

Sequential Statements

Outline

1. VHDL process
2. Sequential signal assignment statement
3. Variable assignment statement
4. If statement
5. Case statement
6. Simple for loop statement

1. VHDL Process

- Contains a set of sequential statements to be executed sequentially
 - The whole process is a concurrent statement
 - Can be interpreted as a circuit part enclosed inside of a black box
 - May or may not be able to be mapped to physical hardware
- Two types of process
 - A process with a sensitivity list
 - A process with wait statement

A process with a sensitivity list

- Syntax

```
process(sensitivity_list)
  declarations;
begin
  sequential statement;
  sequential statement;
  . . .
end process;
```
- A process is like a circuit part, which can be
 - active (known *activated*)
 - inactive (known as *suspended*).
 - A process is activated when a signal in the sensitivity list changes its value
 - Its statements will be executed sequentially until the end of the process

- E.g, 3-input and circuit

```

signal a,b,c,y: std_logic;
process(a,b,c)
begin
  y <= a and b and c;
end process;

```

- How to interpret this:

```

process(a)
begin
  y <= a and b and c;
end process;

```

- For a combinational circuit, all input should be included in the sensitivity list

A process with wait statement

- Process has no sensitivity list
- Process continues the execution until a wait statement is reached and then suspended
- Forms of wait statement:
 - **wait on** signals;
 - **wait until** boolean_expression;
 - **wait for** time_expression;

- E.g, 3-input and circuit

```

process
begin
  y <= a and b and c;
  wait on a, b, c;
end process;

```

- A process can has multiple wait statements
- Process with sensitivity list is preferred for synthesis

2. Sequential signal assignment statement

- Syntax
signal_name <= value_expression;
- Syntax is identical to the simple concurrent signal assignment
- Caution:
 - Inside a process, a signal can be assigned multiple times, but only the last assignment takes effect

- E.g.,

```

process(a,b,c,d)
begin
  y <= a or c;      -- yentry := y
  y <= a and b;     -- yexit := a or c;
  y <= c and d;     -- yexit := a and b;
end process;      -- yexit := c and d;
                  -- y <= yexit

```

- It is same as **process**(a,b,c,d)
- What happens if the 3 statements are concurrent statements?

3. Variable assignment statement

- Syntax
variable_name := value_expression;
- Assignment takes effect immediately
- No time dimension (i.e., no delay)
- Behave like variables in C
- Difficult to map to hardware (depending on context)

- E.g.,


```

process(a,b,c)
  variable tmp: std_logic;
begin
  tmp := '0';
  tmp := tmp or a;
  tmp := tmp or b;
  y <= tmp;
end process;

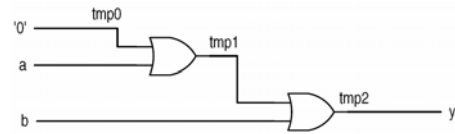
```

- interpretation:


```

process(a,b,c)
  variable tmp0, tmp1, tmp2: std_logic;
begin
  tmp0 := '0';
  tmp1 := tmp0 or a;
  tmp2 := tmp1 or b;
  y <= tmp2;
end process;

```



- What happens if signal is used?


```

process(a,b,c,tmp)
begin
  tmp <= '0'; -- tmp_entry := tmp
  tmp <= tmp or a; -- tmp_exit := '0';
  tmp <= tmp or b; -- tmp_exit := tmp_entry or a;
  tmp <= tmp or b; -- tmp_exit := tmp_entry or b;
end process; -- tmp <= tmp_exit

```
- Same as:


```

process(a,b,c,tmp)
begin
  tmp <= tmp or b;
end process;

```

4. IF statement

- Syntax
- Examples
- Comparison to conditional signal assignment
- Incomplete branch and incomplete signal assignment
- Conceptual Implementation

Syntax

```

if boolean_expr_1 then
  sequential_statements;
elsif boolean_expr_2 then
  sequential_statements;
elsif boolean_expr_3 then
  sequential_statements;
...
else
  sequential_statements;
end if;

```

E.g., 4-to-1 mux

```

architecture if_arch of mux4 is
begin
  process (a, b, c, d, s)
  begin
    if (s="00") then
      x <= a;
    elsif (s="01") then
      x <= b;
    elsif (s="10") then
      x <= c;
    else
      x <= d;
    end if;
  end process;
end if_arch;

```

input	output
s	x
00	a
01	b
10	c
11	d

E.g., 2-to-2² binary decoder

```
architecture if_arch of decoder4 is
begin
  process (s)
  begin
    if (s="00") then
      x <= "0001";
    elsif (s="01") then
      x <= "0010";
    elsif (s="10") then
      x <= "0100";
    else
      x <= "1000";
    end if;
  end process;
end if_arch;
```

input	output
s	x
0 0	0001
0 1	0010
1 0	0100
1 1	1000

RTL Hardware Design
by P. Chu

Chapter 5

19

E.g., 4-to-2 priority encoder

```
architecture if_arch of prio_encoder42 is
begin
```

```
  process (r)
  begin
    if (r(3)='1') then
      code <= "11";
    elsif (r(2)='1') then
      code <= "10";
    elsif (r(1)='1') then
      code <= "01";
    else
      code <= "00";
    end if;
  end process;
  active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

input	output	
r	code	active
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

RTL Hardware Design
by P. Chu

Chapter 5

20

Comparison to conditional signal assignment

- Two statements are the same if there is only one output signal in if statement
- If statement is more flexible
- Sequential statements can be used in then, elsif and else branches:
 - Multiple statements
 - Nested if statements

```
sig <= value_expr_1 when boolean_expr_1 else
value_expr_2 when boolean_expr_2 else
value_expr_3 when boolean_expr_3 else
. . .
value_expr_n;
```

It can be written as

```
process (...)
  if boolean_expr_1 then
    sig <= value_expr_1;
  elsif boolean_expr_2 then
    sig <= value_expr_2;
  elsif boolean_expr_3 then
    sig <= value_expr_3;
  . . .
  else
    sig <= value_expr_n;
  end if;
end process
```

RTL Hardware Design
by P. Chu

Chapter 5

21

RTL Hardware Design
by P. Chu

Chapter 5

22

e.g., find the max of a, b, c

```
if (a > b) then
  if (a > c) then
    max <= a; -- a>b and a>c
  else
    max <= c; -- a>b and c>=a
  end if;
else
  if (b > c) then
    max <= b; -- b>=a and b>c
  else
    max <= c; -- b>=a and c>=b
  end if;
end if;
```

RTL Hardware Design
by P. Chu

Chapter 5

23

e.g., 2 conditional sig assignment codes

```
signal ac_max, bc_max: std_logic;
. . .
ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;
```

```
max <= a when ((a > b) and (a > c)) else
c when (a > b) else
b when (b > c) else
c;
```

RTL Hardware Design
by P. Chu

Chapter 5

24

- 2 conditional sig assign implementations

```

signal ac_max, bc_max: std_logic;
. . .
ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;

max <= a when ((a > b) and (a > c)) else
      c when (a > b) else
      b when (b > c) else
      c;

```

e.g., “sharing” boolean condition

```

if (a > b and op="00") then
  y <= a - b;
  z <= a - 1;
  status <= '0';
else
  y <= b - a;
  z <= b - 1;
  status <= '1';
end if;

```

```

y <= a-b when (a > b and op="00") else
  b-a;
z <= a-1 when (a > b and op="00") else
  b-1;
status <= '0' when (a > b and op="00") else
  '1';

```

Incomplete branch and incomplete signal assignment

- According to VHDL definition:
 - Only the “then” branch is required; “elsif” and “else” branches are optional
 - Signals do not need to be assigned in all branch
 - When a signal is unassigned due to omission, it keeps the “previous value” (implying “memory”)

Incomplete branch

- E.g.,

```

process (a, b)
begin
  if (a=b) then
    eq <= '1';
  end if ;
end process;

```

- It implies

```

process (a, b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= eq;
  end if ;
end process

```

- fix

```

process (a, b)
begin
  if (a=b) then
    eq <= '1';
  else
    eq <= '0';
  end if ;
end process

```

Incomplete signal assignment

• E.g.,

```

process (a,b)
begin
  if (a>b) then
    gt <= '1';
  elsif (a=b) then
    eq <= '1';
  else
    lt <= '1';
  end if;
end process;
    
```

• Fix #1:

```

process (a,b)
begin
  if (a>b) then
    gt <= '1';
    eq <= '0';
    lt <= '0';
  elsif (a=b) then
    gt <= '0';
    eq <= '1';
    lt <= '0';
  else
    gt <= '0';
    eq <= '0';
    lt <= '1';
  end if;
end process;
    
```

• Fix #2

```

process (a,b)
begin
  gt <= '0';
  eq <= '0';
  lt <= '0';
  if (a>b) then
    gt <= '1';
  elsif (a=b) then
    eq <= '1';
  else
    lt <= '1';
  end if;
end process;
    
```

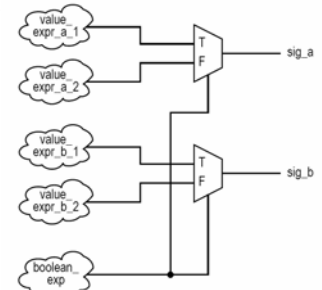
Conceptual implementation

- Same as conditional signal assignment statement if the if statement consists of
 - One output signal
 - One sequential signal assignment in each branch
- Multiple sequential statements can be constructed recursively

e.g.

```

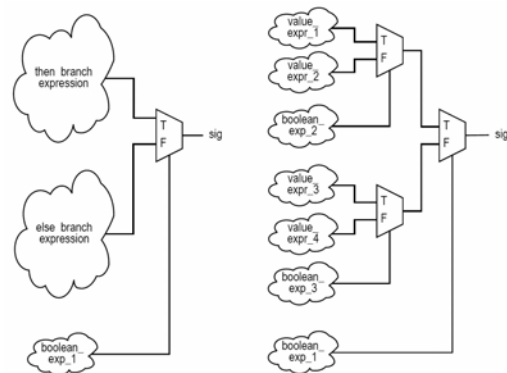
if boolean_expr then
  sig_a <= value_expr_a_1;
  sig_b <= value_expr_b_1
else
  sig_a <= value_expr_a_2;
  sig_b <= value_expr_b_2;
end if;
    
```



e.g.

```

if boolean_expr_1 then
  if boolean_expr_2 then
    signal_a <= value_expr_1;
  else
    signal_a <= value_expr_2;
  end if;
else
  if boolean_expr_3 then
    signal_a <= value_expr_3;
  else
    signal_a <= value_expr_4;
  end if;
end if;
    
```



5. Case statement

- Syntax
- Examples
- Comparison to selected signal assignment statement
- Incomplete signal assignment
- Conceptual Implementation

Syntax

```

case case_expression is
  when choice_1 =>
    sequential statements;
  when choice_2 =>
    sequential statements;
  . . .
  when choice_n =>
    sequential statements;
end case;
    
```

E.g., 4-to-1 mux

```

architecture case_arch of mux4 is
begin
  process (a,b,c,d,s)
  begin
    case s is
      when "00" =>
        x <= a;
      when "01" =>
        x <= b;
      when "10" =>
        x <= c;
      when others =>
        x <= d;
    end case;
  end process;
end case_arch;
    
```

input	output
s	x
00	a
01	b
10	c
11	d

E.g., 2-to-2² binary decoder

```

architecture case_arch of decoder4 is
begin
  proc1:
  process (s)
  begin
    case s is
      when "00" =>
        x <= "0001";
      when "01" =>
        x <= "0010";
      when "10" =>
        x <= "0100";
      when others =>
        x <= "1000";
    end case;
  end process;
END case_arch;
    
```

input	output
s	x
00	0001
01	0010
10	0100
11	1000

E.g., 4-to-2 priority encoder

```

architecture case_arch of prio_encoder42 is
begin
  process (r)
  begin
    case r is
      when "1000"|"1001"|"1010"|"1011"|"
          "1100"|"1101"|"1110"|"1111" =>
        code <= "11";
      when "0100"|"0101"|"0110"|"0111" =>
        code <= "10";
      when "0010"|"0011" =>
        code <= "01";
      when others =>
        code <= "00";
    end case;
  end process;
  active <= r(3) or r(2) or r(1) or r(0);
end case_arch;
    
```

input	output	active
r	code	
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

Comparison to selected signal assignment

- Two statements are the same if there is only one output signal in case statement
- Case statement is more flexible
- Sequential statements can be used in choice branches

```

with sel_exp select
  sig <= value_expr_1 when choice_1,
        value_expr_2 when choice_2,
        value_expr_3 when choice_3,
        . . .
        value_expr_n when choice_n;

```

It can be rewritten as:

```

case sel_exp is
  when choice_1 =>
    sig <= value_expr_1;
  when choice_2 =>
    sig <= value_expr_2;
  when choice_3 =>
    sig <= value_expr_3;
  . . .
  when choice_n =>
    sig <= value_expr_n;
end case;

```

Incomplete signal assignment

- According to VHDL definition:
 - Signals do not need to be assigned in all choice branch
 - When a signal is unassigned, it keeps the “previous value” (implying “memory”)

Incomplete signal assignment

- E.g.,

```

process (a)
  case a is
    when "100"|"101"|"110"|"111" =>
      high <= '1';
    when "010"|"011" =>
      middle <= '1';
    when others =>
      low <='1';
  end case;
end process;

```

- Fix #1:

```

process (a)
  case a is
    when "100"|"101"|"110"|"111" =>
      high <= '1';
      middle <= '0';
      low <= '0';
    when "010"|"011" =>
      high <= '0';
      middle <= '1';
      low <= '0';
    when others =>
      high <= '0';
      middle <= '0';
      low <= '1';
  end case;
end process;

```

- Fix #2:

```

process (a)
  high <= '0';
  middle <= '0';
  low <= '0';
  case a is
    when "100"|"101"|"110"|"111" =>
      high <= '1';
    when "010"|"011" =>
      middle <= '1';
    when others =>
      low <='1';
  end case;
end process;

```

Conceptual implementation

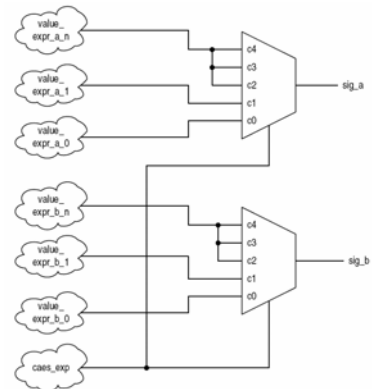
- Same as selected signal assignment statement if the case statement consists of
 - One output signal
 - One sequential signal assignment in each branch
- Multiple sequential statements can be constructed recursively

e.g.

```

case case_exp is
  when c0 =>
    sig_a <= value_expr_a_0;
    sig_b <= value_expr_b_0;
  when c1 =>
    sig_a <= value_expr_a_1;
    sig_b <= value_expr_b_1;
  when others =>
    sig_a <= value_expr_a_n;
    sig_b <= value_expr_b_n;
end case;

```



6. Simple for loop statement

- Syntax
- Examples
- Conceptual Implementation

- VHDL provides a variety of loop constructs
- Only a restricted form of loop can be synthesized
- Syntax of simple for loop:
for index **in** loop_range **loop**
 sequential statements;
end loop;
- loop_range must be static
- Index assumes value of loop_range from left to right

• E.g., bit-wide xor

```

library ieee;
use ieee.std_logic_1164.all;

entity wide_xor is
  port(
    a, b: in std_logic_vector(3 downto 0);
    y: out std_logic_vector(3 downto 0)
  );
end wide_xor;

architecture demo_arch of wide_xor is
  constant WIDTH: integer := 4;
begin
  process(a, b)
  begin
    for i in (WIDTH-1) downto 0 loop
      y(i) <= a(i) xor b(i);
    end loop;
  end process;
end demo_arch;

```

• E.g., reduced-xor

```

library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor_demo is
  port(
    a: in std_logic_vector(3 downto 0);
    y: out std_logic
  );
end reduced_xor_demo;

architecture demo_arch of reduced_xor_demo is
  constant WIDTH: integer := 4;
  signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  process(a, tmp)
  begin
    tmp(0) <= a(0); -- boundary bit
    for i in 1 to (WIDTH-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
  end process;
  y <= tmp(WIDTH-1);
end demo_arch;

```

Conceptual implementation

- “Unroll” the loop
- For loop should be treated as “shorthand” for repetitive statements
- E.g., bit-wise xor

```
y(3) <= a(3) xor b(3);  
y(2) <= a(2) xor b(2);  
y(1) <= a(1) xor b(1);  
y(0) <= a(0) xor b(0);
```

- E.g., reduced-xor

```
tmp(0) <= a(0);  
tmp(1) <= a(1) xor tmp(0);  
tmp(2) <= a(2) xor tmp(1);  
tmp(3) <= a(3) xor tmp(2);  
y <= tmp(3);
```

Synthesis of sequential statements

- Concurrent statements
 - Modeled after hardware
 - Have clear, direct mapping to physical structures
- Sequential statements
 - Intended to describe “behavior”
 - Flexible and versatile
 - Can be difficult to be realized in hardware
 - Can be easily abused

- Think hardware
- Designing hardware is not converting a C program to a VHDL program