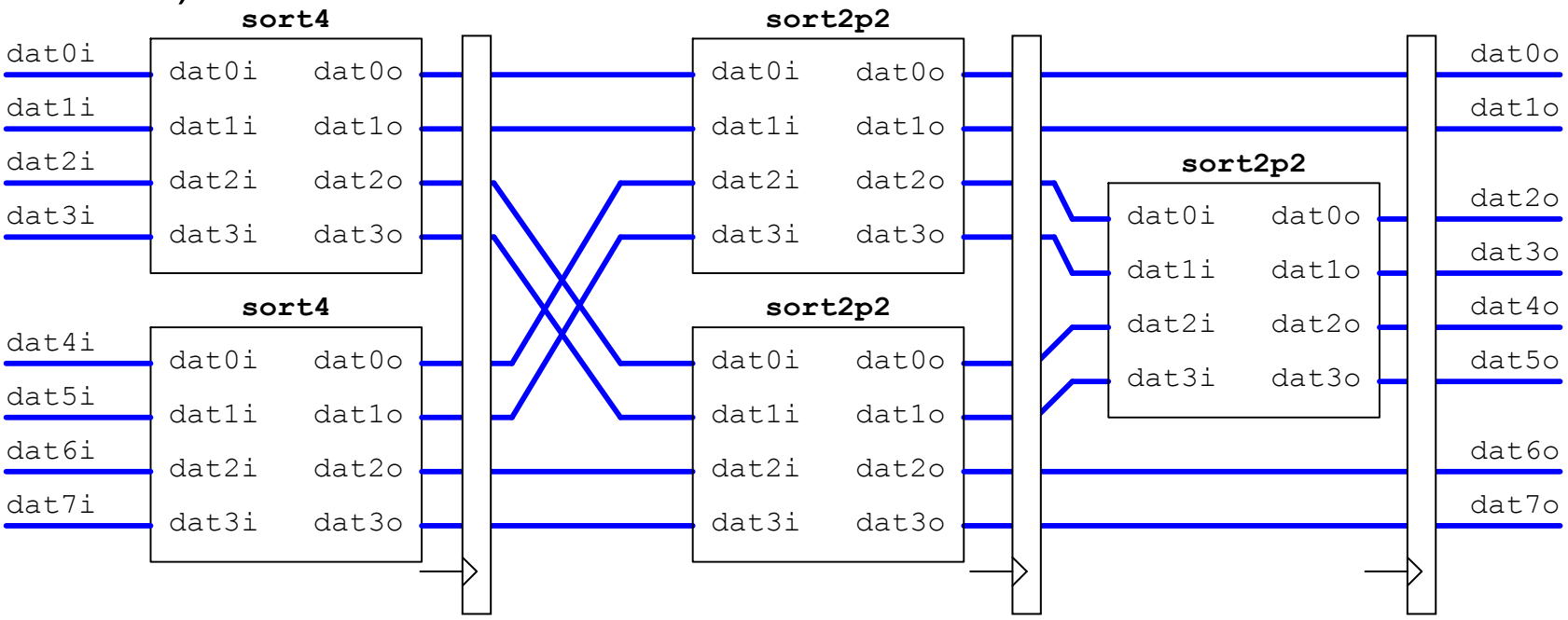
```
#include <iostream>
#include <iomanip>
using namespace std;
int pos(int p, int a, int b, int c, int d)
{
 if (a==p) return 0x00;
 if (b==p) return 0x01;
 if (c==p) return 0x10;
 if (d==p) return 0x11;
 return 0;
}
int main(void)
{
 int a,b,c,d;
 for(a=0; a<4; a++)
  for(b=0; b<4; b++)
   for(c=0; c<4; c++)
    for(d=0; d<4; d++)
     if ((a!=b) && (b!=c) && (c!=d) && (a!=c) && (a!=d) && (b!=d))
     {
      cout<<"when "<<'"'<<dec<<(a>b)<<(b>c)<<(c>d)<<(d>a)<<(a>c)<<(b>d)<<'"'<<"=>";
      cout<<"mux(0)<="<<'"'<<hex<<setw(2)<<setfill('0')<<pos(3,a,b,c,d)<<'"'<<";";
      . . .
```

The LUT 6 → 8 bit in VHDL

```
sel <= d0gt1 & d1gt2 & d2gt3 & d3gt0 & d0gt2 & d1gt3;
process(sel)
begin
 case sel is
  when "000100"=>mux(0)<="11";mux(1)<="10";mux(2)<="01";mux(3)<="00";
  when "001100"=>mux(0)<="10";mux(1)<="11";mux(2)<="01";mux(3)<="00";
  ...
  when others  =>mux(0)<="--";mux(1)<="--";mux(2)<="--";mux(3)<="--";
 end case;
end process;
```

Sorting larger arrays - sort8

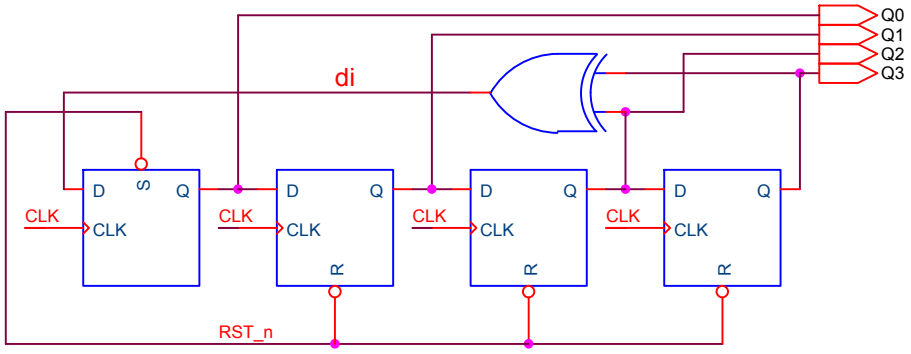
How to extend the sort entity for larger N? For N=8 the simplest solution is shown below, using 5 x sort4 entities, 3 of them slightly simplified (2 of the 6 comparators in the modified sort2p2 are not necessary as the two input pairs are already sorted, the LUT becomes smaller)



Again, pipelining here is straightforward. Generally this approach can be used to double the N, the logic explodes by factor of 4 to 5.

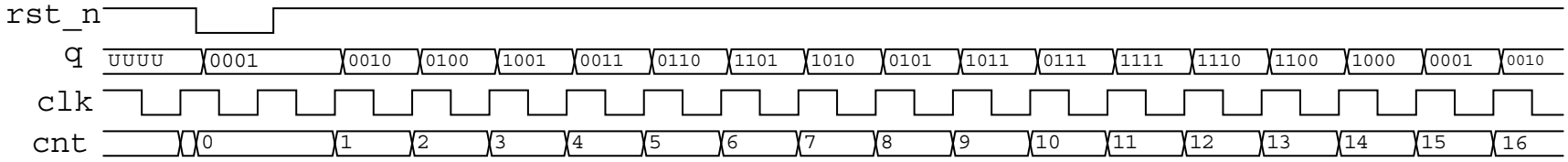
Pseudorandom generators(1)

N-bit shift register with a feedback, used to generate pseudorandom numbers. Note that the number of possible states is 2^N-1, not 2^N! The example below is for N=4.




```
di <= sr(3) xor sr(2); -- or psrg(sr)
process(clk, rst_n)
begin
 if rst_n = '0' then
   sr    <= (others => '0');
   sr(0) <= '1';
 elsif clk'event and clk='1' then
   sr <= sr(sr'high-1 downto 0) & di;
 end if;
end process;
q <= sr;
```

(next slide)



Used: for built-in tests of memories; to generate input data in testbenches; for Monte Carlo simulations; as simple frequency divider

Pseudorandom generators(2)

```
function psrg(sr : std_logic_vector) return std_logic is
variable bit0 : std_logic;
begin
 case sr'length is
  when 32 => bit0 := sr(31) xor sr(21) xor sr( 1) xor sr( 0);
  when 31 => bit0 := sr(30) xor sr(27);
  when 30 => bit0 := sr(29) xor sr( 5) xor sr( 3) xor sr( 0);
  when 29 => bit0 := sr(28) xor sr(26);
  when 28 => bit0 := sr(27) xor sr(24);
  when 27 => bit0 := sr(26) xor sr( 4) xor sr( 1) xor sr( 0);
  when 26 => bit0 := sr(25) xor sr( 5) xor sr( 1) xor sr( 0);
  when 25 => bit0 := sr(24) xor sr(21);
  when 24 => bit0 := sr(23) xor sr(22) xor sr(21) xor sr(16);
  when 23 => bit0 := sr(22) xor sr(17);
  when 22 => bit0 := sr(21) xor sr(20);
  when 21 => bit0 := sr(20) xor sr(18);
  when 20 => bit0 := sr(19) xor sr(16);
  when 19 => bit0 := sr(18) xor sr( 5) xor sr( 1) xor sr( 0);
  when 18 => bit0 := sr(17) xor sr(10);
  when 17 => bit0 := sr(16) xor sr(13);
  when 16 => bit0 := sr(15) xor sr(14) xor sr(12) xor sr( 3);
  when 14 => bit0 := sr(13) xor sr( 4) xor sr( 2) xor sr( 0);
  when 13 => bit0 := sr(12) xor sr( 3) xor sr( 2) xor sr( 0);
  when 12 => bit0 := sr(11) xor sr( 5) xor sr( 3) xor sr( 0);
  when 11 => bit0 := sr(10) xor sr( 8);
  when 10 => bit0 := sr( 9) xor sr( 6);
  when  9 => bit0 := sr( 8) xor sr( 4);
  when  8 => bit0 := sr( 7) xor sr( 5) xor sr( 4) xor sr( 3);
  when  5 => bit0 := sr( 4) xor sr( 2);
  when 15 | 7 | 6 | 4 | 3 =>
             bit0 := sr(sr'length-1) xor sr(sr'length-2);
  when others => bit0 := '-';
 end case;
 return bit0;
end;
```

The feedback expressions for some sizes of the shift register, based on:

Xilinx APP 052 July 7, 1996, ver. 1.1
- table for n=3...168

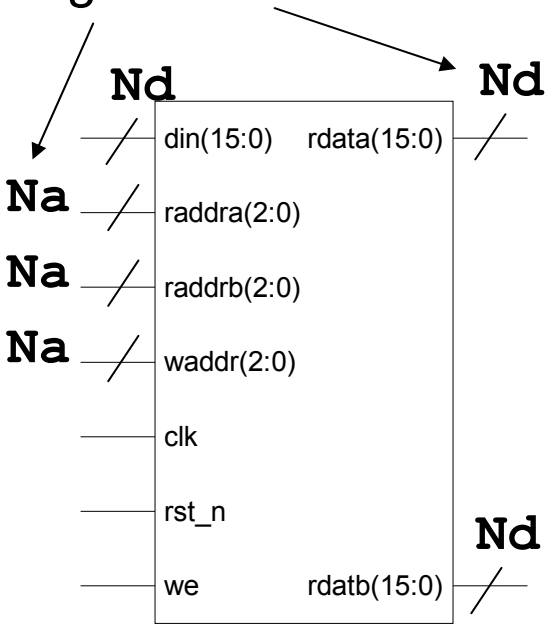
The generator can not exit the state "00..00"!

Registerfile

```
...
subtype reg_data is std_logic_vector(Nd-1 downto 0);
type rf_data_array is array(0 to 2**Na-1) of reg_data;
signal rf_data : rf_data_array;
signal we_i    : std_logic_vector(rf_data_array'range);
begin
 process(waddr, we) -- decoder with enable
 begin
     we_i <= (others => '0');
     we_i(conv_integer(waddr)) <= we;
 end process;
ri: for i in rf_data_array'range generate
 process(clk, rst_n) -- the registers
 begin
  if rst_n = '0' then rf_data(i) <= (others => '0');
  elsif clk'event and clk='1' then
   if we_i(i) = '1' then
       rf_data(i) <= din;
   end if;
  end if;
 end process;
end generate;
 -- the two output mux
 rdata <= rf_data(conv_integer(raddra));
 rdatb <= rf_data(conv_integer(raddrb));
end;
```

As a part of a RISC CPU core

Number and size of registers set through generics



State machines

State machines

- Two or three process implementation
- One process implementation
- Registered outputs and latency
- State encoding, stability
- Possible timing problems
- Partitioning of large machines
- Error protection

State machines in VHDL (1)

```
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;
               elsif L = '1' then next_st <= SL;
               elsif R = '1' then next_st <= SR;
               end if;
    when SL => if L = '0' and R = '1' then next_st <= SR;
                                      else next_st <= S0;
               end if;
    when SR => if L = '1' then next_st <= SL;
                          else next_st <= S0;
               end if;
    when SA => next_st <= S0;
    --when others => ...
    end case;
end process;
```

enumerated type

Next state as a function of the present state and the inputs

`when others` is not necessary in case of enumerated type

1st process

State machines in VHDL (2)

Register to store the current state

```
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;
```

2nd process

Calculate the outputs as a function of the current state (Moore machine):

```
process(present_st)
begin
    LL <= '0'; LR <= '0';
    case present_st is
    when SL => LL <= '1';
    when SR => LR <= '1';
    when SA => LL <= '1'; LR <= '1';
    when others => NULL;
    end case;
end process;
```

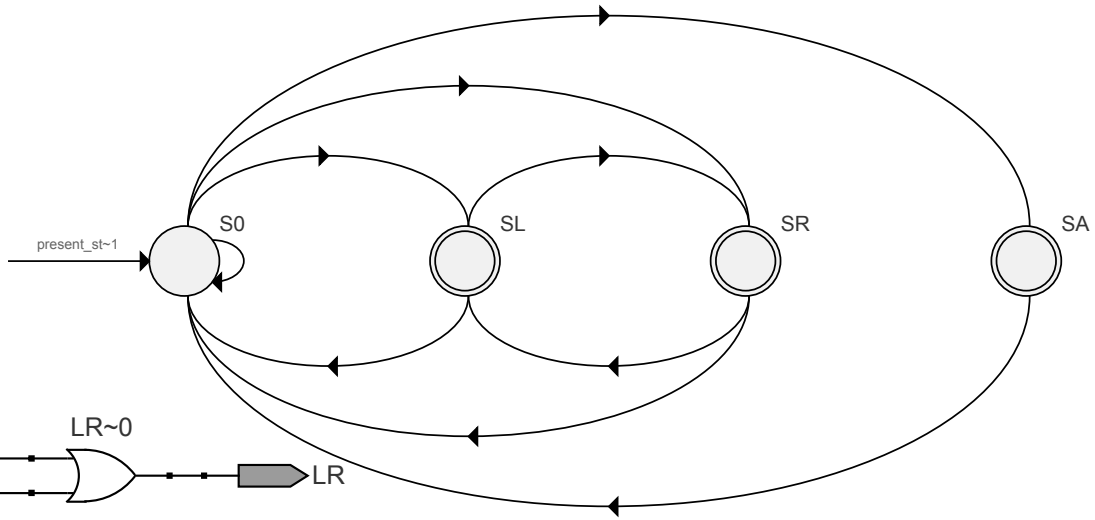
3rd process

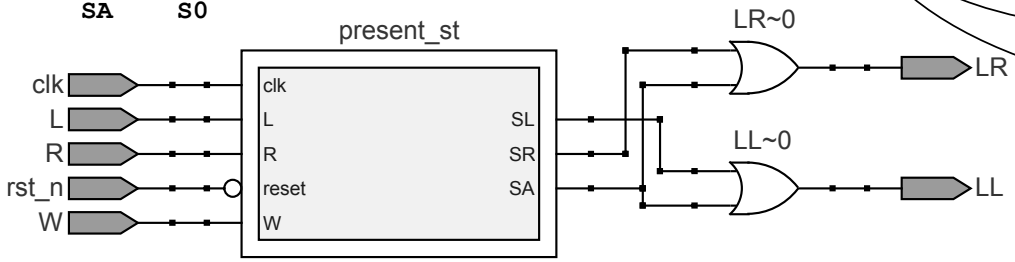
… or current state and inputs (for Mealy machines)

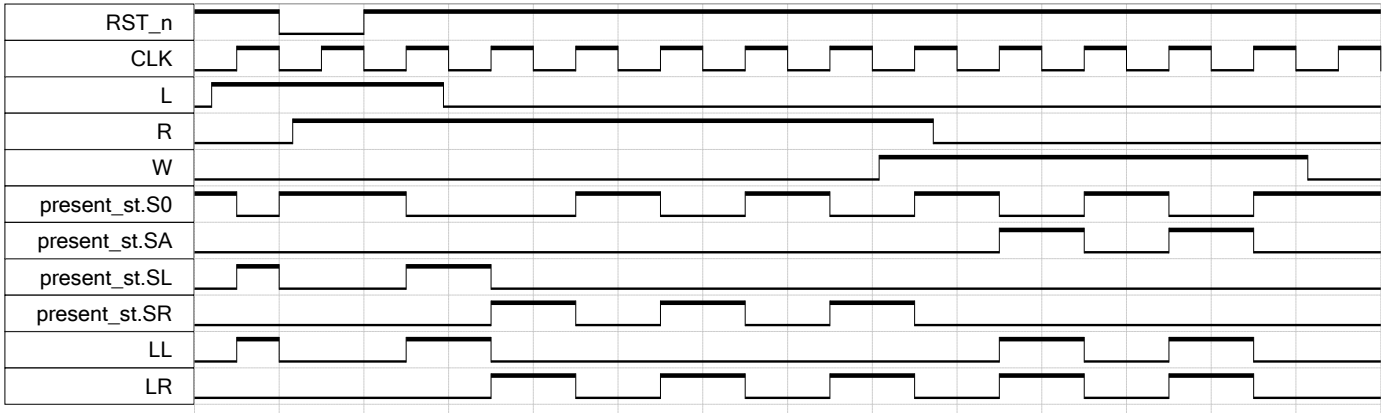
This process can be combined with the first one – two process description

State machines in VHDL (3)

State Source	Destination	Condition
S0	S0	(!L).(!R).(!W)
S0	SL	(L).(!W)
S0	SR	(R).(!L).(!W)
S0	SA	(W)
SL	S0	(!L).(!R)+(L)
SL	SR	(R).(!L)
SR	S0	(!L)
SR	SL	(L)
SA	S0	







State machines in VHDL – one process description

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

Description in one process only

this line depends on the encoding!

The outputs are part of the state coding in this example

- For small machines seems to be the shortest description
- Not recommended for large state machines!

State machines in VHDL – manual encoding

```
subtype state_type is std_logic_vector(1 downto 0);
constant S0 : state_type := "00";
constant SL : state_type := "10";
constant SR : state_type := "01";
constant SA : state_type := "11";
signal present_st, next_st : state_type;
```

← This works always!

OR

```
type state_type is (S0, SL, SR, SA);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of state_type : type is "0001 0010 0100 1000";
```

one-hot

OR

```
attribute ENUM_ENCODING of state_type : type is "00 01 10 11";
```

binary

```
signal present_st, next_st : state_type;
```

The exact way to set the encoding in the VHDL code depends on the software tool used, this example is for ISE (Xilinx)

The encoding style can be set globally for all entities (tool specific)

State machines in VHDL - registered outputs

It is often better to have registered outputs of the state machine. The simplest way is shown below, but this adds latency:

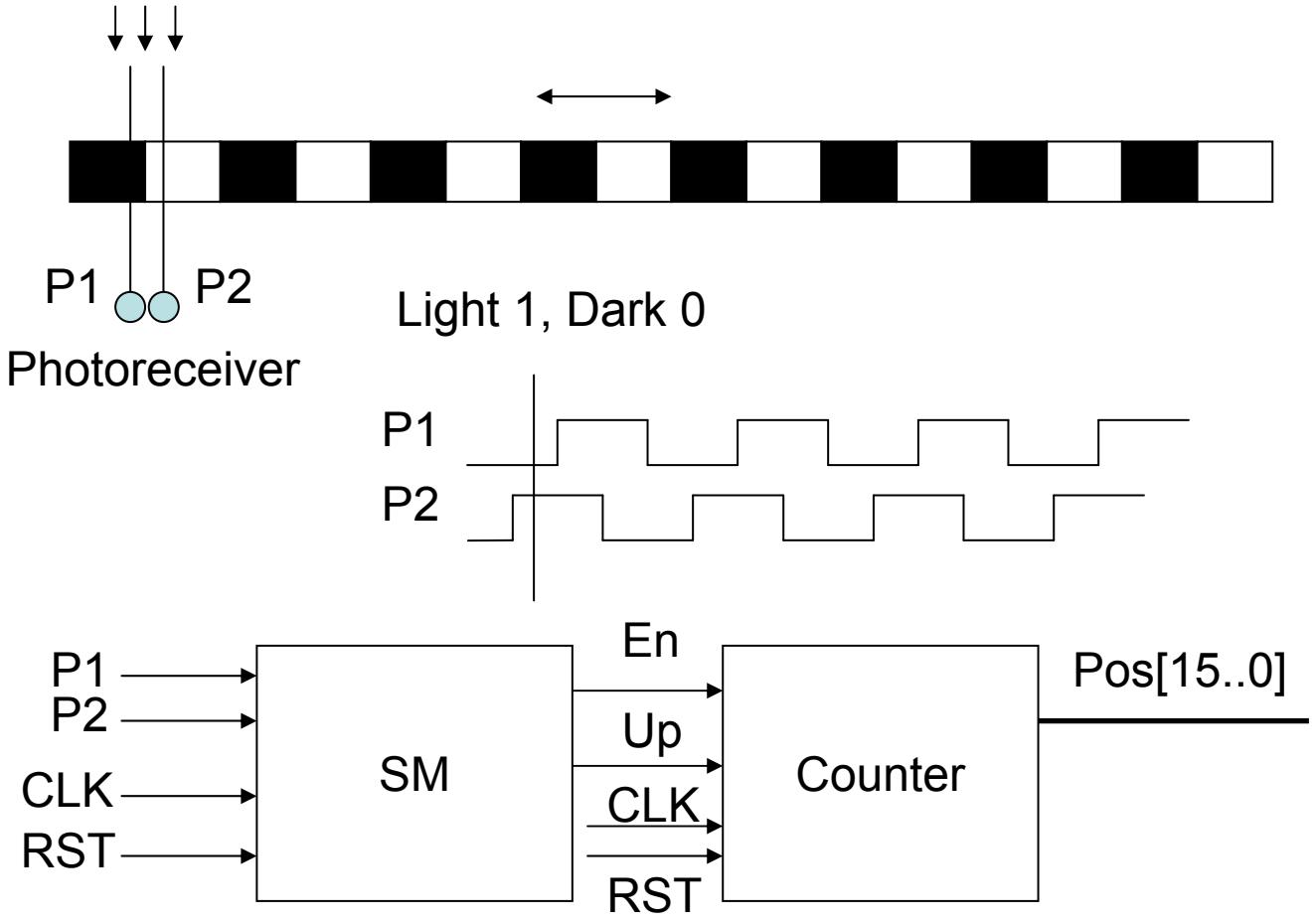
```
process(clk, rst_n)
begin
    if rst_n = '0' then
        present_st <= S0; LL <= '0'; LR <= '0';
    elsif clk'event and clk='1' then
        present_st <= next_st;
        if present_st = SL or present_st = SA then LL <= '1';
                                              else LL <= '0'; end if;
        if present_st = SR or present_st = SA then LR <= '1';
                                              else LR <= '0'; end if;
    end if;
end process;
```

How to avoid this extra latency?

Use the **`next_st`** instead of the **`present_st`**:

```
        if next_st = SL or next_st = SA then LL <= '1';
                                        else LL <= '0'; end if;
        if next_st = SR or next_st = SA then LR <= '1';
                                        else LR <= '0'; end if;
```

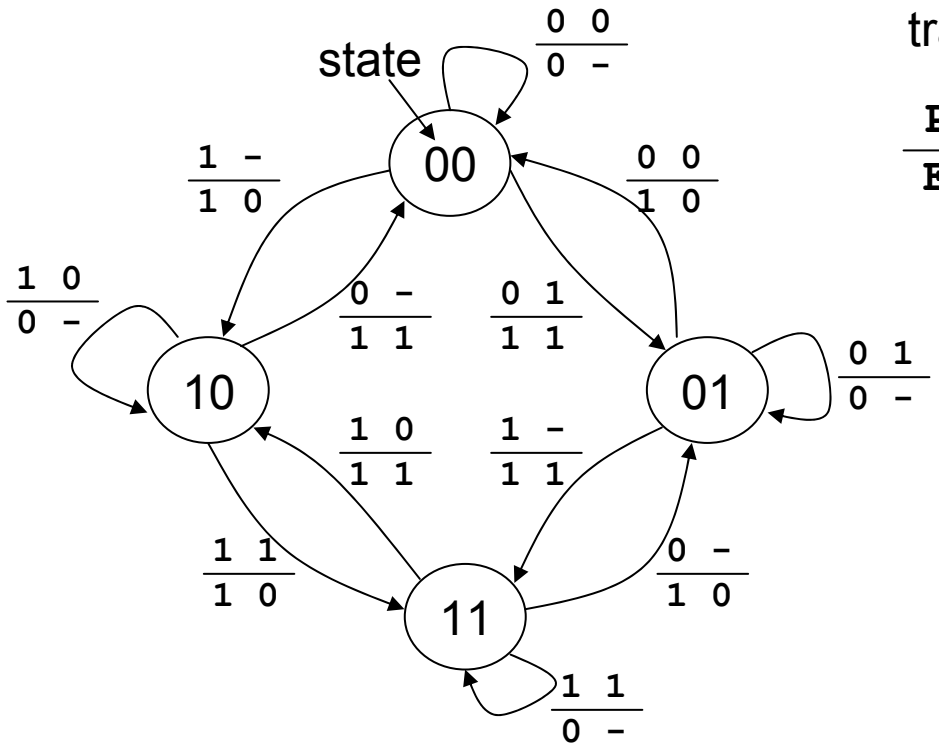
State machines in VHDL – position decoder(1)



Note: CLK is enough faster than P1, 2

State machines in VHDL – position decoder(2)

This is a good example how a Mealy type of machine can significantly reduce the number of the states



transition

P1	P2	inputs
EN	UP	outputs

The outputs are active only while jumping to another state.
How many states are necessary for a Moore machine ?

State machines in VHDL – position decoder(3)

```
type state_type is (S00, S01, S11, S10);
signal present_st, next_st : state_type;
signal p1s, p2s : std_logic;
...
process(present_st, p1s, p2s)
begin
    EN <= '0'; UP <= '-'; -- the default is no counting
    next_st <= present_st;
    case present_st is
    when S00 => if   p1s = '1' then next_st <= S10; EN <= '1'; UP <= '0';
                elsif p2s = '1' then next_st <= S01; EN <= '1'; UP <= '1';
                end if;
    when S01 => if   p1s = '1' then next_st <= S11; EN <= '1'; UP <= '1';
                elsif p2s = '0' then next_st <= S00; EN <= '1'; UP <= '0';
                end if;
    when S11 => if   p1s = '0' then next_st <= S01; EN <= '1'; UP <= '0';
                elsif p2s = '0' then next_st <= S10; EN <= '1'; UP <= '1';
                end if;
    when S10 => if   p1s = '0' then next_st <= S00; EN <= '1'; UP <= '1';
                elsif p2s = '1' then next_st <= S11; EN <= '1'; UP <= '0';
                end if;
    end case;
end process;
```

this process calculates the next state and the outputs

State machines in VHDL – position decoder(4)

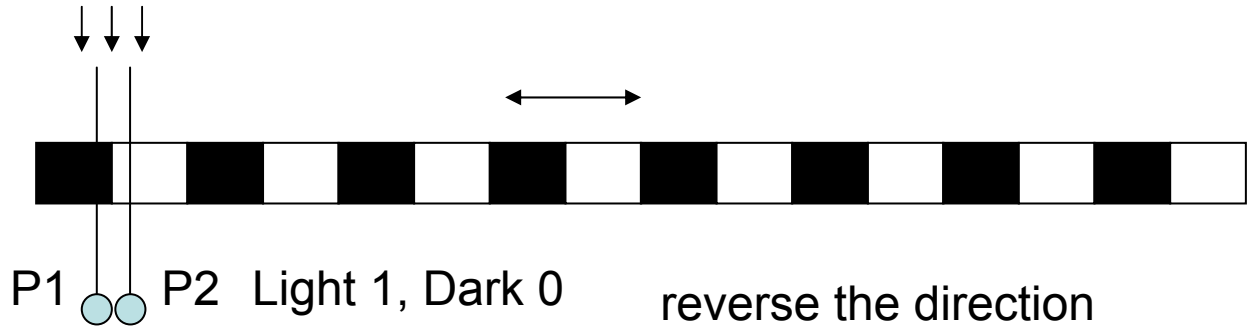
```
process(clk)
variable p12s : std_logic_vector(1 downto 0);
begin
    if clk'event and clk='1' then
        p1s <= P1; -- synchronize the inputs
        p2s <= P2; -- to the Mealy machine!
        p12s := p1s & p2s;
        if rst_n = '0' then -- jump to the correct state
            case p12s is
            when "00" => present_st <= S00;
            when "01" => present_st <= S01;
            when "10" => present_st <= S10;
            when "11" => present_st <= S11;
            when others => NULL;
            end case;
        else
            present_st <= next_st; -- store the next state
        end if;
    end if;
end process;
```

this is very important!

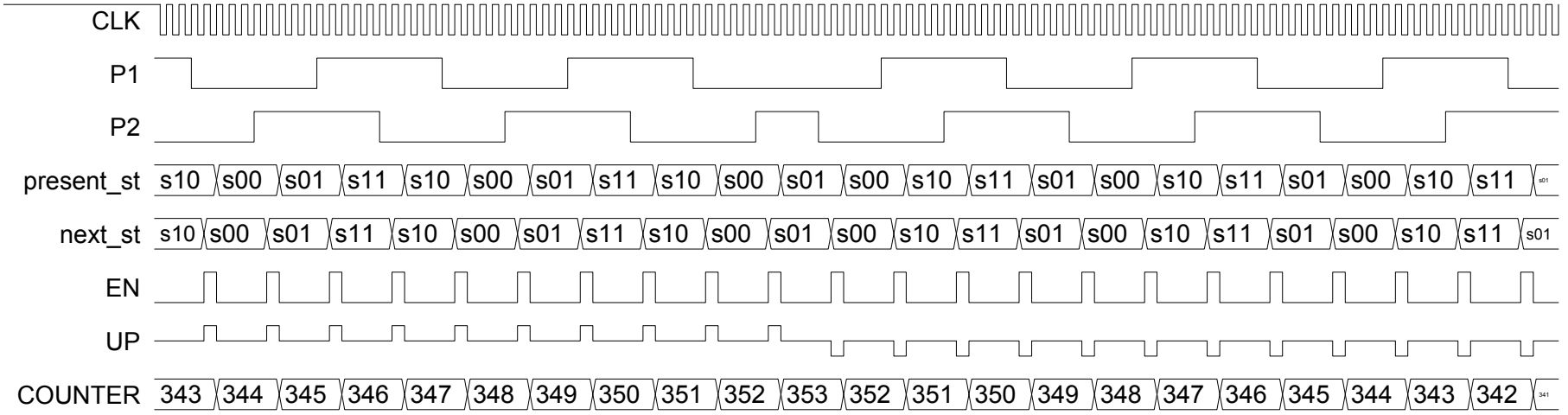
this is no register!

a somehow unusual reset, asynchronous would be dirty

State machines in VHDL – position decoder(5)



Simulation



State machine - encoding

- One-hot
  - + fast decoding
  - - many illegal states

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

5 valid, 27 illegal combinations

- Binary, Gray
  - + without or with only a few (< 50%) illegal states
  - - more complex decoding

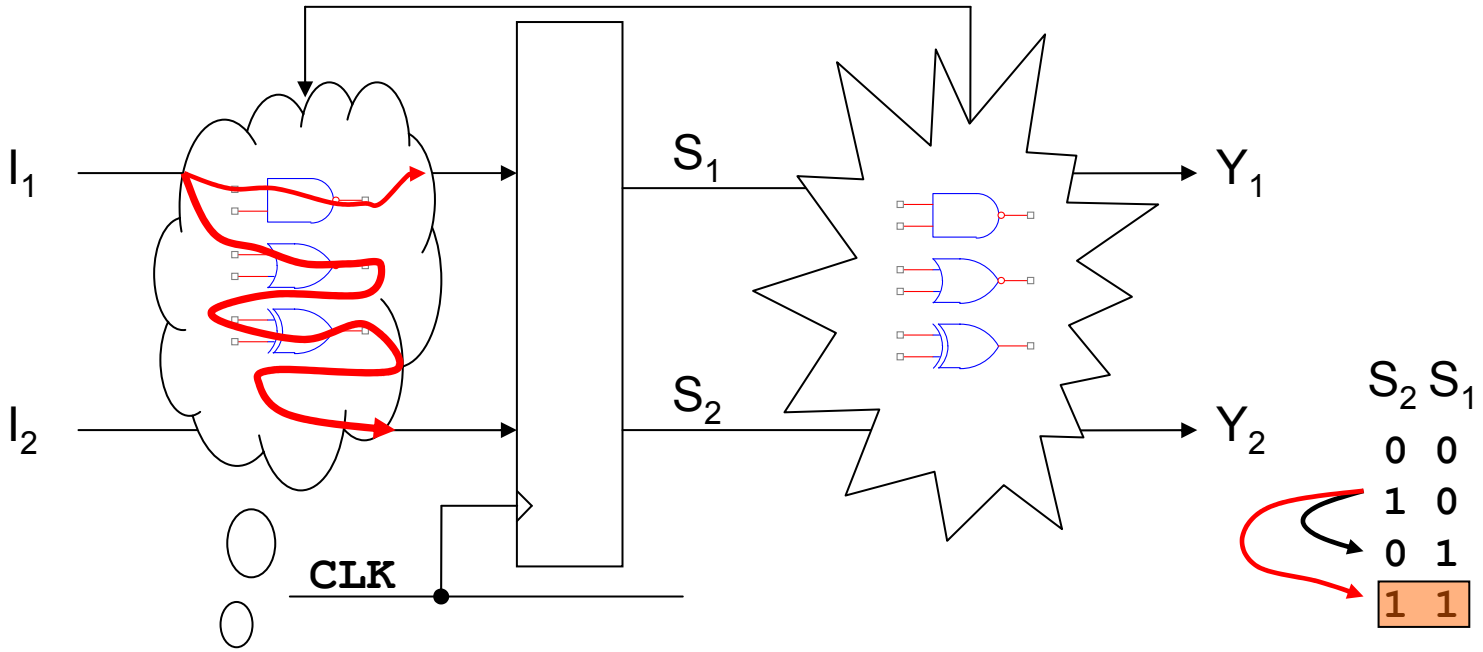
```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

- Using the outputs + eventually additional bits
  - + Synchronous outputs
  - + The decoding is not necessary
  - - Eventually more logic

State machine – illegal states

- In case of 5 states there are 32 possible states in one-hot and 8 in binary coding
- Depending on the realization of the next state logic, once entering an illegal state the state machine may never recover to a normal state
- The standard `when others => sm <= ...` in the description of the state machine doesn't guarantee anything in most cases!
- Only when encoding the states manually, `when others =>` could help (or when activating **safe implementation** option in the synthesis software if available)
- How is it possible for a state machine to enter some wrong state? How to avoid this?

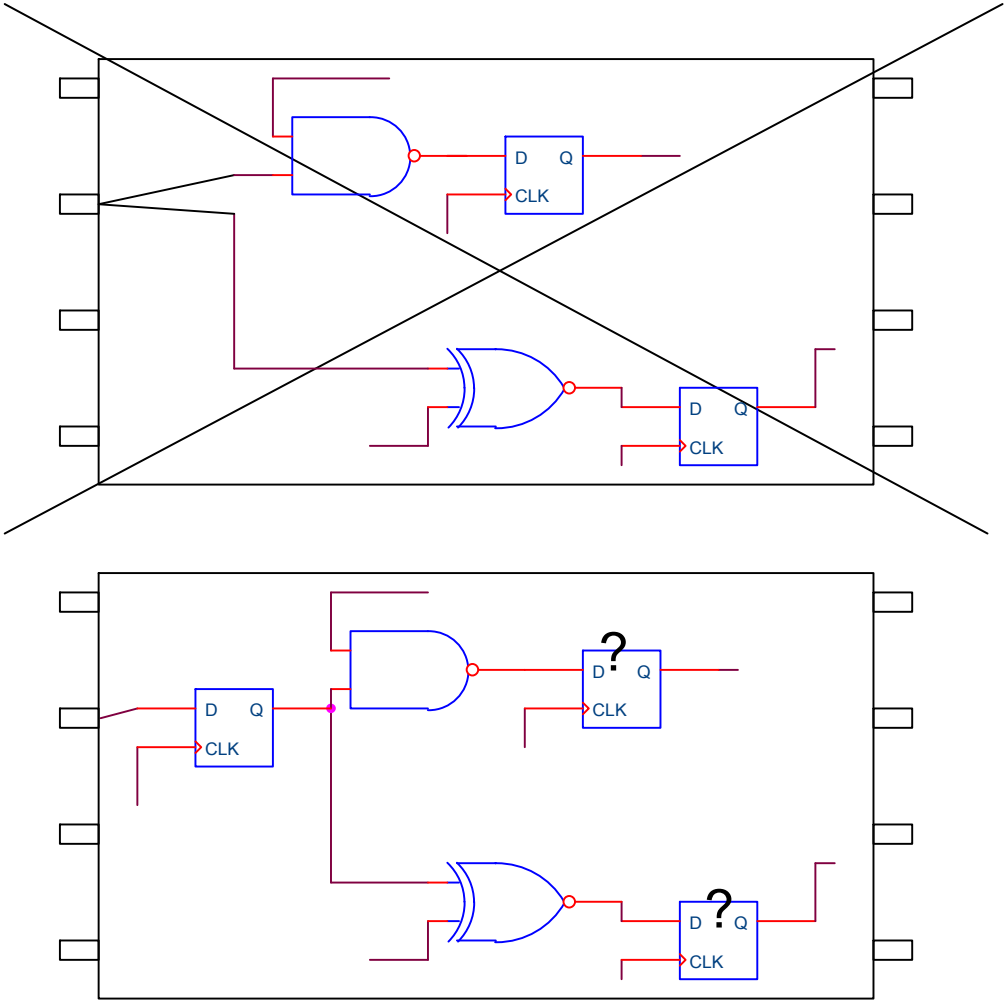
State machine – setup/hold



The propagation delays from the input **I1** to **S1** and **S2** are different. If the input signal is asynchronous to the **CLK**, it can happen that **S1** stores the new, but **S2** the old value, as a consequence the state machine jumps to a totally wrong state (different from the old and the next one), which could be one of the illegal states

→ **All inputs to a state machine must be synchronized to the CLK!!!**

Synchronize input signals



… before using them in the chip

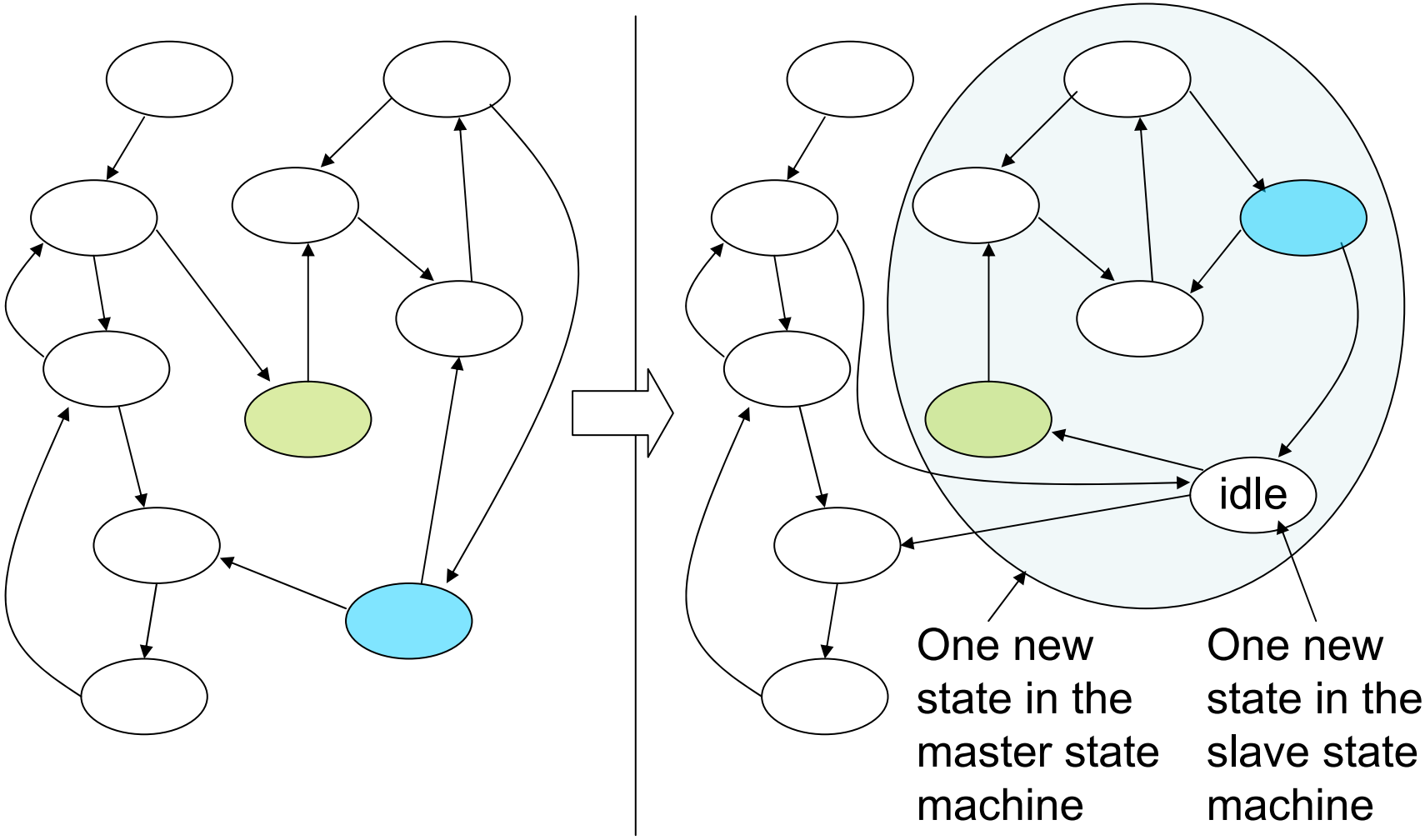
General recommendation, not only for the inputs to the state machines

Due to the different delays, the input signal will be interpreted by a part of the chip as 1, by another part as 0

Partitioning of state machines (1)

- How to encode large state machines? E.g. with 33 states, then when using
  - one-hot encoding we need 33 bits and have about 8.10^9 unused states!
  - when using binary or Gray encoding we have 64 total states
  - Can we do something more?
- YES!
  - first try to minimize the number of the states
  - partition the state machine into several smaller

Partitioning of state machines (2)



Partitioning of state machines (3)

- Similar strategy as when partitioning a large design into several subdesigns (entities) – hierarchy!
- Try to find the independent (orthogonal) parts of the state machine with single transitions at the boundary
- Extract these parts as slave state machines, eventually put some additional states in the master and in the slave machines
- When the master reaches the entry state to the slave machine, it remains there until the slave has finished
- For the slave all other master states act as synchronous reset
- With some care the extra latency when making a transition from one to another machine can be avoided

Error protection (1)

- Because of electrical noise or ionizing radiation it is possible that single bits toggle
- With a Hamming-Code single bit errors can be corrected, by adding control bits to the data word (e.g. for 4 bits 3 control bits are needed, for 32 bits – 6)
- If the register holding the **state of a state machine** has additional Hamming control bits and the coding/decoding logic, a single bit flip in the state register will be corrected at the next clock

Error protection (2)

- The principle of operation of the Hamming-Code is simple:
  - The Hamming distance between two words is defined as the number of differences in all bit positions, e.g. **0**01**0**1 and **1**01**1**1 have Hamming distance of 2
  - The set of the possible data words is extended, but the valid data words remain a small subset, so that the min Hamming distance between any two valid words is e.g. 3
  - In case of one bit error the data word is corrected to the nearest valid data word
  - By increasing the distance, more detection/correction can be achieved