VHDL Sequential Statements
VLSI CAD
Case Western Reserve University

Dan Saab

Sequential Statements

- They manipulate data within processes.
- They are executed in sequence.

Sequential Statements

- **wait statement**
- **assertion statement**
- **report statement**
- **assignment statement**
  - **Variable**
  - **Signals**
- **procedure call statement**
- **if statement**
- **case statement**
- **loop statement**
- **next statement**
- **exit statement**
- **return statement**
- **null statement**

wait statement

- Causes the execution to wait.

[label:] wait [sensitivity clause] [condition clause];

```
wait for 10 ns;            -- timeout clause, specific time delay.
wait until clk='1';        -- condition clause, Boolean condition
wait until A>B and S1 or S2; -- condition clause, Boolean condition
wait on sig1, sig2;        -- sensitivity clause, any event on any
                           -- signal terminates wait
```

- The wait until form suspends a process until a change occurs on one or more of the signals in the statement and the condition is evaluated to be true. A rising edge on NET_DATA_VALID and three rising edges on CLK must occur for this process to cycle:

```
READ_NET: process
begin
  wait until NET_DATA_VALID='1';
  NET_DATA_READ <= '1';
  wait until CLK='1';
  wait until CLK='1';
  NET_BUFFER <= NET_DATA_IN;
  wait until CLK='1';
  NET_DATA_READ <= '0';
end process READ_NET;
```

assertion statement

- Used for internal consistency check or error message generation.
- The assert statement tests the boolean condition. If this is false, it outputs a message containing the report string to the simulator screen. If the message clause is ommited, a default message is output.

[label:] assert boolean_condition [report string] [severity name] ;

```
assert a=(b or c);
assert j<i report "internal error, tell someone";
assert clk='1' report "clock not up" severity WARNING;
```

Functional errors, timing errors can be reported via assert:

```
entity SRflipflop is  port(S,R:in bit; Q:out bit);  end entity SRflipflop;
architecture checking of SRflipflop is
begin
        setreset: process (S, R) is
        begin
                assert S = '1' nand R = '1';
                if S = '1' then
                        Q <= '1';
                end if;
                if R='1' then
                        Q <= '0';
                end if;
        end process setreset
end architecture checking;
```

Functional errors, timing errors can be reported via assert:

```
entity edgetrig is port ( clock: in bit; din : in real; dout: out real); end entity edgetrig;
architecture checktiming of endtrig is
begin
        store: process (clock) is
        variable storedv : real;
        variable pulsestart : time;
        begin
                case clock is
                        when "1" =>
                                pulsestart := now;
                                storedv := din;
                                dout <= storedv;
                        when "0" =>
                                assert now = 0 ns or (now- pulsestart) >= 5 ns
                                                report "clock pulse too short";
                        end case;
        end process store;
end architecture checktiming;
```

An unconditional message can be output by using the literal false

```
if (POINTER < 5) then
        STACK(POINTER) <= ITEM;
        POINTER <= POINTER + 1;
else
        assert false report "Stack overflow“ severity error;
        end if;
```

report statement

- Used to output messages.

[label:] report string [severity name] ;

- Examples

```
report  "finished pass1";  -- default severity name is NOTE
report  "Inconsistent data." severity FAILURE;
```

IF statement

- It allows selection between alternative actions
- control structures

```
[label:] if  condition1  then
                sequence-of-statements
        elsif  condition2  then
                sequence-of-statements
         elsif  condition3  then
                sequence-of-statements
                 ...

        else
                sequence-of-statements
        end if [ label ] ;
```

IF statement (Examples)

```
if en = ’1’ then
        regvalue := datav;
end if
if muxsel = 0 then
        muxout <= i0; -- executed if muxsel = 0
else
        muxout <= i; -- executed if muxsel /= 0
end if

-- starts by evaluating the first condition
-- successive conditions are eval in order until one found true
        if mode = immediate then
                operand := immedoperand;
        elsif opcode = load or opcode = add then
                operand := memoperand;
        else
                operand := addoperand;
        end if
```

- An if statement may be used to infer edge-triggered registers in a process sensitive to a clock signal. Asynchronous reset may also be modelled:

```
process(CLK, RESET)
begin
        if RESET = '1' then
                COUNT <= 0;
        elseif CLK'event and CLK='1' then
                if (COUNT >= 9) then
                        COUNT <= 0;
                else
                        COUNT <= COUNT + 1;
                end if;
        end if
end process;
```

Example: doorcontroller

```
entity doorcontrol is
        port ( desiredvalue, sensorinput : in integer; indicatore : out boolean);
end entity thermostat;

architecture example of doorcontrol is
constant delta : integer := 2;
begin
        controller: process (desiredvalue, sensorinput) is
        begin
                if desiredvalue < sensorinput-delta then
                        indicatore <= true;
                elsif desiredvalue > sensorinput+delta then
                        indicatore <= false;
                end if;
        end process controller;
        …
end architecture example
```

case statement

- Execute one specific case of an expression equal to a choice.
- The choices must be constants of the same discrete type as the expression.
- All possible choices must be included, unless the others clause is used as the last choice:

```
[ label: ] case  expression  is
        when choice1 =>
          sequence-of-statements
        when choice2 =>              -- optional
          sequence-of-statements  -- optional
        ...

        when others =>         -- optional if all choices covered
          sequence-of-statements  -- optional
       end case [label ] ;
```

CASE statement

```
architecture example of ent is
        type aluop is (pass, add, sub);
begin
        alu: process ( sensitivity list )
        variable func: aluop;
        begin
                ...
                case func is
                        when pass =>
                                result := op1;
                        when add =>
                                result := op1 + op2 ;
                        when sub =>
                                result := op1 + op2;
                end case;
                ...
        end process alu;
        …
end architecture example;
```

CASE statement

```
library ieee; use ieee.std logic 1164.all;
entity mux4 is
        port ( sel : in sel_range; d0, d1, d2, d3 : in std_logic; z : out std logic);
end entity mux4;
architecture demo of mux4 is
begin
        outselect: process (sel, d0, d1, d2, d3) is
        begin
                case sel is
                        when 0 =>
                                z <= d0;
                        when 1 =>
                                z <= dl;
                        when 2 =>
                                z <= d2;
                        when 3 =>
                                z <= d3;
                end case;
        end process outselect;
end architecture demo;
```

A range or a selection may be specified as a choice:

```
case INT_A is
        when 0        =>  Z <= A;
        when 1 to 3   =>  Z <= B;
        when 4|6|8    =>  Z <= C;
        when others   =>  Z <= 'X';
end case;
```

Choices may not overlap

```
case INT_A is
  when 0      =>  Z <= A;
  when 1 to 3 =>  Z <= B;
  when 2|6|8
            =>  Z <= C;   -- illegal
  when others =>  Z <= 'X';
end case;
```

A range may not be used with a vector type

```
case VEC is
  when "000" to "010"
            =>  Z <= A;   -- illegal
  when "111"   =>  Z <= B;
  when others  =>  Z <= 'X';
end case;
```

Null statements

- States that no action is perform

```
subtype indexmode is integer range 0 to 7;
variable IR: integer range 0 to 2**16 - 1
...
-- res = a rem b res = a - (a/b)*b
      case indexmode'((IR/2**12) rem 2**3) is
      when 0 =>          IndexValue := 0;
      when 1 | 2 =>      IndexValue := acc ;
      when 4 to 5 =>     IndexValue := Acc1;
      when 7 downto 6 => IndexValue := IndexReg1;
                         IndexValue := IndexReg1;
      when others =>        null;
end case;
nullsection : process (sensitivity-list) is
begin
                null;
end process nullsection;
```

Loop statements

- Loop repeat a sequence os statements indefinitely

```
[label:]    looplabel: loop
                ..
                sequential statement
                ..
            end loop looplabel
```

Example

```
entity counter is
        port ( clk: in bit; count: out natural);
end entity counter;
architecture behavior of counter is
begin
        incrementer: process is
                variable countv : natural := 0;
                begin
                        count <= countv;
                        loop
                                wait until clk = '1';
                                countv := (countv + 1) mod 16;
                                count <= countv;
                        end loop;
                end process incrementer;
end architecture behavior;
```

Exit statement

- Syntax

  [label:] exit [label2] [when condition] ;

- When this statement is executed, any remaining statements in the loop are skipped, and control is transferred to the statement after the end loop keywords loop

Exit statement

```
loop
        if condition then
                exit;
        end if;
        exit when condition;
end loop;
-- control is transfer here
```

```
entity counter is
        port ( clk, reset: in bit; count: out natural);
end entity counter;
architecture behavior of counter is
begin
        incrementer: process is
        variable countv : natural := 0;
        begin
                count <= countv;
                loop
                        loop
                                wait until clk = '1' or reset = '1' ;
                                exit when reset = '1';
                                countv := (countv + 1) mod 16;
                                count <= countv;
                        end loop;
                        -- at this point, reset= '1'
                        countv := 0;
                        count <= countv;
                        wait until reset = '0';
                end loop;
        end process incrementer;
end architecture behavior;
```

Exit transfers control out of a specific loopname

```
outer: loop
...
        inner: loop
            exit outer when cond1; -- transfer control to B
            exit when cond2; -- transfer control to A
        end loop inner;
        A statement
        ...
        exit outer when cond3; -- transfer control to B
        ...
end loop outer;
B statement
...
```

next statement

- Control the execution of a loop: When this statement is executed, the current iteration of the ioop is completed without executing any further statements, and the next iteration begins.

[label:] next [label2] [when condition] ;
;

– next;
– next looplabel;
– next looplabel when condition

next statement (Example)

```
loop
        stat1;
        next when condition
        stat2;
end loop;
```

```
loop
        stat1;
        if not condition
        then
                stat2;
        end if;
end loop
```

while statement

- Syntax:

```
[ label: ] while  condition  loop
                    sequence-of-statements
                end loop [ label ] ;
```

- while loop tests the condition before each iteration. If the condition is true, iteration proceeds. if it is false, the loop is terminated.

For Loop

- Syntax:

```
[ label: ] for variable in range loop
                    sequence-of-statements
                end loop [ label ] ;
```

- The identifier is called the loop parameter, and for each iteration of the loop, it takes on successive values of the discrete range, starting from the left element.

Example cos(a)=1-a^2/2!+ a^4/4!- a^6/6!+…

```
entity cos is
        port (theta: in real; result: out real);
end entity cos;
architecture series of cos is
begin
        summation : process (theta) is
        variable sum, term : real;
        variable n : natural;
        begin
                sum := 1.0
                term := 1.0;
                n =0;
                while abs term > abs (sum / 1.0E6) loop
                        n := n + 2;
                        term := (-term)*theta**2 / real(((n-1) *n))
                        sum := sum + term;
                end loop;
                result <= sum;
        end process summation;
end architecture series;
```

For loop: index hiding

- Example

```
for count in 0 to 127 loop
        outname <= count;
        wait for 5 ns;
end loop;
```

- Loop parameter hides any object of the same name defined outside the loop. The loop parameter does not need to be declared: It may not be modified within the loop:

```
variable a, b: integer;
begin
a := 10;
for a in 0 to 7 loop
        b := a;
end loop;
                -- a= 10, and b=7
```

Attributes such as 'high, 'low and 'range may also be used to define the iterations of a for loop:

```
process (A)
        variable TMP : std_ulogic;
begin
        TMP := '0';
        for I in A'low to A'high loop
                TMP := TMP xor A(I);
        end loop;
        ODD <= TMP;
end process;
```

The range may be any discrete range, e.g. an enumerated type:

```
type PRIMARY is (RED, GREEN, BLUE);
type COLOUR is ARRAY (PRIMARY) of integer range 0 to 255;
-- other statements
MUX: process
begin
        for SEL in PRIMARY loop
                V_BUS <= VIDEO(SEL);
                wait for 10 ns;
        end loop;
end process MUX;
```

Example cos(a)=1-a^2/2!+ a^4/4!- a^6/6!+…

```
entity cos is  port (theta: in real; result: out real); end entity cos;
architecture fixed series of cos is
begin
        summation : process (theta) is
        variable sum, term : real;
        begin
                sum := 1.0;   term := 1.0;
                for n in 1 to 19 loop
                        term := (-term)*theta**2 / real(((n-1) *n));
                        sum := sum + term;
                end loop;
                result <= sum;
        end process summation;
end architecture fixed series;
```