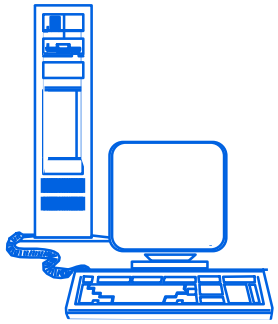
VHDL

Digital Systems



«The designer’s guide to VHDL»
Peter J. Andersen
Morgan Kaufman Publisher

Bring laptop with installed Xilinx

HDL Introduction

- VHDL (DoD project) and Verilog (private project)
- High level languages for digital systems simulation but almost always used for the synthesy
- The target is to make the design more effective (development time and costs reduction, big and complex systems design etc.)
- HDL languages allow to define the hardware behaviour (intrinsic parallelism and delays)
    - In this course only a *small portion* of the language will be presented, sometimes with some inaccuracies for a simplified FPGA design
- VHDL - Very High-level Design Language
- Used for instance for FPGA (Field Programmable Gate Arrays) and ASIC (Application Specific Integrated Circuits) design and test – (*Processors are ASIC)*

VHDL

- **VHDL is a language that uses the *structures* of C language with all modifications required by the specific context**

- **In this course the Xilinx Vivado software will be used**

FPGA vs traditional hardware design

The FPGA e HDL modern technologies allow to reduce dramatically the *time to market*: the *complete* prototype development can be implemented on a computer.

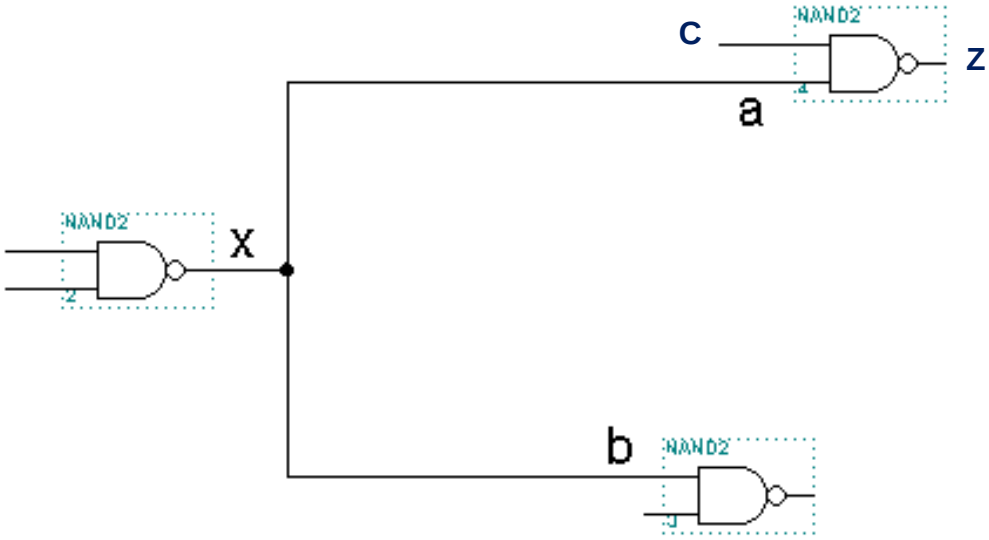
The production cost of a FPGA design can be more expensive of an ASIC implementation. But being VHDL a standard, an ASIC based on the same VHDL program used for the FPGA prototype is always possible.

A FPGA can be field reconfigured (a new program) in case of bugs or for improved functionality

Timing and Concurrency

Signals propagation uses wires and is NOT istantaneous because of the physical characteristics of conductors/components (parasite phenomena, gates delays, etc)

Consider for instance the following logical network:



In order to describe the signal z propagation with a high level language (i.e. C) we could write

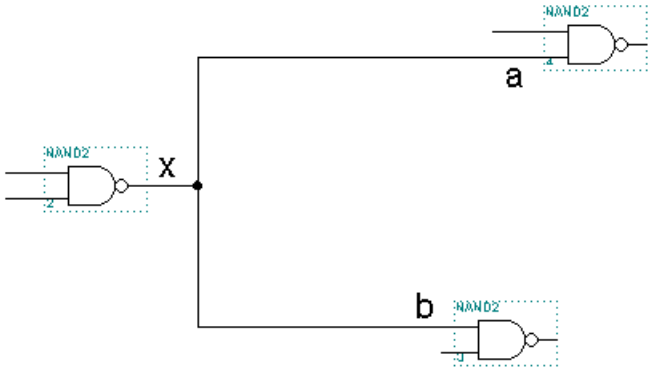
```
a <=x;          -- assign x to a
z<=a and c; -- assign z
```

With these two statements it seems that x=a e z=ac occur at the same time!!!! False!! z will change after the NAND delay

Timing and Concurrency

- **CONCURRENCY: in the traditional programming languages two assigning instructions must be executed one after the other with the same *sequence* of the program. But physically the electric signal x propagates concurrently towards a and b (NOT towards a and *then* towards c as the previous C program would implicate).**

- **TIMING: the two assignment statements do not take into consideration the signal propagation delay (never zero !!) and the gates delay**



Real network: different wires lenghts imply that a and b are not simultaneous. In the FPGA design systems the delays depend on the circuit technology and how the signals are «routed» within the integrated circuit

Timing and Concurrency

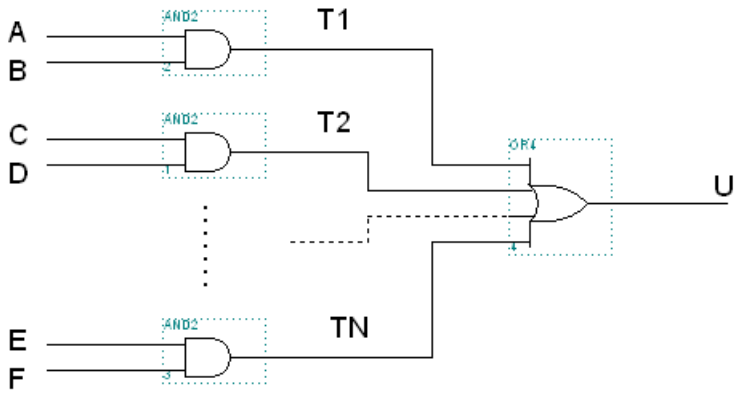
- When a programmer produces high level code (C/C++, Java, etc) he decomposes the problem into an instruction set which will be sequentially executed (*sequential programming paradigm*)..
- On the contrary a hardware designer decomposes the project into interconnected blocks which react to «events» and produce in turn «events»; the «events» are the signal transitions. This implies that all blocks whose behaviour depends on those events are parallel evaluated according to the new signals values. In VHDL the *statements* are *parallel* (simultaneously) executed as is the case for the real systems (all blocks of a system evolve in parallel *and not serially)*
- The blocks evaluation order must have *no ininfluence* on the final result (that is no matter which is the blocks evaluation order the overall status of the system once all events have been handled must be always the same).
- This programming uses the *parallel programming paradigm*. Since the result of the computation must be independent from the sequence of the statements execution all statements can be executed in parallel without any statement waiting for the end of another

Concurrency → The possibility of "executing" several operations in parallel, typical of the hardware behaviour

Timing → The capacity of modelling the propagation times within a circuit

An example

Let's analyse the following combinatorial ciruit and let's suppose that the delays (gates, wires etc) are identical. The code which describes the behaviour of this circuit must produce a result (the ouptut of each gate) which depends on the input only *and not on the order* of the AND and OR evaluation



The statements which describe the blocks can be coded according to the VHDL paradigm (<= indicates the signal modification):

```
T1 <=  A and B;
T2 <=  C and D;
TN <=  E and F;
U  <= T1 or T2 or T3;
```

Equivalent

```
U  <= T1 or T2 or T3;
T1 <=  A and B;
T2 <=  C and D;
TN <=  E and F;
```

VHDL Entities

In VHDL entities (objects) are available. Each one of them has a data type and a value (strongly typified language).

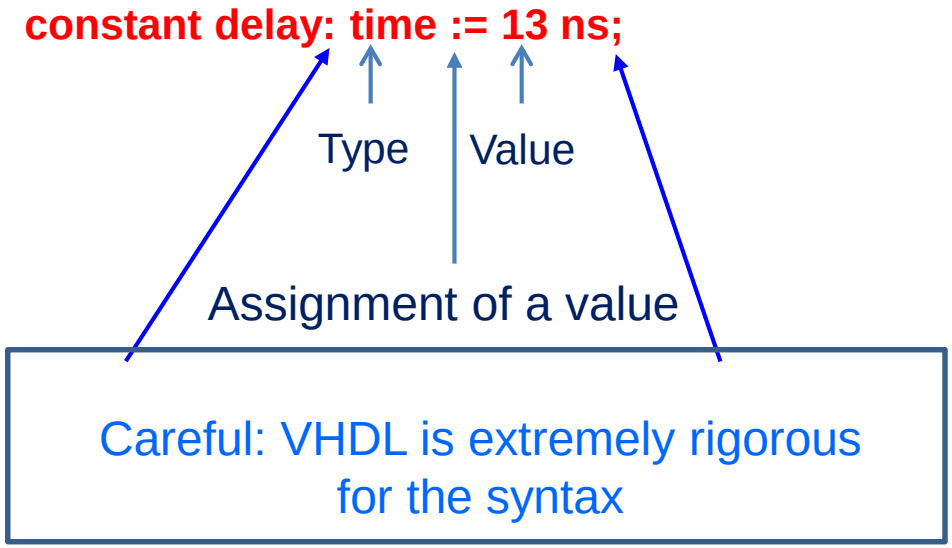
- **Constants**
- **Variables**
- **Signals**

Now we analyse how all these entities must be declared and how a value can be assigned to them

Constants

Entities which cannot change their values. Useful to make the code well readable

Careful: always use the symbol «;» as an end of a statement



A «constant» is always defined within an «architecture» (see later)

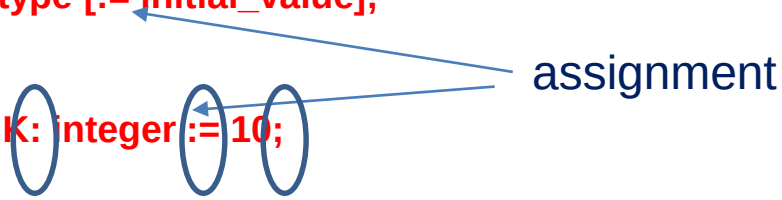
Variables

(NOT signals !!!)

Objects which can modify their values during the simulation. They can be declared only within a *process* (see later) and are local to the *process* (the initial value is optional)

variable name: type [:= initial_value];

Example



Usage example(IJK variabile) – [N.B «for» clause can be used only within a *process* (see later)]

for IJK in 7 downto 3 loop – execute *5 times (values 7 to 3 included) !!! Double «-» is a comment*

X(IJK) := Y(IJK); -- X and Y are variables vectors [arrays (see later)]

end loop; -- *all* loop statements are executed *concurrently*

NB a variable has *no hardware meaning* and is used *only* to define the execution flow of the program. IT IS NOT a signal

NB variables can be declared or have values to be assigned *only* within a *process* – *see later* – and are local *to that block*

The assignment is performed through the operator := and has immediate effect

Examples **z:= y; -- z and y variables (not signals!!)**

Signals

Physical entities (*signals, actually…)* which can modify their values during the simulation (an initial value is optional) *with a delay* depending either from the technology or from a synchronism signal (typically the clock signal – see later the synchronous circuits - process). A signal is defined within an «architecture» - (see later).

```
signal name: type [:= initial_value];

                Examples

signal A      : std_logic := '1';  -- N.B. single apex
signal B      : std_logic := '0';
signal C      : std_logic := '1';
C <= A and B;
```

Important!

N. B. The *assignment* is performed with the operator <= and is NOT IMMEDIATE (that is it happens after a *delay* depending on the technology and the nature of the circuit – sequential, combinatorial etc.). std_logic is a technology *type* of the signal which defines its electrical behaviour. It is the only type used in this course

Predefined data types in VHDL

In the VHDL language (Standard Package) the following data types are also defined:

- **std_logic**
- **bit ('0','1') -- not used in our context. *We use only std_logic***
- **boolean ('TRUE','FALSE')**
- **integer**
- **positive**
- **time**

N.B. The use of std_logic is *unavoidable* in Xilinx if a simulation must be executed when all networks and their interconnections are mapped onto a specific FPGA. The simulation accounts for the specific delays of the circuit

std_logic type

A std_logic signal can have logical values ‘0’ and ‘1’

It is possible to define *vectors* of signal

signal vector_example : std_logic_vector (7 downto 0) := “ 01001001“;

(definition) (name) (type) (size) (initial value – double apex)

For binary values of a single bit the single apex «’» symbol is used while for binary configurations of 2 or more bits the double apex «”» must be used. A hexadecimal notation can be used

Important! Double apex!!

Examples a <= ‘1’; vector_example <= “10011100”; vector_example <= x“9C” -– (hexadecimal notation)

```
for IJK in 4 to 10 loop     -- executes 7 times (values 4 and 10 included) !!! (IJK is an integer variable)
    A(IJK) <= B(IJK) ;      -- A and B are signal vectors (arrays)
end loop;                   -- all loop statements are executed at the same time
```

Integer type

Integer range depends on the platform and is [$-2^{31}-1$, $+2^{31}-1$].

Example

```
constant     data_bus_width:     integer  :=  32; --  integer constant of decimal value 32
(definition)       (name)            (type)      (value)
```

Positive type

Positive are integer numbers from 1 to $2^{31}-1$.

Time type

To the time type an attribute is associated (ms, ns, ps, etc). In Xilinx it can be used *only* for the simulations (testbench programs – see later) .

Esempio constant delay: time:= 5 ns;

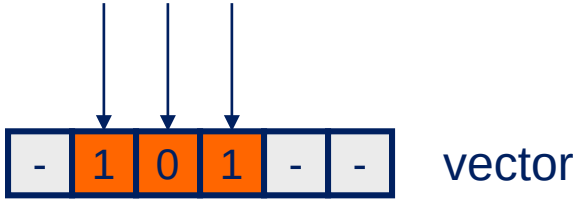
Using indexes the single elements of an array can be accessed. For instance

y(2) <= a(1)

To a set of contiguous elements a value can be assigned

y <= “01”;

y(4 downto 2) <= “101”



Chain *operator* &

& operator (concatenation) allows to define bit strings

Example: signal y <= “101” & “011” & ‘1’; -- the result is 1010111 – Comments in Xilinx are green

A:=“101”; -- constant or variable

B:= “011”;

C:= ‘1’;

y <= A&B&C; -- would have produced the same result

Chaining is NOT *a logical action* with physical meaning but a tool to express more clearly an expression by underlining its components

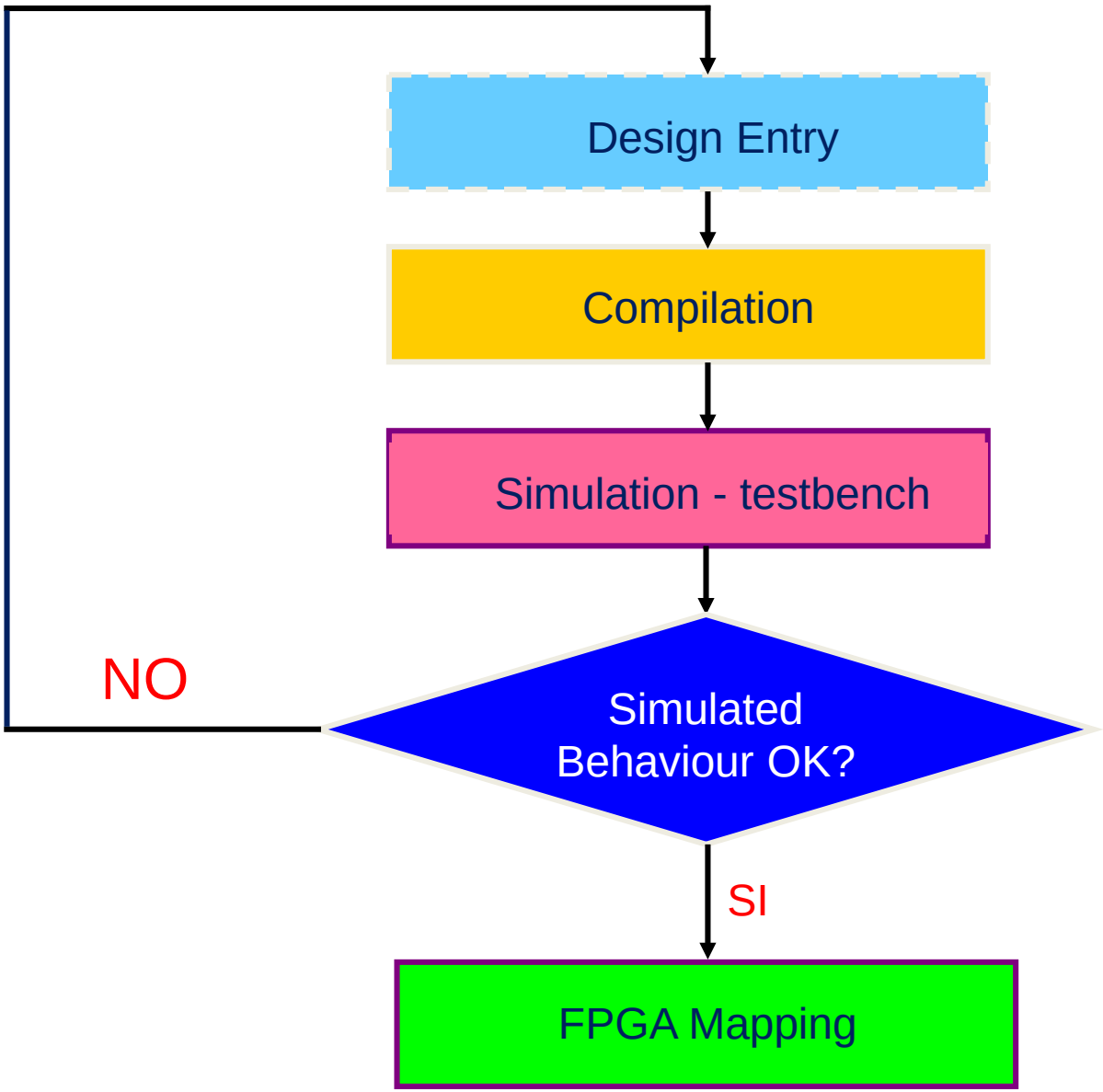
Logical *operators* in VHDL (signals and variables)

And:	and
Or:	or
Not:	not
Nand:	nand
Nor:	nor
Xor:	xor
Xnor:	xnor

- not operator is of highest priority (as in the logical expressions). The priority can be modified by means of the parenthesis (better to used them always)
- If vectors are used the number of involved bits must be the same.
- Statements examples (y,a,b,w,x,h,k signals):

y <= not a; z <= a and b; w <= x or (not y); k(8 downto 5) <= h(10 downto 7);

Design steps



Typical VHDL code structure (design - *not testbench*)

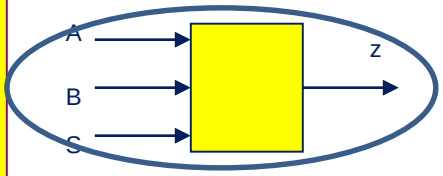
```
library IEEE;
use IEEE.std_logic_1164.all;  -- necessary when std_logic is used
```

Libraries declarations

```
-- Entity declaration

entity name_entity is

  Port     (signal_name : direction type);

end name_entity;
```

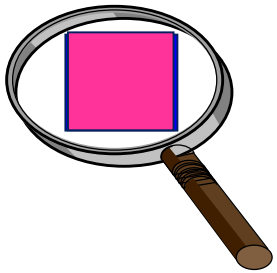
Interface specification

Entity




```
-- Architecture description

architecture architecture_entity of name_entity is

begin
                <processing statements>

end architecture_entity ;
```

Architectural specification



Interface specification (Entity)

```
entity example  is

                port ( sign_1     : inout   std_logic:='0'; ---- notice inout
                       sign_2     : in      std_logic_vector (7 downto 0) := "01001001";
                       enable     :  in     std_logic:='0';
                       ck         : in      std_logic:='0';
                       z          : out     std_logic:='0' -- last element -  no semicomma
                        );
end example;
```

«Entity»: defines the network «externally visible» input and output signals

Port allows to specify:

- **signal_names: the network signals**
- **direction: input (in), output (out) , bidirectional (inout) signals which are used within the architecture**
- **type: the signal type (*std_logic* or *std_logic_vector*)**

Architectural specification (Architecture)

```
architecture Behavioural of Example is

begin
        <processing statements>
end Behavioural ;
```

Within the architecture section the network logic (behaviour) is specified

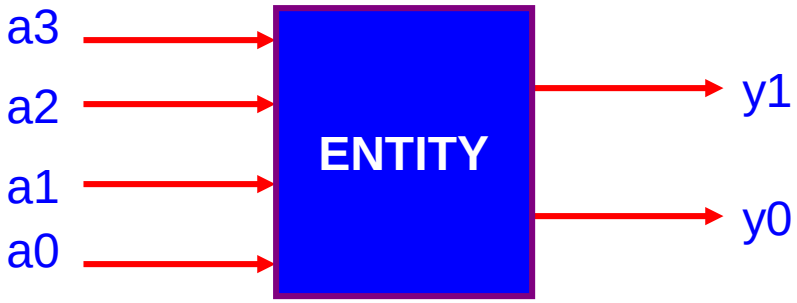
Within the architecture (in the <*architecture declarations*>) it is possible to define objects. They are typically signals (signals internal to the architecture in addition to those externally defined in the entity) which can be used (*scope*) only between the clauses *«architecture» and «end»)*

```
signal T1,T0 : std_logic;
signal T3,T4 : std_logic;
```

It must be noticed that the signal declaration has a form different from that implicit in Port where the *direction* is mandatory.

The section within which the network logic is defined is between begin and end.

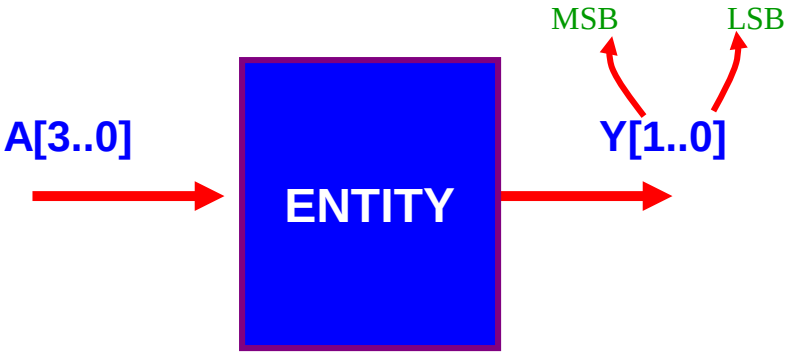
Scalar form




```
port ( a3,a2,a1,a0 :      in      std_logic;
       y1,y0 :            out     std_logic);
```

Notice the position of the character ; and of the parenthesis

Vector form (bus) - downto




```
port ( a :   in   std_logic_vector    (3 downto 0);
       y :   out  std_logic_vector    (1 downto 0));
```

Example - Hello VHDL (a two inputs AND device)

```
library ieee;
use ieee.std_logic_1164.all;
-- first example of vhdl code
-- a simple AND  circuit

-- interface specification

entity file_vhdl is
        port (  x :     in    std_logic;
                y :     in    std_logic;
                z :     out   std_logic);
end file_vhdl;


-- structure description

architecture behavioural of file_vhdl is

begin
        z <= x and y;
end behavioural;
```

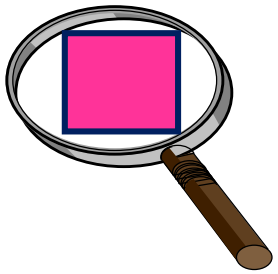
-- indicates comments

Network input/output signals

Interface specification
(black box - entity)

Blue colour: "reserved words"

Architecture:
network behaviour



Network internal structure

- **Comments are preceded by - - and in Xilinx are green**
- **VHDL language is not *case sensitive***
- **No space in the names of signals, variables etc.**
- **Names starting or ending with "_" are not allowed**
- **Names starting with a number are not allowed**
- **Names including "-" are not allowed**
- **Names including two consecutive "_" are not allowed**
- **Reserved words cannot be used for names (i.e. BIT, AND etc.)**
- **VHDL files extension should be .vhd**
- **No multilines comments are allowed (a «--» must be used for each line)**
- **All instructions end with ";"**
- ***VHDL files are text only and can be edited with any text editor***

Component

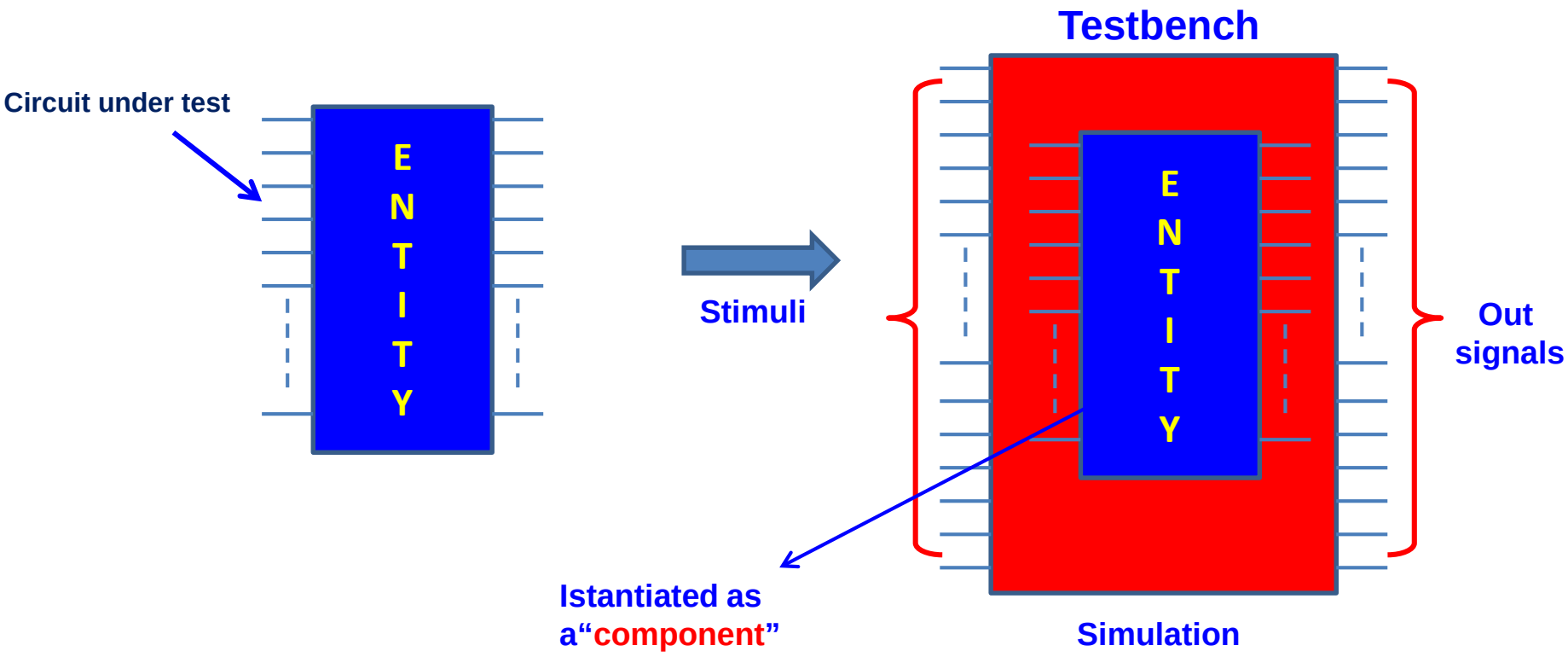
- **Each logical network, no matter how complex, can be shortly defined as a «component» (as is the case wih the integrated circuits)**

- **A «component» can be viewed as a «subroutine» of a language where the dummy input and output variables are defined in the PORT and the links to the higher level circuit are defined in the PORT MAP -- See next slide**

Testbench

A testbench is a specific VHDL code (a program for the simulation of the circuit under test) which allows the analysis of the behaviour of the circuit upon specific input stimuli.

A testbench can be schematized as follows



N.B. The network under test is istantiated with the command “component”. The circuit is in fact a component of the test system. In the FPGA design programs commands “after” and “wait” can be used only for the testbench stimuli while the system (entity) delays depend only on the technology of the chosen FPGA and on the placement and route. In Xilinx when a new VHDL file is inserted the designer must declare whether the «new source» is a source or a testbench file

Testbench signals delay modelling

VHDL allows to define the testbench stimuli delays through the command wait

N.B. The statements blocks of a VHDL testbench – differently from the circuit under test – are executed *sequentially* - that is all the statements between two *wait* are executed in parallel while those after a *wait* are executed once the *wait period* ended

```
wait for 15 ns;
  statement 1;  -- all these
  statement 2;  --   statements
  ---           --     are executed
  statement n;  --       in  parallel
wait for 50 ns; -- here the test program stops for 50 ns
```

N.B. the wait command in Xilinx can be used only in the testbenches

«HELLO» testbench

```
entity testbench_of_hello_vhdl is -- n.b. in the testbenches no port
end    testbench_of_hello_vhdl;

architecture behavior of testbench_di_hello_vhdl is

-- component declaration for the unit under test (uut)

   component hello_vhdl
              port (x : in   std_logic;     -- input and output signals declaration
                    y : in    std_logic                  -- of the system (entity) under test
                    z : out std_logic );                      -- indicated as "component"
    end component;

  --inputs
              signal x_test : std_logic ;   -- testbench internal signals declaration
              signal y_test : std_logic ;                    -- part of the architecture

-- x_test e y_test not initialized !!
   --outputs

              signal z_test : std_logic;
```

Component

testbench architecture internal signals

N.B. In the testbench the command process is used which will be better explained later

Process !

Port map Correspondence of (x,z,z) signals and the testbench signals (x_test, y_test, z_test.)

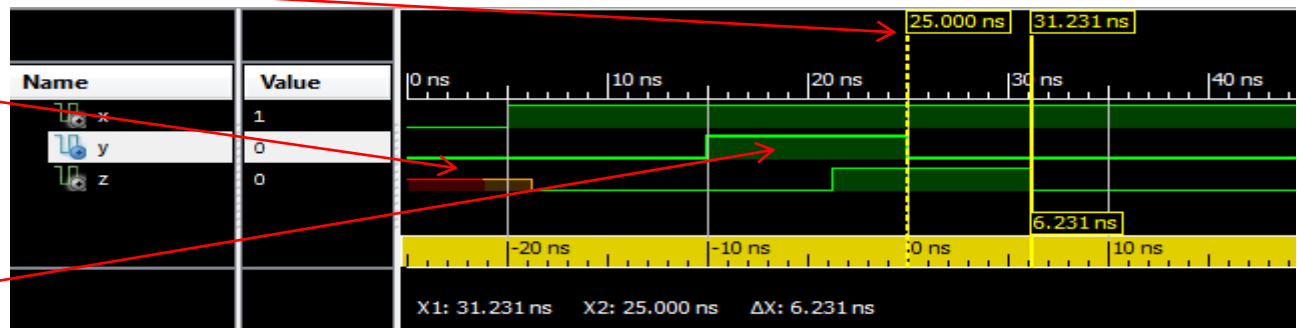
N.B.The internal signals can have the same name used in the circuit under test (easier to remember)

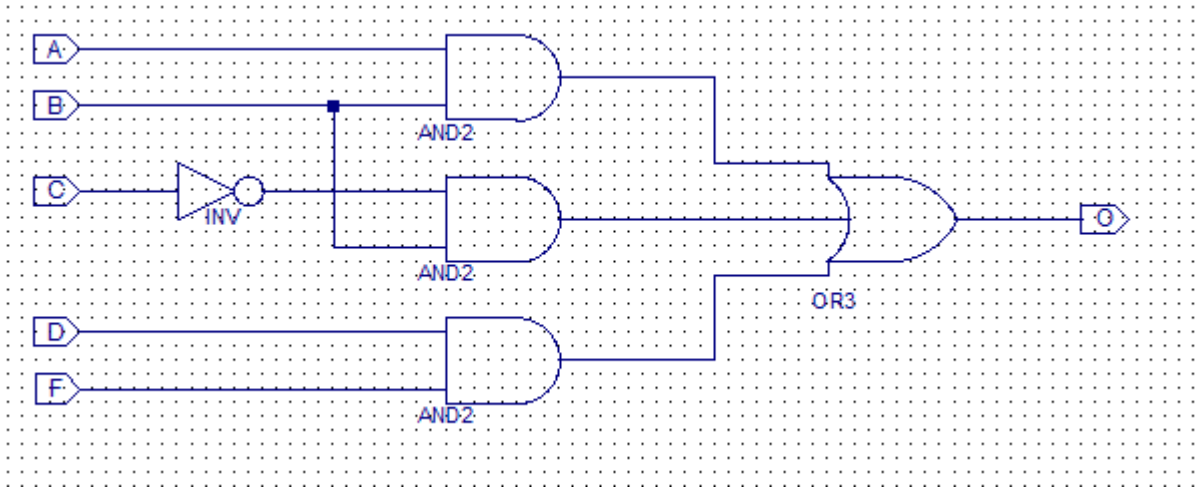
```
begin
-- instantiate the unit under test (uut)

   uut: hello_vhdl port map       ( x => x_test,     -- corrispondence between the
                                    y => y_test,          -- testbench signals and the
                                    z => z_test );          -     - «component» signals

-- very often the internal signals names are chosen identical to those of the port
-- stimuli process

   stim_proc: process
   begin
                                wait for 5 ns;          -- wait  5 nanoseconds

-- stimuli
-                               x_test<='1'; y_test<='0'; wait for 10 ns;
                                x_test<='1'; y_test<='1'; wait for 10 ns;
                                x_test<='1'; y_test<='0'; wait for 10 ns;

                                wait;           -- wait forever – test end
   end process;
end;
```

Time set

Red signal: undefined value

AND gate delay






```
library ieee;
use ieee.std_logic_1164.ALL;

entity xyz is
   port ( A                    : in    std_logic;
          B                    : in    std_logic;
          C                    : in    std_logic;
          D                    : in    std_logic;
          F                    : in    std_logic;
          O                    : out   std_logic);
end xyz;

architecture BEHAVIORAL of xyz is

O <= (A and B ) or (B and (not(C))) or (D and F) ;

end BEHAVIORAL;
```

Derive the truth table
Design and simulate in Xilinx Vivado !!!!

Signals assignment and "*Delta delay*"

- **It must be once again underlined that in VHDL it is of the utmost importance to remember that a *signal* is NOT immediately updated BUT after a delay which depends on the FPGA technology**
- ***Variables not signals* are immediately updated. *Variable have NO physical meaning.* For instance let a and b signals and I a variable**

```
I := I+1   -- immediate
a <= b   --  after  the intrinsic network delay
```

- **A typical mistake is to assign a value to a signal and *then test immediately the value of that signal*. The test provides an incorrect result since it is executed *at the same time of the assignment statement*. The signal will be updated only after the so called delta delay (in case of synchronous networks one clock period – see later)**

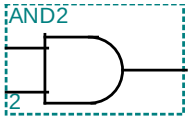
Statement when-else

```
<signal_name>   <=   <signal/value> when <condition-1> else
                     <signal/value> when <condition-2> else
                                          . . . .
                                          . . . .
                     <signal/value> when <condition-n> else
                     <signal/value>;
```

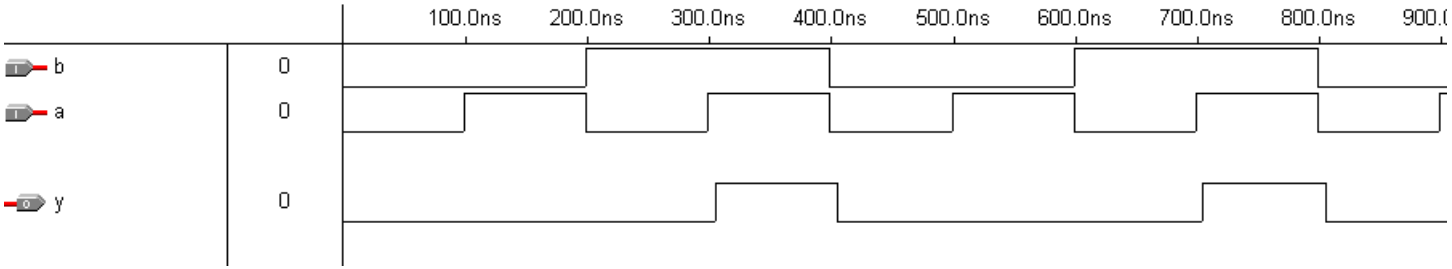
- **In case multiple conditions are verified the first value is assigned to the *signal_name* which satisfies the when condition**

VHDL core for modelling a two-inputs AND using when-else

Example (and2)

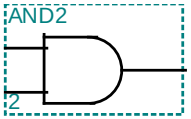



```
entity and2 is
   port (a, b :          in    std_logic;
               y   :     out   std_logic);
end and2;

architecture architecture_and2 of and2 is

begin

                y<='1' when (a='1' and b='1') else
                     '0';

end architecture_and2;
```

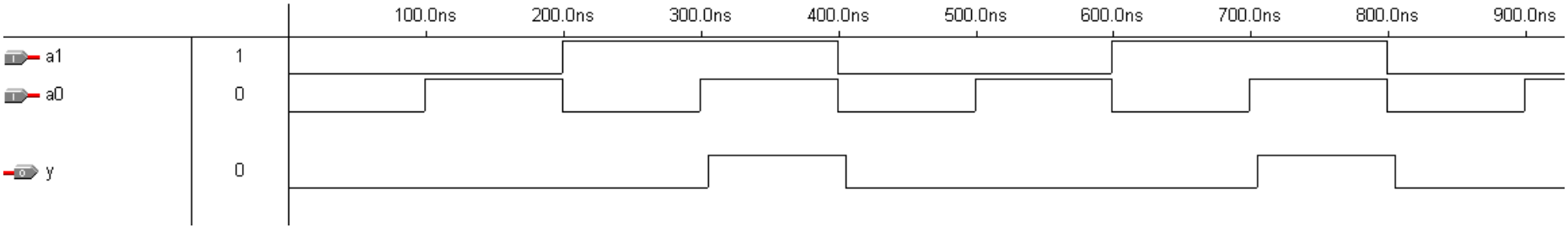


Alternative VHDL code using when-else again

Example (and2)

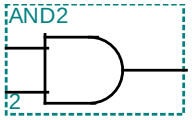



```
entity and2whenelse1 is
   port (a : in     std_logic_vector (1 downto 0); -- vector notation
         y : out  std_logic);
end and2whenelse1;

architecture arch_and2whenelse1 of and2whenelse1 is

begin

        y <=  '0' when a(0)='0' else
              '0' when a(1)='0' else
              '1';

end arch_and2whenelse1;
```

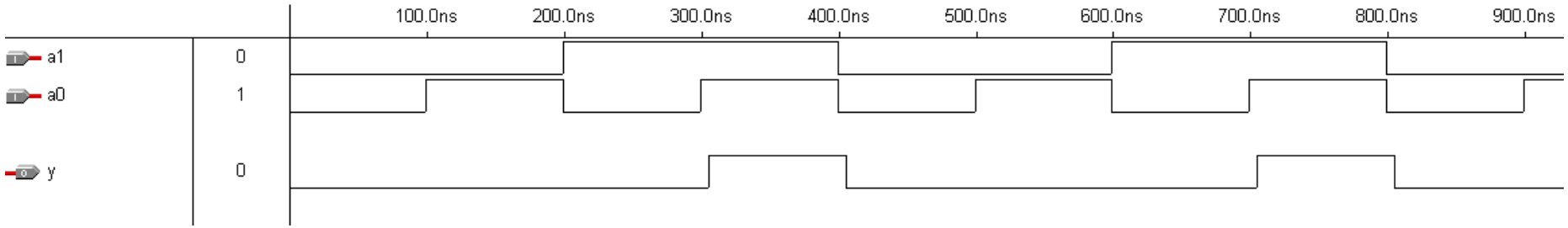


Another when-else alternative

Example (and2)

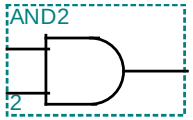



```
entity and2whenelse2 is
   port (a  : in     std_logic_vector (1 downto 0);
         y : out   std_logic);
end and2whenelse2;

architecture arch_and2whenelse2 of and2whenelse2 is

-- Here we use a vector!!

begin

                y <= '0' when a(0)='0' else
                     '1' when a(1)='1' else  -- a(0) already verified  not 0
                     '0';

end arch_and2whenelse2;
```



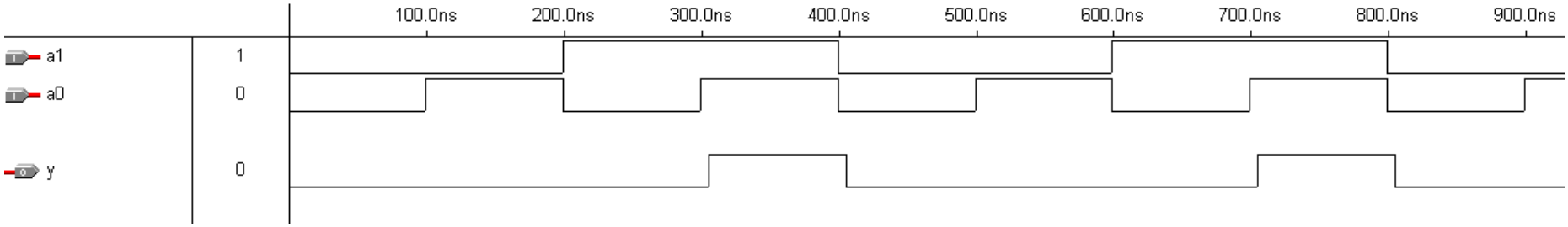
Or…….

Example (and2)




```
entity and2whenelse3 is
   port (a : in     std_logic_vector (1 downto 0);
         y : out    std_logic);
end and2whenelse3;

architecture arch_and2whenelse3 of and2whenelse3 is

begin

      y <= '1' when   a="11" else  --  notice the double apex:
                                       -- a is in this case is a vector to which
            '0';                       --  two values are assigned

end arch_and2whenelse3;
```

Important!



Statement with-select-when

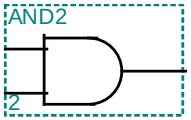
```
with <expression> select
<signal_name>   <=     <signal/value> when <condition1>,
                       <signal/value> when <condition2>,
                                                  . . . .
                                                  . . . .
                       <signal/value> when others;
```

Typically a signal or a vector

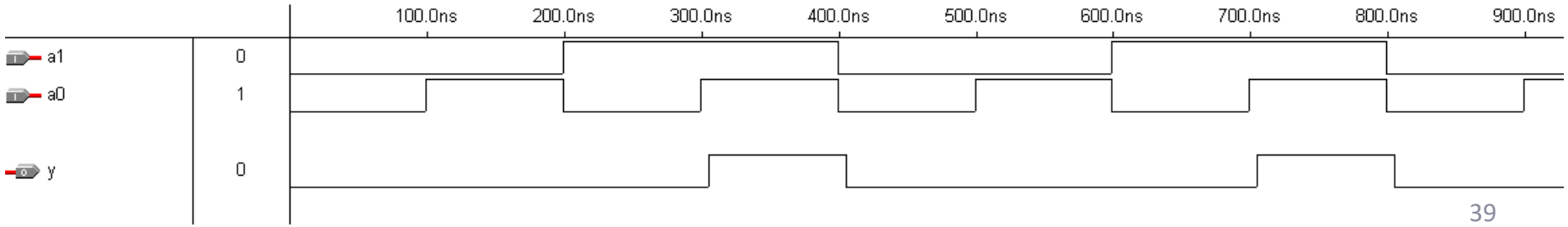
- **All conditions are simultanously verified with a SELECT (different from when-else). Conditions must be *mutually exclusive***
- **In case multiple conditions could be met they must be grouped (otherwise an illicit *multiple assignment* could take place with unpredictable result)**
- **when others option allows to handle the case when *no condition is met***

AND2 VHDL using with-select-when

Example (and2)

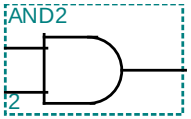



```
entity and2with is
   port (a : in     std_logic_vector (1 downto 0);
         y : out    std_logic);
end and2with;

architecture arch_and2with of and2with is

begin

with a select

    y <= '1' when "11",
         '0' when "00",
         '0' when "01",
         '0' when "10";

end arch_and2with;
```

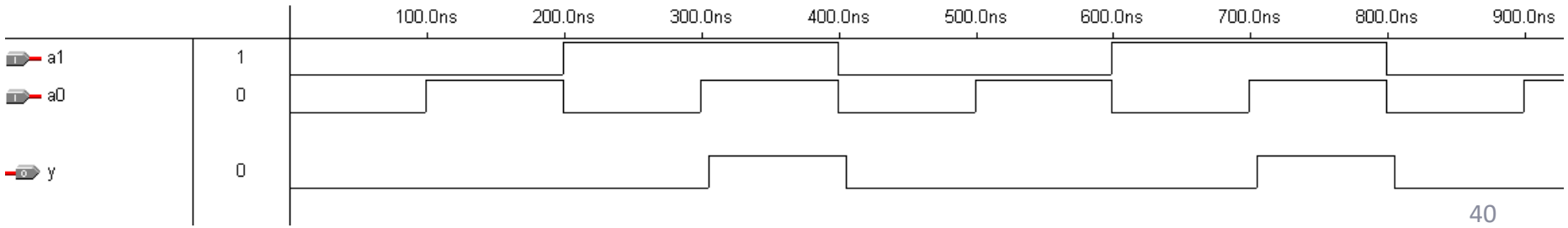


Or…

Example (and2)




```
entity and2withothers is
   port (a : in    std_logic_vector (1 downto 0);
         y : out  std_logic);
end and2withothers;

architecture arch_and2withothers of and2withothers is

begin

with a select
        y <= '1' when "11",
             '0' when others;

end arch_and2withothers;
```



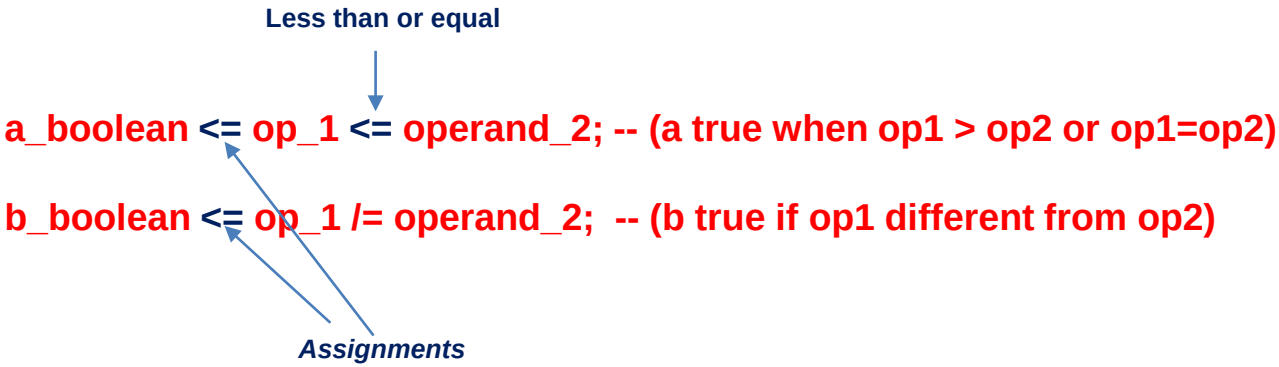
VHDL relational operators

These operators can be used for operands of the same type and produce a BOOLEAN value (TRUE or FALSE).

Equal:	**=**
Different:	**/=**
Less than:	**<**
Less than or equal:	**<=**
Greater than:	**>**
Greater than or equal:	**>=**

Careful ! It looks like the signals assignment but in this case these are variables . All depends on the context

Example:



'46A, '47A, 'LS47 FUNCTION TABLE (T1)

DECIMAL OR FUNCTION	INPUTS						$\overline{BI}/\overline{RBO}$†	OUTPUTS							NOTE
	$\overline{LT}$	$\overline{RBI}$	D	C	B	A		a	b	c	d	e	f	g	
0	H	H	L	L	L	L	H	ON	ON	ON	ON	ON	ON	OFF	1
1	H	X	L	L	L	H	H	OFF	ON	ON	OFF	OFF	OFF	OFF	
2	H	X	L	L	H	L	H	ON	ON	OFF	ON	ON	OFF	ON	
3	H	X	L	L	H	H	H	ON	ON	ON	ON	OFF	OFF	ON	
4	H	X	L	H	L	L	H	OFF	ON	ON	OFF	OFF	ON	ON	
5	H	X	L	H	L	H	H	ON	OFF	ON	ON	OFF	ON	ON	
6	H	X	L	H	H	L	H	OFF	OFF	ON	ON	ON	ON	ON	
7	H	X	L	H	H	H	H	ON	ON	ON	OFF	OFF	OFF	OFF	
8	H	X	H	L	L	L	H	ON	ON	ON	ON	ON	ON	ON	
9	H	X	H	L	L	H	H	ON	ON	ON	OFF	OFF	ON	ON	
10	H	X	H	L	H	L	H	OFF	OFF	OFF	ON	ON	OFF	ON	
11	H	X	H	L	H	H	H	OFF	OFF	ON	ON	OFF	OFF	ON	
12	H	X	H	H	L	L	H	OFF	ON	OFF	OFF	OFF	ON	ON	
13	H	X	H	H	L	H	H	ON	OFF	OFF	ON	OFF	ON	ON	
14	H	X	H	H	H	L	H	OFF	OFF	OFF	ON	ON	ON	ON	
15	H	X	H	H	H	H	H	OFF	OFF	OFF	OFF	OFF	OFF	OFF	
BI	X	X	X	X	X	X	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF	2
RBI	H	L	L	L	L	L	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF	3
LT	L	X	X	X	X	X	H	ON	ON	ON	ON	ON	ON	ON	4

H = high level, L = low level, X = irrelevant

NOTES:
1. The blanking input ($\overline{BI}$) must be open or held at a high logic level when output functions 0 through 15 are desired. The ripple-blanking input ($\overline{RBI}$) must be open or high if blanking of a decimal zero is not desired.
2. When a low logic level is applied directly to the blanking input ($\overline{BI}$), all segment outputs are off regardless of the level of any other input.
3. When ripple-blanking input ($\overline{RBI}$) and inputs A, B, C, and D are at a low level with the lamp test input high, all segment outputs go off and the ripple-blanking output ($\overline{RBO}$) goes to a low level (response condition).
4. When the blanking input/ripple blanking output ($\overline{BI}/\overline{RBO}$) is open or held high and a low is applied to the lamp-test input, all segment outputs are on.

†$\overline{BI}/\overline{RBO}$ is wire AND logic serving as blanking input ($\overline{BI}$) and/or ripple-blanking output ($\overline{RBO}$).

f_e = A + !BC using the algebra theorems (see next slide)
(If f_e = 1 the segment is off)
A,B,C,D negative true (L)

«Behavioural» description

- **Very often it is useful to avoid a «structural» description of the network to be synthesized describing only its behaviour (for instance a microprocessor)**

- **It is therefore necessary to use a «behavioural» description where not the structure but only the behaviour of the circuit is defined.**

- **This is obtained through the «process» statement which is allowed only within an «architecture» definition as it is the case for the signals assignment (already seen in the testbench)**

- **A process definition MUST include a «sensitivity list» that is a list of all input signals whose change (one at least) triggers all the activities of the process. (*The only exception is the testbench which has no sensitivity list*). In the following example a new value of a takes place ONLY if there are events for b or c *that is* if b or c change. In the synchronous sequential networks a typical event is the clock (see later)**

```
compute_xor: process (b,c)
begin
        a<=b xor c;
end process;
```

- **Sensitivity list: is the list of the signals for which the process is sensible. They represent the *state change* which possibily provokes a change in the internal expressions of the process. If none of these events takes place the process keeps inactive. In Xilinx sensitivity list all input signals (no matter if they act or not) must be in any case indicated**

Signals attributes

- **In order to control the signals evolution, attributes can be associated to the signals.**

- **An event for a signal taks place whenever a new value is associated to the signal (i.e. the signal changes is value). For instance**

b'event

means that either b changes from 0 to 1 or from 1 to 0). Event attributes is typically used to indicate a signal edge. For instance a positive edge of signal b can be detected with:

(b'event) and (b= '1') multiple condition on signal b, true if b is changed and b is become 1 (a transition from 0 to 1)

- **Last_value: a signal which assumes the previous value of a signal**

A <= B'last_value

There are many other attributes here not presented

Array attributed

In VHDL there are also array attributes. For instance for a vector

VECTOR: std_logic_vector(4 downto 0);

VECTOR'length provides value 5 (like «sizeof» in C)

Statement others

With others it is possible to set the content of some or of all elements of a vector not otherwise explicitly set. For instance

```
vector <= (0=>'1', others =>'0');
```

(the right arrow means always assignment). This statement set to 1 the LSbit and all others to 0 (How many ? With others *it is not necessary* to know this information).

Conversion functions

In the IEEE libraries some functions are defined to convert values. For instance in the IEEE.std_logic_arith a function is defined to convert from integer to STD_LOGIC_VECTOR (and many others)

Integer with or withou sign

<slv_sig> = CONV_STD_LOGIC_VECTOR(<int_sig>, <int_size>);

Vettore *Bits number*

For instance if half_word is a signal type STD_LOGIC_VECTOR(15 downto 0) and value is type integer, the conversion of value to a STD_LOGIC_VECTOR can be achieved through:

half_word = CONV_STD_LOGIC_VECTOR(value, 16);

In Xilinx ISE the functions are displayed when the toolbar icon  is clicked. With the same command the precompiled components can be used.

Computation

- All expressions within begin and end of an architecture are computed *simultaneously independently* from the order they appear in the VHDL code
- With process statement (in practice *always* used in the real designs) it is possible to define a set of instructions (or of sequential structures – i.e. if-then-else – «blocks») which are executed upon the change of a *sensitivity list* signal
- Within a process *block* (i.e. if, case, for etc. see later) the instructions are computed *sequentially* from the top downward. All blocks of a process are concurrently executed (but for wait or after - only in the testbenches allowed).
- Each process is concurrently executed with other triggered processes definined in the architecture.
- *Remember: signals in an architecture must be declared before architecture begin and are common to all processes*
- *Variables on the contrary are local in each single process and must be declared after process and before begin (of each process).* Variables with the same name in different processes have no mutual relation. Multiple processes can coexist within an architetture

N.B. within the processes the previously examined structures (when-else, with-select-when etc.) CANNOT be used. They are allowed *externally* to processes. These structures and processes can coexist within an architecture (although very unlikely)

Sequential Statements

Sequential statements **can be used only within processes (PROCESS) (also in functions (FUNCTION) and procedures (PROCEDURE))**

The set of instructions of a process are *concurrent statements*

- **if then else**
- **if then elsif else**
- **case when others**
- **wait**
- **loop**

Example

```
process_example_1: process(sensitivity list)
begin

   assignment            --  operation a
   if then else          --  operations b
   assignment            --  operation c
   for loop              --  operations d
   case when others      --  operations e
    ----------------

end process process_example_1;
```

```
process_example_2: process(sensitivity list)
begin

   assignment            --  operation a
   if then else          --  operations b
   assignment            --  operation c

---- a,b,c, concurrent

   wait for 10 ns             --  testbench only

   for loop              --  operations d
   case when others      --  operations e

------ d,e concurrent  but  delayed 10 ns

end process process_example_2;
```

In example 1 a,b,c,d,e etc. are executed *in parallel* but the statements within *if then else* and *for loop* are executed *sequentially*. The same for the example 2 but *d,e* operations because of wait are delayed.

N.B. wait and after in Xilinx only in the testbenches

Statement if-then-else

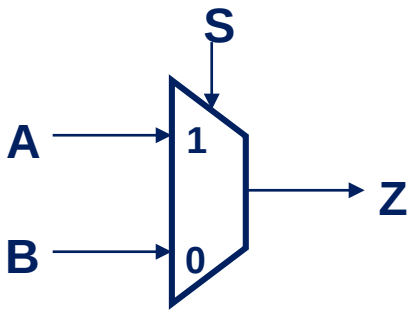
```
if   <condition>     then
                          <instruction_1>;
                     else
                          <instruction_2;
end if;
```

The statement if-then-else is used to execute an instruction block following the value of a boolean expression (*<condition>*). Example:

```
process_1: process(b)
begin
        if (b'event) and (b='1') then     – transition of b from 0 to 1
                z <='1';                  --     statements
        else
                z <='0';                  --          sequentially
        end if;                           --                executed

end process process_1; -- In practice a positive edge of signal b provokes   z=1
                       --  otherwise z=0
```

The "condition" can test also a variable (for instance within a loop)

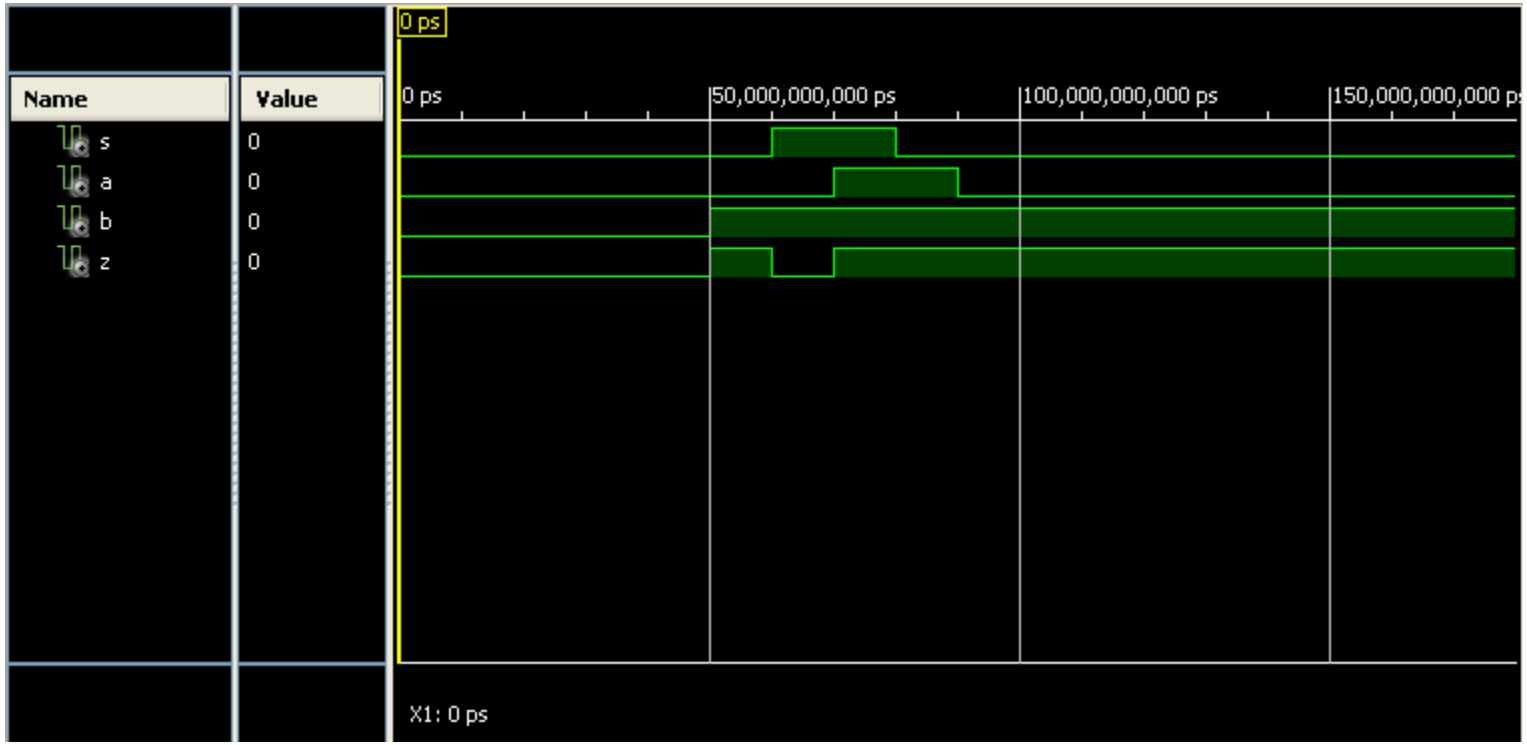
2-ways mux (if-then-else)




```
entity mux_2_ways_if_then_else is
   port ( a : in   std_logic;
          b : in   std_logic;
          s : in   std_logic;
          z : out std_logic);
end mux_2_wys_if_then_else;

architecture behavioral of mux_2_ways_if_then_else is
begin
       process_mux_2: process(a,b,s)
              begin            -- of the process
              if s='1' then
                           z<= a ;
              else
                           z<=b;
              end if;
       end process process_mux_2;
end behavioral;
```

How would you design a 2-ways MUX without a process ?

Z <= A and S or B and notS



What would happen if A and B signals were not inserted in the *sensitivity list* of the process ? Xilinx sends an error message if *all input signals* (no matter if they are used) are not inserted in the sensitivity list !!!

Statement if-then-elsif-else

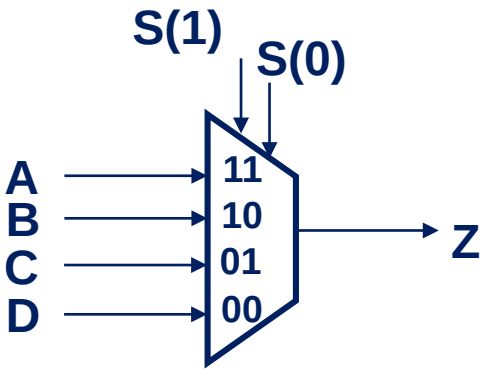
```
if      <condition_1>      then <instruction_1>;
elsif   <condition_2>      then <instruction_2>;
                  . . . .
                  . . . .
elsif   <condition_n-1>    then <instruction_n-1>;
else    <instruction_n>;
end if;
```

The statement if-then-else can be further expanded in order to allow multiple conditions evaluations using elsif.

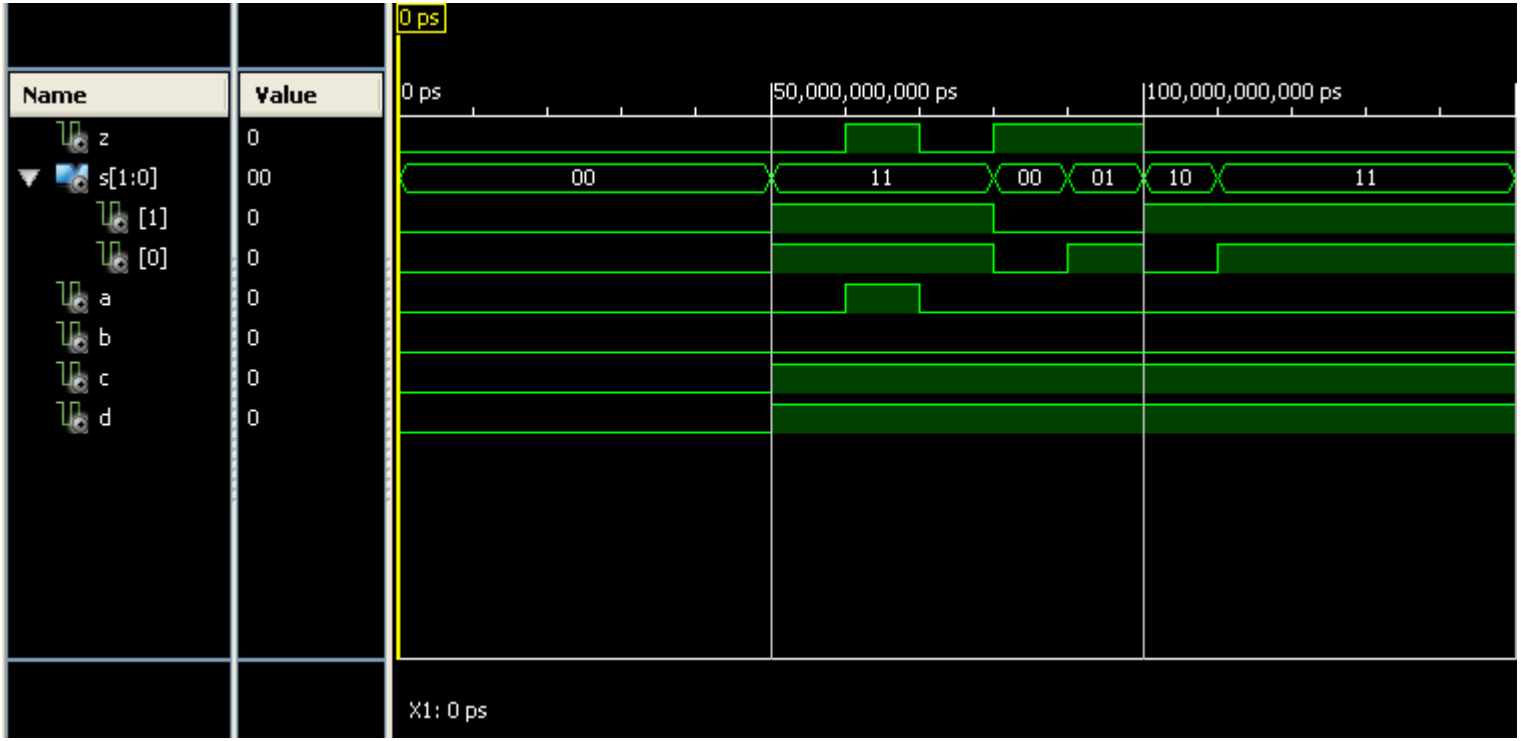
Nested if are obviously allowed

It must be underlined that the execution of the *<instruction_i>* is activated ONLY if the previous conditions (1,2 …,i-2, i-1) are not met. *Sequential evaluation !!!*

4 ways mux (if-then-elsif-else)



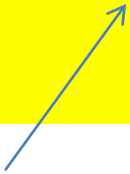

```
entity mux_4_ways_if_then_else_elseif is
   port (  s  : in std_logic_vector (1 downto 0);
           a : in std_logic;  b : in std_logic; c : in std_logic; d : in std_logic;
           z : out std_logic);
end mux_4_ways_if_then_else_elseif;

architecture behavioral of mux_4_ways_if_then_else_elseif is
begin
           process_mux_4: process(a,b,c,d,s)
           begin
                      if     s="11"  then z<=a;
                      elsif s="10“  then z<=b;
                      elsif s="01“  then z<=c;
                      else  z<=d ;
                      end if;
           end process process_mux_4;
end behavioral;
```



It must be noticed that Xilinx simulator displays vectors using eye notation. These «eyes» can be always splitted showing the single signals.

Statement case-when-others

```
case <selection_signal> is
                    when <value_1>    => <instruction_1>;
                    when <value_2>    => <instruction_2>;
                    when <value_3>    => <instruction_3>;
                                  ..          ..          ..
                                  ..          ..          ..
                    when <value_n-1> => <instruction_n-1>;
                    when others       => <instruction_n>;
end case;
```



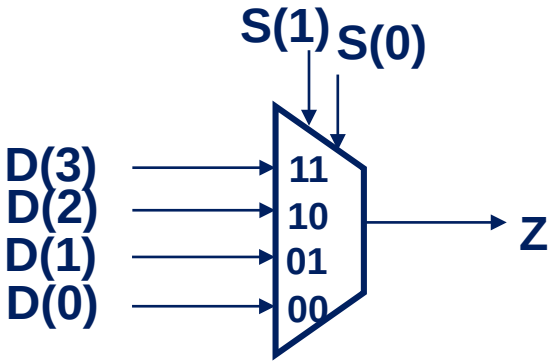
N:B: notice the arrow direction

The statement case-when is used for executing a set of instructions on the basis of the value of the signal <*selection_signal*>.

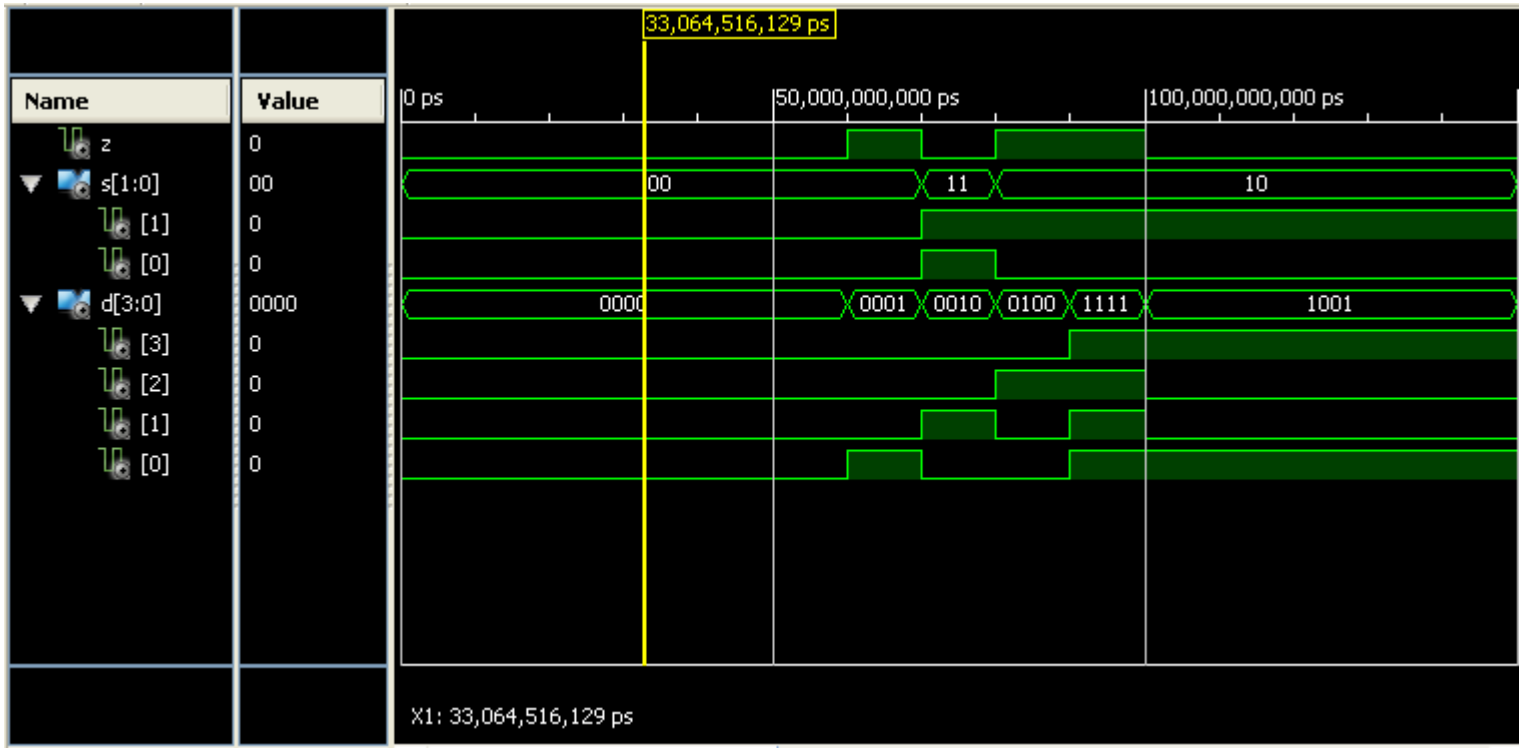
The options must be mutually exclusive

With others it is possible to execute the <*instruction_n*> when none of the previous conditions are verified. If no action must be executed in this case, the reserved word null (no_op) can be used.

4-ways mux (case-when-others)




```
entity mux_4_ways_case_when_others is
   port ( s : in std_logic_vector   (1 downto 0);
          d : in std_logic_vector   (3 downto 0);
          z : out std_logic);
end mux_4_ways_case_when_others;

architecture behavioral of mux_4_ways_case_when_others is
begin
            process_mux_4: process(s,d)
            begin
                        case s is
                                    when "11"     => z <= d(3);
                                    when "10"     => z <= d(2);
                                    when "01"     => z <= d(1);
                                    when "00"     => z <= d(0);
                                    when others  => z <= d(0);-- always necessary even
                                                                -- when meaningless
                        end case;
            end process process_mux_4;
end behavioral;
```


33,064,516,129 ps
Name
Value
0 ps
50,000,000,000 ps
100,000,000,000 ps
z
0
s[1:0]
00
00
11
10
[1]
0
[0]
0
d[3:0]
0000
0000
0001
0010
0100
1111
1001
[3]
0
[2]
0
[1]
0
[0]
0
X1: 33,064,516,129 ps

N.B. STD_LOGIC can assume 9 values but in our context we use only 4: 'U' (undefided) , '0' , '1' and 'Z' (high impedance – tristate). The statement type indicates the possible values of an «object»

```
type std_logic is        ('U', -- not initialized
                          'X', -- unknown
                          '0', -- 0
                          '1', -- 1
                          'Z', -- high impedance– tristate…..
                          'W', -- unkown (weak)
                          'L',  -- 0 (weak)
                          'H', -- 1 (weak)
                          '-'), -- indifferent
```

NB Important !!! With this «type " (std_logic) it is possible in Xilinx to execute *additions and subtractions* with vector (*even of different size* !) which are interpreted in this case as their binary values (*they are in fact internally converted)* provided IEEE library is used.

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

Example

```
signal: addend_6    std_logic_vector (5 downto 0) := "010010"; --  18
signal : addend_2   std_logic_vector (1 downto 0) := "11";          --  3
signal : result     std_logic_vector (5 downto 0) ;

result <= addend_6 – addend_2; -- in result  we find  "001111" that is 15
```

Statement Loop

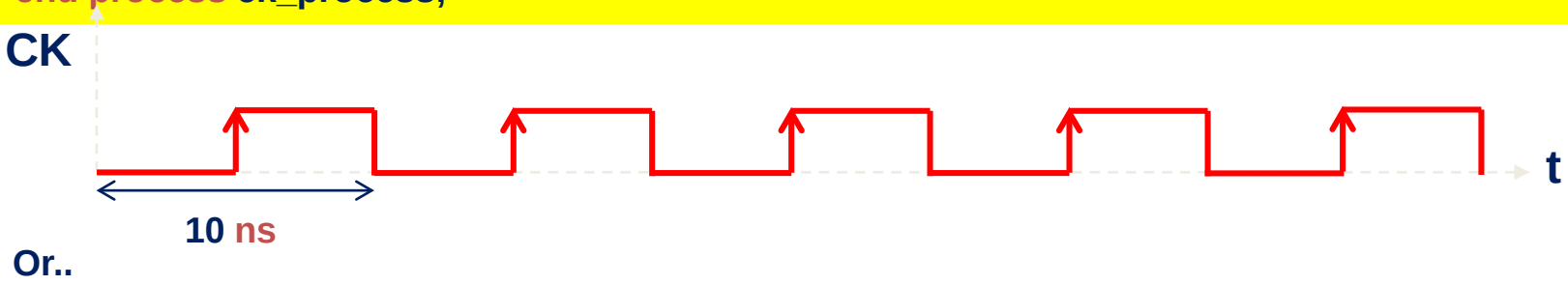
This statement for *<variable> in <initial_value>* downto *<final_value>* loop or for *<variable> in <initial_value>* to *<final_value>* loop is easily interpreted. *All operations within the loop are simultaneously executed*. Example (*I is an integer variable* and val_max_index a *constant integer*)

```
for I in 0 to val_max_index -1   loop -- I variable.
                                                --NB val_max_index -1  because of the 0 and therefore
                                                        -- the loop is val_max_index times repeated
    A(I) <= B(I) exor B(I+1); -- A and B std_logic_vector

end loop;
```

Periodical signals generation with wait

The statement wait can be used (only in the testbench) to generate periodical signals (*no sensitivity list*). For instance:

```
ck_process :process
            begin
                for i in 735 downto  0  loop

                        ck <= '0';
                        wait for 5 ns;
                        ck <= '1';
                        wait for 5 ns;

                end loop;
end process ck_process;
```



Or..

```
ck_process :process
            begin

                for i in 623 downto  449  loop

                        ck <= not(ck);
                        wait for 5 ns;

                end loop;
end process ck_process;
```

Vivado templates

In Vivado select *tools* and then *templates:* you will find a wealth of information and thousands examples of constructs ready-to-use. An example for loop (while)