Sequential VHDL

Katarzyna Radecka

DSD COEN 313
kasiar@ece.concordia.ca
Overview

- Process
- Sensitivity List
- Wait Statements
- If Statements
- Case Statements
- Loop Statements
Three Styles of VHDL

**Behavioral**

**Boolean**

\[ c = a \lor b \]

**VHDL**

\[ c <= a \lor b; \]

**Structural**

**RTL level**

**VHDL**

COMPONENT and_gate

\[
\begin{align*}
& \text{port}(
& \quad \text{i1, i2: in std_logic;}
& \quad \text{o: out std_logic};
& \text{END COMPONENT;}
& \text{\ldots}
& \text{G0: and_gate}
& \text{port map(}
& \quad \text{i1 => a, i2 => b,}
& \quad \text{o1 => c);}
\end{align*}
\]

**Timing**

**Schematic**

\[ c <= a \lor b \text{ after 10 ns;} \]
Modeling Styles

- Possibly many different architecture bodies of one entity corresponding to alternative implementations performing the same function (design functionality)

- Styles of architecture body
  - Behavioral
  - Structural
  - Mixed structural/behavioral
Behavioral Architecture Body

- Describes function in abstract way
- Include process statements
  - Collections of actions executed in sequence
    - Similar to standard programming languages like C
    - VHDL in general concurrent language – all statements outside processes executed in parallel (like in hardware)
- Sequential statements include: evaluation of expressions, assignment of values to variables and signals, conditional executions, repeated executions and subprogram calls
Process

- Typical VHDL feature
- Useful for better visualizing changes happening in designs
- Statements executed sequentially
  - Only for simulation purposes – not extended to synthesis
    - All sequentially executed combinational instructions synthesized to parallel ones
- Order of coding instructions matters
  - Similarly to C/C++ sequential flow may be broken by if/else, case, etc. instructions
Process – Syntax 1

- Starts with key word `process`
  - Optionally label can be assigned to process

- Sensitivity list
  - Placed in () right after key word `process`
  - List of all signals to which is sensitive
    - Upon changing of values of at least one of signals process is invoked
  - Relevant only at pre-synthesized level (pre-synthesized simulations)
  - Transparent for post-synthesis netlists
    - Circuits produce output for any set of inputs regardless of whether all inputs were declared on the sensitivity list
ma: process(a, b, d, c_in)
    variable m_tmp: bit_vector(7 downto 0) := "00000000";
begin
    mult_tmp := a * b;
    sum <= mult_tmp + d + c_in;
end process;
Sensitivity List

- List of signals to which process is sensitive
  - Event on any signal from the list triggers activation of process
    - Sequential execution of instructions until last one in process
      - Process is then suspended waiting for the next event on sensitivity list
Sensitivity List – What to Put There?

- Sensitivity list of processes describing combinational designs include all signals inputted to process
  - If some process input signals omitted from sensitivity list, then events on these signals not triggering process execution
    - If process activated by changes of other signals from sensitivity list, then calculations inside process not reflecting changes of values of signals not on the sensitivity list
Example: Sensitivity List

or_gate: process (a)
begin
  c <= a or b;
end process;

b not in Sensitivity list

Pre-synthesis simulations

Synthesized netlist

Post-synthesis waveform
Example: Sensitivity List, cont.

**entity** mux **is**
  **port** (data_0,
            data_1, sel: in std_logic;
            output: out std_logic);
**end** mux;

**architecture** incom **of** mux **is**
**begin**
  **process** (data_1, sel)
    **begin**
      **if** (sel = ‘0’) **then**
        output <= data_0;
      **else**
        output <= data_1;
    **end** **process**;
**end** incom;

Changes on data_0 not reflected at output
Sensitivity List and Synthesis

- Processes and sensitivity list only on simulation level
- Incorrect simulations due to incomplete sensitivity list transparent to synthesized circuit
  - Circuits simulating with errors due to incomplete sensitivity list synthesized into correct netlist
    - Potential problems in verification, when post synthesis netlist checked vs. pre-synthesis RTL
Sensitivity List and Wait

- Process can be suspended by means of sensitivity list
- Wait statement – alternative way of suspending processes
- Wait statements and sensitivity lists are mutually exclusive
  - Process sensitivity list signals are implicitly *wait on*
- Process suspended upon encountering wait statement
  - Reactivated when wait period ends
Process Execution

- **ACTIVE**: Simulator startup, Event on signal in sensitivity list, Wait expires
- **SUSPENDED**: Execute a wait, Reach end of the process
Wait Statements

- Suspends execution of process
- Three basic kinds of *wait* statement
  - `wait on sensitivity-list;`
  - `on signals`
  - `wait until sensitivity-list;`
  - `until conditions`
  - `wait for sensitivity-list;`
  - `for signals`
- Combinations of above conditions in single *wait* statement also possible
  - `wait on sensitivity-list until boolean-expression for time-expression;`
Wait On

- Process suspends and wait for event to occur on signals A, B, or C
  
  - Upon event on A, B, or C process resumes execution from next statement onwards
  
  - When wait is the last instruction, process resumes from first statement
Wait Until

- Process suspends until specified condition becomes true
  - Upon encountering even on signals A or B
    - “=“ condition evaluated
    - If fulfilled process resumes operation from next statement onwards, otherwise it remains suspended
Wait For

- Execution of wait statement at time T causes suspense of process for 10 ns
  - Next process statement executed at time T+10ns

\textbf{wait for} 10 ns;
Wait On Sensitivity-List For
Time-Expression

- Process suspended until event on CLOCK for a timeout of 20 ns
  - If no event on CLOCK within 20 ns process resumes operation from execution of next statement

```plaintext
wait on CLOCK for 20 ns;
```
Wait Until *Boolean-Statement*
For *Time-Expression*

- Process suspends for max 50 ms until value of SUM is greater than 100
  - Boolean condition evaluated every time event on SUM happens
- If Boolean condition not satisfied for 50 ms, process resumes execution from next statement following wait
Wait/Sensitivity List

- If no explicit sensitivity list in process, then at least one wait statement should be in process
  - Otherwise process never get suspended and remains in infinite loop during initialization phase of simulation
    - Error reported if both sensitivity list and wait statements present in single process
Example: Wait Statements

half_add: process is
begin
  sum <= a XOR b after T_pd;
  carry <= a AND b after T_pd;
  \texttt{wait on} a, b;
end process;

half_add: process(a,b) is
begin
  sum <= a XOR b after T_pd;
  carry <= a AND b after T_pd;
-- not \texttt{wait on} a,b; needed
end process;
Clock Generation

- One of ways of clock generation for SIMULATION purposes only (testbenches) using wait for statement

```vhdl
entity clk is
  port(clk: out std_logic);
end clk;

architecture cyk of clk is
begin
  process
  begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;
end cyk;
```

No inputs to process -> no sensitivity list needed
Process: Allowed Statements

- Only following sequential statements allowed inside process:
  - **if** statement
  - **case** statement
  - **wait** statement
  - **loop** statement (for, while)

- Additionally allowed signal and variable assignments

- Concurrent statements such as **when** and **with** not allowed inside processes
If Statement

- Used to select sequence of statements for execution based on fulfilling some condition
- Condition – any expression evaluating to Boolean value

```plaintext
if boolean-expression then
    sequential-statements
{elsif boolean-expression then  -- elsif clause; if can have 0 or
    sequential-statements}
[else  -- more elsif clauses
    sequential-statements]
end if;
```
If/else - Syntax

**Boolean expression**

if \( (\text{clk}'\text{event}) \text{ and } (\text{clk} = '1') \) then
q1 <= d1;
q2 <= d2;
end if;

**Modeling of rising clock edge**

if \( k = 0 \) then
var1 := a+1;
elsif \( k = 1 \) then
var2 := b+a;
else
var1 := '0';
var2 := '0';
end if;

**Key word**

sequential statements

if \(((\text{clk}'\text{event}) \text{ and } (\text{clk} = '1'))\) then
q1 <= d1;
q2 <= d2;
end if;
Evaluation of If Statement

- Executed by checking each condition sequentially until the true condition found or all cases exhausted
- Else and elsif statements optional
- Possibility of nesting if statements
if grade_per < 40 then
    grade_letter <= 'R';
elsif ((grade_per > 41) and (grade_per < 50)) then
    grade_letter <= 'D';
elsif ((grade_per > 51) and (grade_per < 60)) then
    grade_letter <= 'C';
elsif ((grade_per > 61) and (grade_per < 70)) then
    grade_letter <= 'B';
elsif (grade_per > 71) then
    grade_letter <= 'A';
else
    grade_letter <= 'F';
end if;

if cntrl1 = '0' then
    if cntrl2 = '0' then
        out_a <= "00";
    else
        out_a <= "01";
    end if;
else
    if cntrl2 = '0' then
        out_a <= "10";
    else
        out_a <= "11";
    end if;
end if;
Case Statement

- Sequential counterpart of with-select concurrent statement

```
Case expression is
  when choices => sequential-statements   -- branch 1
  when choices => sequential-statements   -- branch 2
  -- any number of branches can be specified
[when others => sequential-statements]   -- last branch
end case;
```

- Only one branch with matching condition selected for execution
Selection choices represented as single value or range of values joined by or operand represented as |, or by others clause

- All possible values must be covered in case exactly once
- *Others* clause optionally used to implicitly represent all missing explicit values of expression
  - If present, located as last branch in case statement
Case Syntax

Note: All values of expression must be covered by case statement. If some values are excluded, then “when others” expression is added to the end of all listed case conditions.

type alu is (comp, add, sub);
variable alu_sel: alu;
case alu_sel is
  when comp =>
    data_out <= inp1;
  when add =>
    data_out <= inp1+inp2;
  when others =>
    data_out <= inp2;
end case;
Example: Case Statement

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

entity process_mux is
port( data_0, data_1 : in std_logic;
     select_input : in std_logic;
     output : out std_logic);
end process_mux;

architecture using_case of process_mux is
begin -- of architecture

-- define a process with optional name (Mux_Process)
Mux_Process: process( data_0, data_1, select_input)
begin -- of process Mux_Process

case select_input is
    when '0' => output <= data_0;
    when others => output <= data_1;
end case;

end process;

end using_case;
```
If vs. Case

- If/else – used when behavior of design may depend on value of *more than one* signal expression
- Case – used when behavior of design may depend on value of *one* signal expression
- **If/else**’s are synthesized as cascaded MUXes; a case statement is a single MUX (only 2 levels of logic!)
  - If area is critical for design use if/else structure
    - Result – small but slow design
  - If time is critical for design use case statements
    - Result – fast but bigger designs
Loop Statements

- Used to iterate through a set of sequential statements
- Syntax

```
[loop label: ] iteration-scheme loop
  sequential-statements
end loop [loop label];
```
Loop - Three Forms

sum := 0;

L1: for i in 1 to N loop
    sum := sum + 1;
end loop;

j := 0;
sum := 10;
while j < 20 loop
    sum := sum*2;
    j := j+1;
end loop;

j := 0;
sum := 1;

L2: loop
    sum := sum*10;
    j := j+1;
    exit L2 when sum > 100;
end loop;

j := 0;
sum := 10;
while j < 20 loop
    sum := sum*2;
    j := j+1;
end loop;
Loop For

- Syntax

  ```
  for identifier in range
  ```

- Example: calculation of \( y^N, N > 2 \)

  ```
  power_N := y;
  for iter in 2 to N loop
    power_N := power_N * y;
  end loop;
  ```

- Body of loop executed \( N-1 \) times
Loop For, cont.

- **Body of loop executed** $N-1$ **times**
  - Loop identifier $i$ incremented by 1 at the end of each iteration
  - $i$ implicitly declared within for loop to belong to integer type of the range 2 to $N$

- **No explicit declaration of loop identifier needed**
  - When other variable of name $i$ exists outside for loop then these two variables treated separately, with $i$ from the for loop assigned only to the loop
Range of For Loop

- Range can be integer, but also enumeration type
  - To denote hexadecimal numbers
  - Example: type ht is (‘0’,’1’, …,’A’,…,’F’)

```ml
for i in ht’('A') to ('F") loop
```
Loop While

- **Syntax**
  
  ```
  while boolean-expression
  ```

- **Example:**
  
  ```
  power_N := y;
  while i < N loop
    power_N := power_N * y;
  end loop;
  ```
Loop While, cont.

- Statements inside loop executed sequentially and repeated until loop condition is true
  - Execution of loop terminated when loop condition becomes false
Loop Statement

- No iteration scheme specified
- Statements inside loop repeatedly executed until some action causes loop termination
  - `exit`, `next`, `return` statements used to terminate loop
    - If no exit statements, then loop executed indefinitely

```plaintext
power_N := y;
L: loop
  power_N := power_N * y;
  exit when power_N > 250;
end loop L;
```
Processes Communication

- More than one process possible in a single architecture body
- Each process in architecture body executed concurrently with other processes and other concurrent VHDL statements
  - Remember, that all statements inside processes executed sequentially
Process Communication, cont.

- Communication among processes via internal signals
  - Values of internal signals generated in process A may be used as input signals in process B
    - Placed on process B sensitivity list
Example: Microprocessor

- Simple CPU used to control read and write operations to memory
  - CPU imposes memory read and write operations through signals data_read and data_write
  - Memory manifests readiness of accepting new data by setting signal mem_ready

- Memory location of instruction to be executed stored in PC register
  - Length of each instruction 2 bytes (16 bites)
  - Memory address space 8 bits
VHDL Code of Microprocessor

```vhdl
signal address: integer;
signal mem_read: word;
signal mem_write: word;
signal data_read: std_logic;
signal data_write: std_logic;
signal mem_ready: std_logic;
begin
  cpu: process
  variable PC: integer;
  variable instr_reg: word;
  begin
    loop
      address <= PC;
      data_read <= '1';
      wait until mem_ready = '1';
      instr_reg <= mem_read;
      wait until mem_ready = '0';
      PC := PC+2;
    end loop;
  end process cpu;

  memory: process
  type data_type is array (0 to 63) of word;
  variable store: data_type;
  begin
    wait until data_read = '1' or data_write = '1';
    if data_read = '1' then
      data_read <= store(PC/2);
      data_out <= store(PC/2);
      mem_ready <= '1';
      wait until data_read = '0';
      mem_ready <= '0';
    elsif data_write = '1' then
      store(PC/2) <= data_in;
      mem_ready <= '1';
      wait until data_write = '0';
      mem_ready <= '0';
    end if;
  end process memory;
```

Multiple Signal Assignments

- Signals can be assigned values several times within a process:
  - All of intermediate assignments invalid
  - Value assigned as last in single process execution cycle counts
    - Signal assignment does not take place until process execution finishes

- If value of data must be changed several times in one process use variable instead of signal to represent it
Example: Multiple Signal Assignments - Concurrent

Compilation failed
Example: Multiple Signal Assignments - Process

library IEEE;
use IEEE.std_logic_1164.all;

entity sig_ass_proc is
  port(
    data0, data1 : in integer;
    data_out : out integer);
end sig_ass_proc;

architecture behavioral of sig_ass_proc is
  signal test_sig1: integer;
  signal test_sig2: integer;

begin
  process(data0, data1)
  begin
    test_sig1 <= data0;
    test_sig2 <= test_sig1 + data1;
    test_sig1 <= test_sig2;
    data_out <= test_sig1;
  end process;
end behavioral;

Compilation passed
Example: Multiple Signal Assignments – Process (Sims)

Wrong simulation results
Example: Multiple Variable Assignments

Compilation passed

data1 ignored as not visible at outputs
Example: Multiple Variable Assignments, cont.

Compilation and simulations passed