Sequential Process, Flip Flops, Test Bench, Live Examples

Gaurav Verma
Asst. Prof.
ECE Dept.
Sequential VHDL

- Concurrent and sequential data processing
- Signal and variable assignment
- Process statement
  - Combinational process
  - Clocked process
- If statement
- Case statement
- Multiple statement
- Null statement
- Wait statement
Concurrent and sequential data processing

- Designing sequential processes presents a new design dimension to hardware designers who have worked at gate level
- First we look at concurrent and sequential data processing
- The difference between sequential and concurrent assignment is important to understand
- Electronics are parallel by nature
- In practice, all the gates in a circuit diagram can be seen as concurrently “executing” units
- In VHDL there are language constructions which can handle both concurrent and sequential data processing
- Many of the VHDL commands are only valid in the concurrent or sequential part of VHDL
Concurrent VHDL constructions

- **Process** statement
- **When** else statement
- **With** select statement
- **Signal** declaration
- **Block** statement
Sequential VHDL constructions

- If-then-else statement
- Case statement
- Variable statement
- Variable assignment
- Loop statement
- Return statement
- Null statement
- Wait statement
VHDL commands being valid in both the concurrent and the sequential parts

- **Signal** assignment
- Declaration of **types** and **constants**
- **Function** and **procedure** calls
- **Assert** statement
- **After** delay
- **Signal** assignment
Use of commands

- It is important to know where the various VHDL commands can be used and where the VHDL code is concurrent or sequential.
- In brief, it can be said that the VHDL code is concurrent throughout the architecture except for in processes, functions and procedures.

```vhdl
architecture rtl of ex is
concurrent declaration part
begin
concurrent VHDL
process (...)
sequential declaration part
begin
sequential VHDL
end process;
concurrent VHDL
end;
```

Process, one concurrent statement
Comparison of signal and variable assignment

- **Signals** and **variables** are used frequently in VHDL code, but they represent completely different implementations.
- **Variables** are used for sequential execution like other algorithmic programs, while **signals** are used for concurrent execution.
- **Variable** and **signal** assignment are differentiated using the symbols “:=” and “<=”
- A **variable** can only be declared and used in the sequential part of VHDL.
- **Signals** can only be declared in the concurrent part of VHDL but can be used in both the sequential and concurrent parts.
- A **variable** can be declared for exactly the same data types as a **signal**.
- It is also permissible to assign a **variable** the value of a **signal** and vice versa, provided that they have the same data **type**.
- The big difference between a **signal** and a **variable** is that the signal is only assigned its value after a delta delay, while the variable is given its value immediately.
Variable assignment statement

variable_assignment_statement ::= [ label : ] target := expression ;

Examples:
- var := 0;
  var receives the value 0.
- a := b;
  a receives the current value of b.
- a := my_function ( data, 4 );
  a receives the result of the function my_function as a new value.
Signal assignment statement

signal_assignment_statement ::= target <= [ delay_mechanism ] waveform ;

Example:

A <= B after 5 ns;

The value of B is assigned to signal A after 5 ns.
The inertial delay model is used.
Differences between signal and variable processing

- The lines in the sequential VHDL code are executed line by line
  - That is why it is called sequential VHDL
- In concurrent VHDL code, the lines are only executed when an event on the sensitivity list occurs
  - This sensitivity list is explicit in case of process
  - Anyway the sensitivity list means the arguments of the operation
Differences between signals and variables

- Sum1 and sum2 are signals
  p0: process
  begin
  wait for 10 ns;
  sum1<=sum1+1;
  sum2<=sum1+1;
  end process;

- Sum1 and sum2 are variables
  p1: process
  variable sum1, sum2: integer;
  begin
  wait for 10 ns;
  sum1:=sum1+1;
  sum2:=sum1+1;
  end process;

<table>
<thead>
<tr>
<th>Time</th>
<th>Sum1</th>
<th>Sum2</th>
<th>Sum1</th>
<th>Sum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10 + Δ</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>20 + Δ</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>30 + Δ</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Information transfer

- Variables cannot transfer information outside the sequential part of VHDL in which it is declared, in the previous example process `p1`.
- If access is needed to the value of `sum1` or `sum2`, they must be declared as signals or the value of the variable assigned to a signal.

**Entity ex is**

```vhdl
Entity ex is
  port(sum1_sig, sum2_sig: out integer);
end;
```

**Architecture bhv of ex is**

```vhdl
Architecture bhv of ex is
begin
  p1: process
    variable sum1, sum2: integer;
    begin
      wait for 10 ns;
      sum1:=sum1+1; sum2:=sum1+1;
      sum1_sig<=sum1; sum2_sig<=sum2;
    end process;
end;
```

- **VHDL-87**: Variables can only store temporally values inside a process, function or procedure.
- **VHDL-93**: Global shared variables have been introduced which can transfer information outside the process.
Global variables

Architecture bhv of ex is
shared variable v: std_logic_vector(3 downto 0);
begin
  p0: process(a, b)
    begin
      v:=a & b;
    end process;
  p1: process(d)
    begin
      if v="0110" then c<=d;
      else c<=(others=>‘0’);
    end if;
  end process;
end;

- **Shared variables** are not accessible in the concurrent part of VHDL code, but only inside other **processes**.
- Nor can a **shared variable** be included in a sensitivity list of a process.
- **Shared variables** can give rise to non-determinism.
- Synthesis tools do not support **shared variables**.
Process statement

- The **process** concept comes from software and can be compared to a sequential program.
- If there are several processes in an architecture, they are executed concurrently.
- A **process** can be in Waiting or Executing state.

If the state is Waiting, a condition must be satisfied, e.g. **wait until clk='1';**.
- This means that the process will start when \(clk\) has a **rising edge**.
- Then the process will be Executing.
- Once it has executed the code, it will wait for the next rising edge.
- The condition after the until means not only the necessity of the value ‘1’, but also the necessity of **changing** the value to ‘1’.
Example to test the “wait until” command - circuit description

entity cir is
  port (a, clk: in bit; y: out bit);
end;

architecture bhv of cir is
begin
  process
  begin
    y <= a;
    wait until clk='1';
  end process;
end;
Example to test the “wait until” command - stimulus generator

entity stm is
  port (a,clk: out bit; y: in bit);
end;

architecture dtf of stm is
begin
  a<='1' after 20 ns,       '0' after 25 ns,
      '1' after 45 ns,       '0' after 58 ns;
  clk<='1' after 10 ns,    '0' after 30 ns,
      '1' after 50 ns,    '0' after 70 ns;
end;
Example to test the “wait until” command - test bench I

entity bnc is
end;

use std.textio.all;
Architecture str of bnc is
  Component cir port (a,clk:in bit;y:out bit); end Component;
  Component stm port (a,clk:out bit;y:in bit); end Component;
  for all: cir use entity work.cir(bhv);
  for all: stm use entity work.stm(dtf);
  Signal a,clk,y:bit;
Begin
...

Example to test the “wait until” command - test bench II

Begin
  c1: cir port map(a, clk, y);
  c2: stm port map(a, clk, y);

header: process
variable dline: line;
begin
  write (dline, string'(" TIME a clk y ")),
  writeln (output, dline);
  wait;
end process;
Example to test the “wait until” command - test bench III

data_monitoring:process (a,clk,y)
variable dline:line;
begin
  write (dline,now,right,7);
  write (dline,a,right,6);
  write (dline,clk,right,4);
  write (dline,y,right,4);
  writeline (output,dline);
end process;
end;
Example to test the “wait until” command - simulation result

# Loading c:\\_vhdl\\bin\\lib\\std.standard
# Loading c:\\_vhdl\\bin\\lib\\std.textio
# Loading c:\\_vhdl\\bin\\lib\\work.bnc(str)
# Loading c:\\_vhdl\\bin\\lib\\work.cir(bhv)
# Loading c:\\_vhdl\\bin\\lib\\work.stm(dtf)
run
Example to test the “wait until” command - simulation result

<table>
<thead>
<tr>
<th>TIME</th>
<th>a</th>
<th>clk</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ns</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 ns</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20 ns</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>25 ns</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>30 ns</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>45 ns</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50 ns</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>50 ns</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>58 ns</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>70 ns</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

quit
In some contexts, an expression involving an overloaded item may need to be qualified

So that its meaning may be unambiguous

An expression is qualified by enclosing the expression in parentheses and prefixing the parenthesised expression with the name of a type and a tic (')

Consider the following procedures declarations:

```
procedure to_integer (x: in character; y: inout integer);
procedure to_integer (x: in bit; y: inout integer);
```

The procedure call `to_integer ('1', n)` is ambiguous

Solution: `to_integer (character('1'), n);` or `to_integer (bit('1'), n);`
Qualified expression in the previous example

- In the package std.texio the overloaded procedure write is the next:
  
  procedure WRITE(L : inout LINE; VALUE : in bit_vector;
  
    JUSTIFIED: in SIDE := right;FIELD:in WIDTH := 0);

  procedure WRITE(L : inout LINE; VALUE : in string;

    JUSTIFIED: in SIDE := right;FIELD:in WIDTH := 0);

- Since the compiler does not check the expression between the quotation marks, the next procedure call may be ambiguous

  write (dline, " TIME a clk y ");

- In this case the error message:

  ERROR: bnc.vhd (17): Subprogram “write” is ambiguous.

  ERROR: bnc.vhd (17): type error resolving function call: write

  ERROR: bnc.vhd (38): VHDL Compiler exiting

- The solution is:

  write (dline, string'(" TIME a clk y "));
Process syntax and an example

- process_statement ::= [ process _label : ]
  [ postponed ] process [ ( sensitivity_list ) ]
  process_declarative_part
  begin
  process_statement_part
  end [ postponed ] process
  [ process _label ] ;

- Example: process
  begin
  if ( clock = '1' ) then q <= d after 2 ns ;
  end if ;
  wait on clock ;
  end process;

- This process describes a flip-flop. The positive clock-edge is detected by the wait - statement and by checking the condition clock = `1` .
- 2 ns after the clock-edge the value of input d is on output q .
Process execution

- Once the process has started, it takes the time \( \Delta \) time (the simulator’s minimum resolution) for it to be moved back to Waiting state
- This means that no time is taken to execute the process
- A process should also be seen as an infinite loop between `begin` and `end process`;
- \( \Delta \) time is used to enable the simulator to handle concurrency
- In reality, \( \Delta \) time is equal to zero
The process is started when the signal \( clk \) goes low.

- It passes through two statements and waits for the next edge.
- The programmer does not have to add loop in the process: it will restart from the beginning in any case.
- In the model these two statements will be executed in a \( \Delta \) time which is equal to the resolution of the simulator.
Modeling the process delay

- In practice the statements will take a different length of time to implement.
- This delay in the process can be modeled as follows:
  \[ c_{\text{out}} = \text{not} \ (a_{\text{in}} \text{ and } b_{\text{in}}) \text{ after } 20 \text{ ns}; \]
  \[ d_{\text{out}} = \text{not} \ a_{\text{in}} \text{ after } 10 \text{ ns}; \]
- This means that \( c_{\text{out}} \) will be affected 20 ns and \( d_{\text{out}} \) 10 ns after the start of the process.
- The simulator will place the result in a time event queue for \( c_{\text{out}} \) and \( d_{\text{out}} \):
  - \( A_{\text{in}} = 1 \)
  - \( B_{\text{in}} = 0 \)
  - \( C_{\text{out}} = 1 \quad \text{time} = x + 20 \text{ ns} \)
  - \( D_{\text{out}} = 1 \quad \text{time} = x + 10 \text{ ns} \)
- If \( a_{\text{in}} \) or \( b_{\text{in}} \) is changed and \( clk \) gives a falling edge, another event will be linked into the queue \textit{relative} to the time at which the change took place.
Types of the process

- There are two types of process in VHDL:
  - Combinational process
  - Clocked process
- Combinational processes are used to design combinational logic in hardware.
- Clocked processes result in flip-flops and possibly combinational logic.
Combinational process

- In the combinational process all the input signals must be contained in a sensitivity list of the process.
  - signals all to the right of <= or in if / case expression
- If a signal is left out, the process will not behave like combinational logic in hardware.
- In real hardware, the output signals can change if one or more input signals to the combinational block are changed.
- If one of these signals is omitted from the sensitivity list, the process will not be activated when the value of the omitted signal is changed, with the result that a new value will not be assigned to the output signals from the process.
- The VHDL standard permits signals to be omitted from the sensitivity list.
  - It is only in the design of simulatable models, and not for synthesis, that is there any point in omitting a signal from the sensitivity list.
  - If a signal is omitted from the sensitivity list in a VHDL design for synthesis, the VHDL simulation and the synthesized hardware will behave differently.
  - This is a serious error; it leads to incomplete combinational process.
Example for a combinational process

Example:

```
process (a, b, c)
begin
    d <= (a and b) or c;
end process;
```

The synthesis result:

![Logic Diagram]

A
B
C
V
D
Incomplete combinational process

In the case of design with combinational processes, all the output signals from the process must be assigned a value each time the process is executed. If this condition is not satisfied, the signal retain its value. The synthesis tool will perceive and resolve this requirement by inferring a latch for the output which is not assigned a value throughout the process. The latch will be closed when the old value for the signal has to be retained. Functionally, the VHDL code and the hardware will be identical.

But the aim of a combinational process is to generate combinational logic. If latches are inferred, the timing deteriorate with increased number of gates. What is more, the latch will normally break the test rules for generating automatic test vectors.

Processes will give rise to latches by mistake are called incomplete combinational processes.

The solution is simple: include all the signals which are “read” inside the process in the sensitivity list for combinational processes.
Clocked process

- Clocked processes are synchronous and several such processes can be joined with the same clock.
- No process can start unless a falling edge occurs at the clock (\(clk\)), if the clocking is: `wait until clk='0';`
- This means that data are stable when the clock starts the processes, and the next value is laid out for the next start of the processes.

**Example**
- Process A’s output signal \(d\_out\) is connected to the input of process B.
- The requirement is that \(d\_out\) must be stable before clock starts the processes.
- The longest time for process B to give the signal \(d\_out\) a stable value once the clock has started, determines the shortest period for the clock.
- In the next example this time is 10 ns.
Example for clocked processes

A: process
begin
  wait until clk='0';
  c_out<= not (a_in and b_in);
  d_out<= not b_in after 10 ns;
end process;

B: process
begin
  wait until clk='0';
  e_out<= not (d_out and c_in);
  f_out<= not c_in;
end process;
Component and architecture of the example

```vhdl
entity comp_ex is
port (clk,
     a_in,
     b_in,
     c_in: in std_logic;
     c_out,
     d_out,
     e_out,
     f_out: out std_logic);
end;

architecture rtl of comp_ex is
-- internal signal declaration
begin
  A: process
  begin
    ...
  end process;
  B: process
  begin
    ...
  end process;
end;
```

ETEC 301 - DCS-II  Mr. GAURAV VERMA
**Flip-flop synthesis with clocked signals**

-ex: process
begin
  wait until clk='1';
  d_out<= d_in after 1 ns;
end process;

- Clocked processes lead to all the signals assigned inside the process resulting in a flip-flop.
- This example shows how a clocked process is translated into a flip-flop.
- As figure shows, \( d_{in} \) is translated to \( d_{out} \) with 1 ns delay. When it has been synthesized, the transfer will be equal to the flip-flop’s specification.
Flip-flop synthesis with variables

ex: `process
   variable count:
       std_logic_vector (1 downto 0);
begin
   wait until clk='1';
   count:=count + 1;
   if count="11" then
      q<='1';
   else
      q<='0';
   end if;
end process;`

- Variables can also give rise to flip-flops in a clocked process.
- **If a variable is read before it is assigned a value, it will result in a flip-flop for the variable.**
- In this example with variable `count` the synthesis result will show three flip-flops.
- One for signal `q` and two for variable `count`. This is because we read variable `count` before we assign it a value `(count:=count+1;`).
Testable flip-flop synthesis

If a signal is not assigned a value in a clocked process, the signal will retain the old value.

The synthesis will result a feedback of the signal $d$ from the output signal from the flip-flop via a multiplexor to itself.

This design is good from the point of view of testability.
Flip-flop synthesis with gated clock

\[
\text{clk2} \leftarrow \text{clk and en};
\]

\text{ex2: } \text{process} \begin{align*}
\text{begin} \\
\text{wait until clk2='1';} \\
q &= d; \\
\text{end process;}
\end{align*}

- The clock can be gated with signal \text{en}.
- This design is not good from the point of view of testability and design methodology.

\begin{tabular}{c c c c c}
\hline
CLK & \text{EN} & \text{D} & \text{Q} & \text{QB} \\
\hline
\end{tabular}
Synthesis of adder and flip-flop

ex: process (clk, reset)
begin
    if reset='1' then
        q<=(others=>'0');
    elsif clk'event and clk='1' then
        if en='1' then
            q<=a+b;
        end if;
    end if;
end process;

- All logic caused by a signal assignment in a clocked process will end up on the "left" of the flip-flops before the flip-flop’s input.
If statement

- if_statement ::= [if_label:] 
  if condition then
    sequence_of_statements
  { elsif condition then
    sequence_of_statements }
  [ else sequence_of_statements ]
end if [ if_label ] ;

- Example:
  if a > 0 then
    b := a ;
  else
    b := abs (a+1) ;
  end if ;

- If the value of a is bigger than 0 then b is assigned the value of a, otherwise that of abs (a+1)
- Several elsifs are permitted, but just one else
Multiplexor modeling with if statement

\[
\text{if } \text{sel} = '1' \text{ then} \\
\quad \text{c} \leq \text{b}; \\
\text{else} \\
\quad \text{c} \leq \text{a}; \\
\text{end if;}
\]

The synthesis result:

![Multiplexer diagram]
Case statement

- `case_statement ::= [ case _label : ]
  
  case expression is
  
  case_statement_alternative
  
  { case_statement_alternative }  
  
  end case [ case _label ];

- Example:
  
  `case a is
  
  when '0' => q <= "0000" after 2 ns;
  
  when '1' => q <= "1111" after 2 ns;
  
  end case;`

- The value of bit `a` is checked. If it is 0 then `q` is assigned the value "0000" after 2 ns, otherwise it is assigned the value "1111", also after 2 ns.
Use of others

- All the choices in `case` statement must be enumerated
- The choices must not overlap
- If the `case` expression contains many values, the `others` is usable

```vhdl
entity case_ex1 is
  port (a: in integer range 0 to 30; q: out integer range 0 to 6);
end;
architecture bhv of case_ex1 is
begin
  p1: process(a)
  begin
    case a is
    when 0 => q<=3;
    when 1 | 2 => q<=2;
    when others => q<=0;
  end case;
end process;
end;
```
Simulation example of “others” - entity declaration

```vhdl
entity cir is
  port (a: in integer range 0 to 30;
        q: out integer range 0 to 6);
end;
```
Simulation example of “others” - architecture

architecture bhv of cir is
begin
p1: process(a)
begin
    case a is
        when 0 => q<=3;
        when 1 | 2 => q<=2;
        when others => q<=0;
    end case;
end process;
end;
Simulation example of “others” - stimulus generator

entity stm is
  port (a: out integer range 0 to 30;
       q: in integer range 0 to 6);
end;

architecture dtf of stm is
begin
  a<=0 after 10 ns,
     1 after 20 ns,
     6 after 30 ns,
     2 after 40 ns,
     0 after 50 ns,
     10 after 60 ns;
end;
Simulation example of “others” - test bench I

entity bnc is
end;

use std.textio.all;
architecture str of bnc is
component cir port (a:in integer range 0 to 30;q:out integer range 0 to 6); end component;
component stm port (a:out integer range 0 to 30;q:in integer range 0 to 6); end component;
for all:cir use entity work.cir(bhv);
for all:stm use entity work.stm(dtf);
signal a,q:integer:=0;
signal aid:Boolean;
Simulation example of “others” - test bench II

begin
    c1: cir port map(a,q);
    c2: stm port map(a,q);
fejlec: process
variable dline: line;
begin
    write (dline,string'(" time a q aid"));
    writeln(output,dline);
    write
        (dline,string'("==================================
        ==")));
    writeln(output,dline);
    aid<=True;
wait;
end process;
Simulation example of “others” - test bench III

data_monitoring:process
  (a,q,aid)
variable dline:line;
begin
  if aid=True then
    write (dline,now,right,7);
    write (dline,a,right,6);
    write (dline,q,right,4);
    write (dline,aid,right,6);
    writeln (output,dline);
  end if;
end process;
end;
Simulation example of “others” - result

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>a</th>
<th>q</th>
<th>aid</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0 ns</td>
<td>0</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>20 ns</td>
<td>1</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>20 ns</td>
<td>1</td>
<td>2</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>30 ns</td>
<td>6</td>
<td>2</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>30 ns</td>
<td>6</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>40 ns</td>
<td>2</td>
<td>0</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>40 ns</td>
<td>2</td>
<td>2</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>50 ns</td>
<td>0</td>
<td>2</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>50 ns</td>
<td>0</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>60 ns</td>
<td>10</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>#</td>
<td>60 ns</td>
<td>10</td>
<td>0</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

ETEC 301- DCS-II          Mr. GAURAV VERMA
Use of range

- It is permissible to define ranges in the choice list.
- This can be done using `to` and `downto`.

```vhdl
entity case_ex2 is
  port (a: in integer range 0 to 30; q: out integer range 0 to 6);
end;
architecture rtl of case_ex2 is
begin
  p1: process(a)
  begin
    case a is
      when 0 => q<=3;
      when 1 to 17 => q<=2;
      when 23 downto 18 => q<=6;
      when others => q<=0;
    end case;
  end process;
end;
```

ETEC 301 - DCS-II
Mr. GAURAV VERMA
Use of vectors - bad solution

- It is not permissible to define a range with a vector, as a vector does not have a range.

Example (bad):

```vhdl
entity case_ex3 is
port (a: in std_logic_vector(4 downto 0);
     q: out std_logic_vector(2 downto 0));
end;
architecture rtl of case_ex3 is begin
    p1: process(a) begin
        case a is
            when "00000" => q<="011";
            when "00001" to "11110" => q<="010"; -- Error
            when others => q<="000";
        end case;
    end process;
end;
```

ERROR!
Use of vectors - good solution

If a range is wanted, std_logic_vector must first be converted to an integer.

This can be done by declaring a variable of type integer in the process and then converting the vector using conversion function conv_integer

```plaintext
entity case_ex4 is
port (a: in std_logic_vector(4 downto 0);
     q: out std_logic_vector(2 downto 0)); end;
architecture rtl of case_ex4 is begin
p1: process(a)
variable int: integer range 0 to 31;
begin
int:=conv_integer(a);
case int is
    when 0 =>q<="011";
    when 1 to 30 =>q<="010"; -- Good
    when others =>q<="000";
end case;
end process;
end;
```
Use of concatenation - bad solution

- Is it not permissible to use concatenation to combine the vectors as one choice.
- This is because the **case** `<expression>` must be static.

```vhdl
-- E R R O R !
entity case_ex5 is
port (a,b: in std_logic_vector(2 downto 0);
    q: out std_logic_vector(2 downto 0));
end;
architecture rtl of case_ex5 is
begin
p1: process(a,b)
beginn
case a & b is
    -- Error
    when "000000" => q<="011";
    when "001110" => q<="010";
    when others => b<="000";
end case;
end process;
end;
```

ETEC 301- DCS-II
Mr. GAURAV VERMA
Use of concatenation applying variable

The solution is either to introduce a variable in the process to which the value $a & b$ is assigned or to use what is known as a qualifier for the subtype.

```vhdl
entity case_ex6 is
port (a, b: in std_logic_vector(2 downto 0);
     q: out std_logic_vector(2 downto 0));
end;
architecture rtl of case_ex6 is
begin
p1: process(a, b)
variable int: std_logic_vector (5 downto 0);
begin
int := a & b;
case int is
    when "000000" => q<="011";
    when "001110" => q<="010";
    when others => b<="000";
end case;
end process;
end;
```

ETEC 301- DCS-II               Mr. GAURAV VERMA
Use of concatenation applying qualifier

```vhdl
entity case_ex7 is
port (a, b: in std_logic_vector(2 downto 0);
     q: out std_logic_vector(2 downto 0));
end;
architecture rtl of case_ex7 is
begin
  p1: process(a, b)
  subtype mytype is std_logic_vector (5 downto 0);
  begin
    case mytype'(a & b) is
    when "000000" => q<="011";
    when "001110" => q<="010";
    when others => b<="000";
  end case;
end process;
end;
```
Multiple assignment

- In the concurrent part of VHDL you are always given a driver for each signal assignment, which is not normally desirable.
- In the sequential part of VHDL it is possible to assign the same signal several times in the same process without being given several drivers for the signal.
- This method of assigning a signal can be used to assign signals a default value in the process.
- This value can then be overwritten by another signal assignment.
- The following two examples are identical in terms of both VHDL simulation and the synthesis result.
Multiple assignment - examples

architecture rtl of case_ex8 is
begin
  p1: process (a)
  begin
    case a is
      when "00" =>  q1<="1";
        q2<='0';
        q3<='0';
      when "10" =>  q1<="0";
        q2<='1';
        q3<='1';
      when others =>  q1<="0";
        q2<='0';
        q3<='1';
    end case;
  end process;
end;

architecture rtl of case_ex9 is
begin
  p1: process (a)
  begin
    q1<="0";
    q2<='0';
    q3<='0';
    case a is
      when "00" =>  q1<="1";
      when "10" =>  q2<='1';
      when others =>  q3<='1';
      end case;
  end process;
end;
Multiple assignment - conclusions

- If we compare them, we can see that there are fewer signal assignments in the second example.
- The same principle can be applied if, for example, `if-then-else` is used.
- The explanation for this is that no time passes inside a process, i.e. the signal assignments only overwrite each other in sequential VHDL code.
- If, on the other hand, the same signal is assigned in different processes, the signal will be given several drivers.
Null statement

null_statement::<=
[ label : ] null ;

- The null statement explicitly prevents any action from being carried out.
- This statement means “do nothing”.
- This command can, for example, be used if default signal assignments have been used in a process and an alternative in the case statement must not change that value.
Null statement - example

architecture rtl of ex is
begin
  p1: process (a)
  begin
    q1<="0";
    q2<='0';
    q3<='0';
    case a is
      when "00" =>  q1<="1";
      when "10" =>  q2<='1';
      q3<='1';
      when others => null;
    end case;
  end process;
end;
Null statement - conclusions

- In the previous example, **null** could have been left out.
- Readability is increased if the null statement is included.
- If null is omitted, there is a risk that, if someone else reads the VHDL code, they will be uncertain whether the VHDL designer has forgotten to make a signal assignment or whether the line should be empty.
Wait statement

wait_statement ::= [ label : ] wait [ sensitivity_clause ]
[ condition_clause ]
[ timeout_clause ];

Examples:
- wait ;
  - The process is permanently interrupted.
- wait for 5 ns ;
  - The process is interrupted for 5 ns.
- wait on sig_1, sig_2 ;
  - The process is interrupted until the value of one of the
    two signals changes.
- wait until clock = '1' ;
  - The process is interrupted until the value of clock is 1.
Wait statement in a process

There are four ways of describing a wait statement in a process:

- \textbf{process} (a,b)
- \textbf{wait until} a=1
- \textbf{wait on} a,b;
- \textbf{wait for} 10 ns;

The first and the third are identical, if \textbf{wait on} a,b; is placed at the end of the process:

\begin{verbatim}
p0: process (a, b) begin if a>b then q<='1'; else q<='0'; end if; end process;
p1: process begin if a>b then q<='1'; else q<='0'; end if; wait on a,b; end process;
\end{verbatim}
Features of the wait statement

- In the first example the process will be triggered each time that signal *a* or *b* changes value (a’event or b’event)
- **Wait on** *a*, *b*; has to be placed **at the end** of the second example to be identical with the first example because all processes are executed at stat-up until they reach their first **wait** statement.
- **That process also executed at least once, which has sensitivity list and there is no changes in the values of the list members**
- If a **wait on** is placed anywhere else, the output signal’s value will be different when simulation is started.
- If a sensitivity list is used in a process, it is not permissible to use a **wait** command in the process.
- It is permissible, however, to have several **wait** commands in the same process.
Details of the wait’s types

- **Wait until** \( a = '1' \); means that, for the **wait** condition to be satisfied and execution of the code to continue, it is necessary for signal \( a \) to have an event, i.e. change value, and the new value to be ‘1’, i.e. a rising edge for signal \( a \).

- **Wait on** \( a, b; \) is satisfied when either signal \( a \) or \( b \) has an event (changes value).

- **Wait for** 10 ns; means that the simulator will wait for 10 ns before continuing execution of the process.

  - The starting time point of the waiting is important and not the actual changes of any signal value.
  - It is also permissible to use the wait for command as follows:
    
    ```
    constant period:time:=10 ns;
    wait for 2*period;
    ```

- The wait alternatives can be combined into: **wait on** \( a \) **until** \( b = '1' \) **for** 10 ns;\), but the process sensitivity list must never be combined with the wait alternatives

- Example: wait until \( a = '1' \) for 10 ns;

  The **wait** condition is satisfied when \( a \) changes value or after a wait of 10 ns (regarded as an or condition).
Examples of wait statement

- **type** a: **in** bit; c1, c2, c3, c4, c5, c6, c7: **out** bit;

**Example 1**

```vhdl
process (a)
begin
  c1 <= not a;
end process;
```

**Example 2**

```vhdl
process
begin
  c2 <= not a;
  wait on a;
end process;
```

**Example 3**

```vhdl
process
begin
  wait on a;
  c3 <= not a;
end process;
```

**Example 4**

```vhdl
process
begin
  wait until a='1';
  c4 <= not a;
end process;
```

**Example 5**

```vhdl
process
begin
  c5 <= not a;
  wait until a='1';
end process;
```

**Example 6**

```vhdl
process
begin
  c5 <= not a;
  wait for 10 ns;
end process;
```

**Example 7**

```vhdl
process
begin
  c5 <= not a;
  wait until a='1' for 10 ns;
end process;
```
Simulation results

---

A

---

10 20 30 40 50 60

time [ns]

C1

---

C2

---

C3

---

C4

---

C5

---

C6

---

C7

---

10 ns

---

20 ns

---

ETEC 301-DCS-II

Mr. GAURAV VERMA
Wait statement in synthesis tools

- Synthesis tools do not support use of `wait for 10 ns;`.
- This description method produces an error in the synthesis.
- The majority of synthesis tools only accept a sensitivity list being used for combinational processes, i.e. example 1, but not example 2, can be synthesized.
- But some advanced systems accept example 2, while example 3 is not permitted in synthesis.
- Example 4 is a clocked process and results in a D-type flip-flop after synthesis.
- Examples 3 and 5 can only be used for simulation, and not for design.
- The wait command is a sequential command, it is not permissible to use `wait` in functions, but it can be used in procedures and processes.
An Example

Combinational

Entity rsff is
Port ( set, reset: IN Bit;
    q, qb : INOUT Bit);
End rsff;

Architecture netlist of rsff is
Component nand2
port (a, b : in bit;
    c: out bit);
End component
Begin
U1 : port map (set, q, qb, q);
U2 : port map (reset, q, q, qb);
End first;

architecture sequential of rsff is
Begin
Process(set, reset)
Begin
If set='1' and reset = '0' then
q<='0' after 2ns;
qb <= '1' after 4ns;
elseif set='0' and reset = '1' then
q<='1' after 4ns;
qb <= '0' after 2ns;
elseif set='0' and reset = '0' then
q<='1' after 2ns;
qb <= '1' after 2ns;
Endif;
End process;
End first;

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Qb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Entity 3add is
Port ( a,b,c: std_logic;
    z: out std_logic);
End 3add;

Architecture first of 3add is
Begin
    z <= a and b and c;
End first;

Architecture second of 3add is
Begin
    Process (a,b,c)
    Variable temp:std_logic;
    Begin
        temp := b and c;
        z <= a and temp;
    End;
End second;
Synthesis Example

Entity 3sel is
Port (a,b,c: std_logic;
     z: out std_logic);
End 3sel;

- A concurrent signal assignment that requires sequential hardware

Architecture cond of 3add is
Begin
z<= a when b='1'
    else b when c='1'
    else z;
End first;