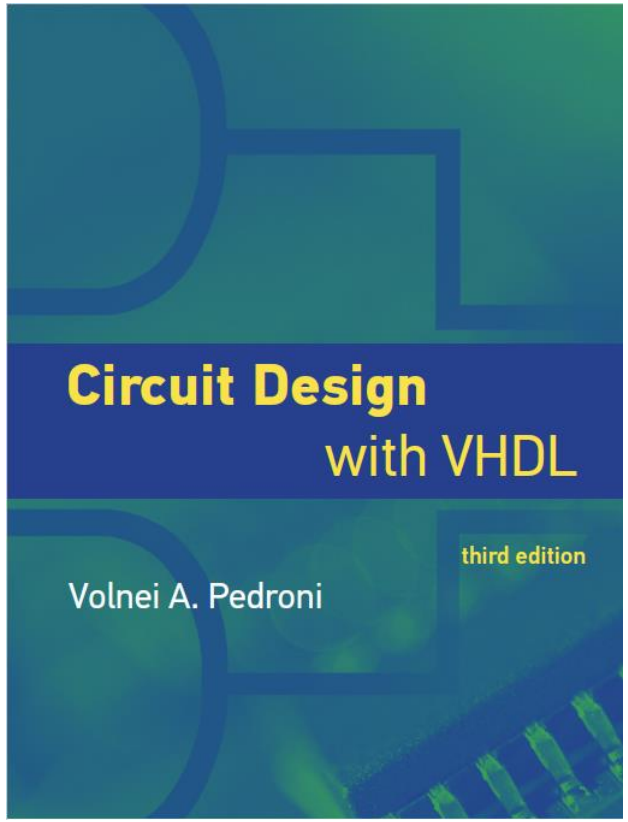


Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 10 (and 11)

Concurrent Code

Revision 1

Book Contents

Part I: Digital Circuits Review

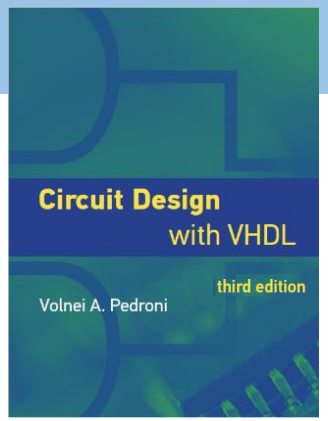
1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

Part II: VHDL

5. Introduction to VHDL
6. Code Structure and Composition
7. Predefined Data Types
8. User-Defined Data Types
9. Operators and Attributes
10. Concurrent Code
11. Concurrent Code – Practice
12. Sequential Code
13. Sequential Code – Practice
14. Packages and Subprograms
15. The Case of State Machines
16. The Case of State Machines – Practice
17. Additional Design Examples
18. Intr. to Simulation with Testbenches

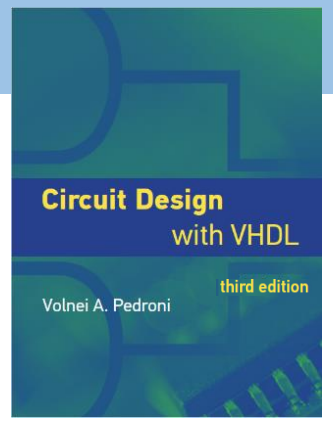
Appendices

A. Vivado Tutorial
B. Quartus Prime Tutorial
C. ModelSim Tutorial
D. Simulation Analysis and Recommendations
E. Using Seven-Segment Displays with VHDL
F. Serial Peripheral Interface
G. I2C (Inter Integrated Circuits) Interface
H. Alphanumeric LCD
I. VGA Video Interface
J. DVI Video Interface
K. TMDS Link
L. Using Phase-Locked Loops with VHDL
M. List of Enumerated Examples and Exercises



VHDL for Synthesis Slides

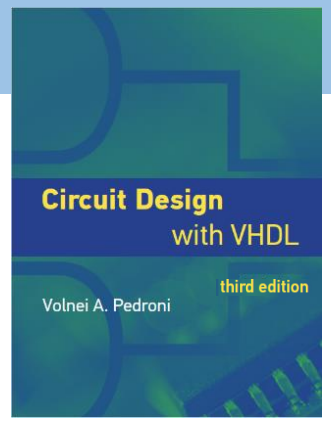
Chapter	Title
5	Introduction to VHDL
6	Code Structure and Composition
7	Predefined Data Types
8	User-Defined Data Types
9	Operators and Attributes
10	Concurrent Code
11	Concurrent Code – Practice
12	Sequential Code
13	Sequential Code – Practice
14	Packages and Subprograms



Chapters 10-11

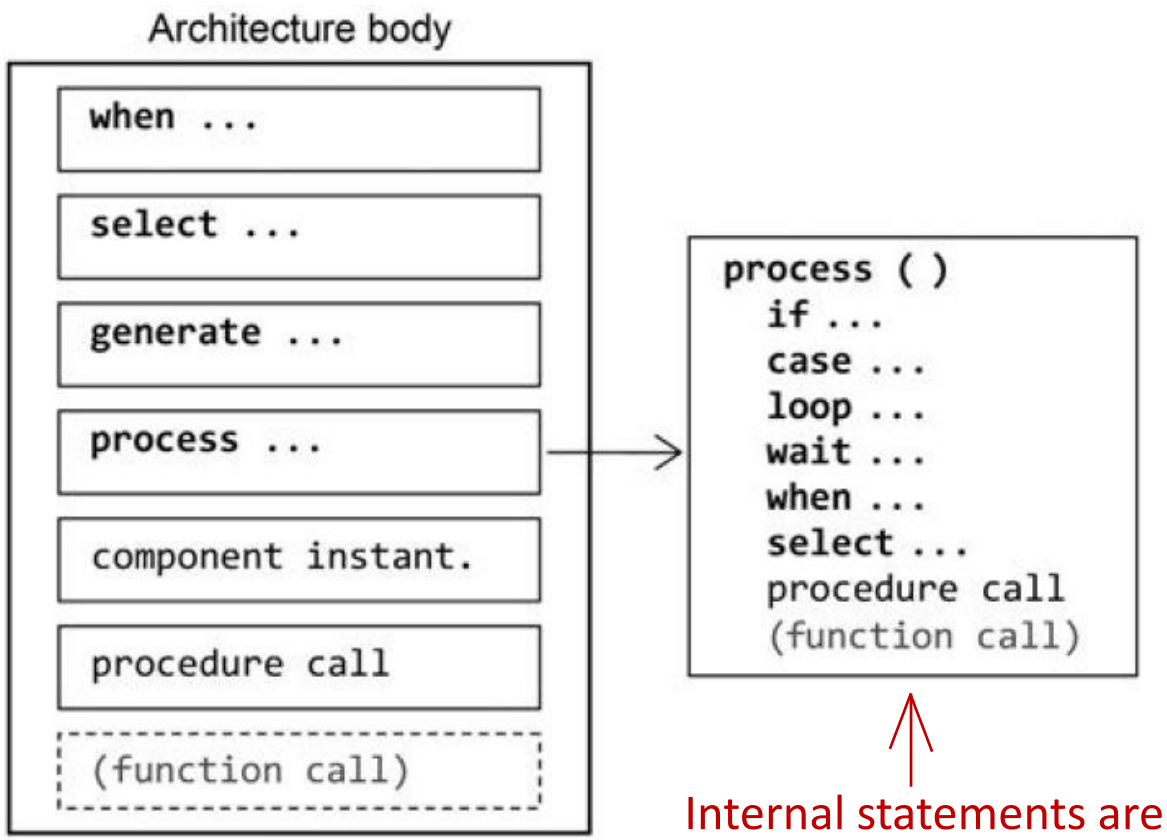
Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL



1. Concurrent statements

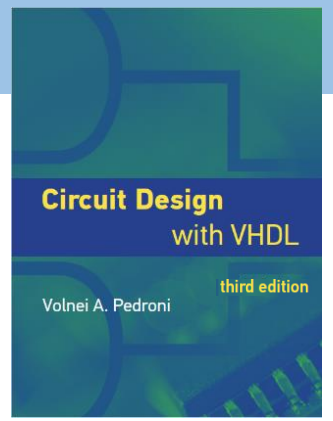
1. Concurrent statements



Internal statements are sequential
(statatem. that can go in a process;
here, the order matters)

Concurrent statements
(stat. that can go in an architecture;
the order doesn't matter)

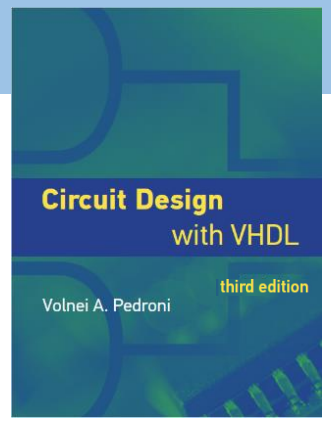
- Some statements are concurrent *and* sequential
- Statements left out:
  block → because of its little use
  assert → doesn't generate hardware
  (will be seen in ch. 14)
- *Function call* included for clarity (it's not a statement but rather part of an expression)



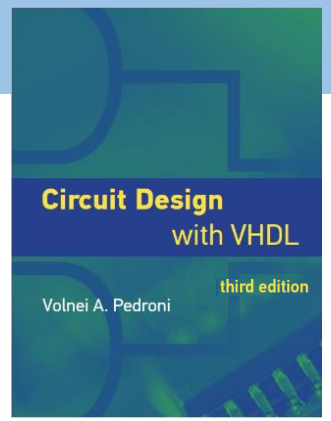
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL



2. Concurrent code



2. Concurrent code

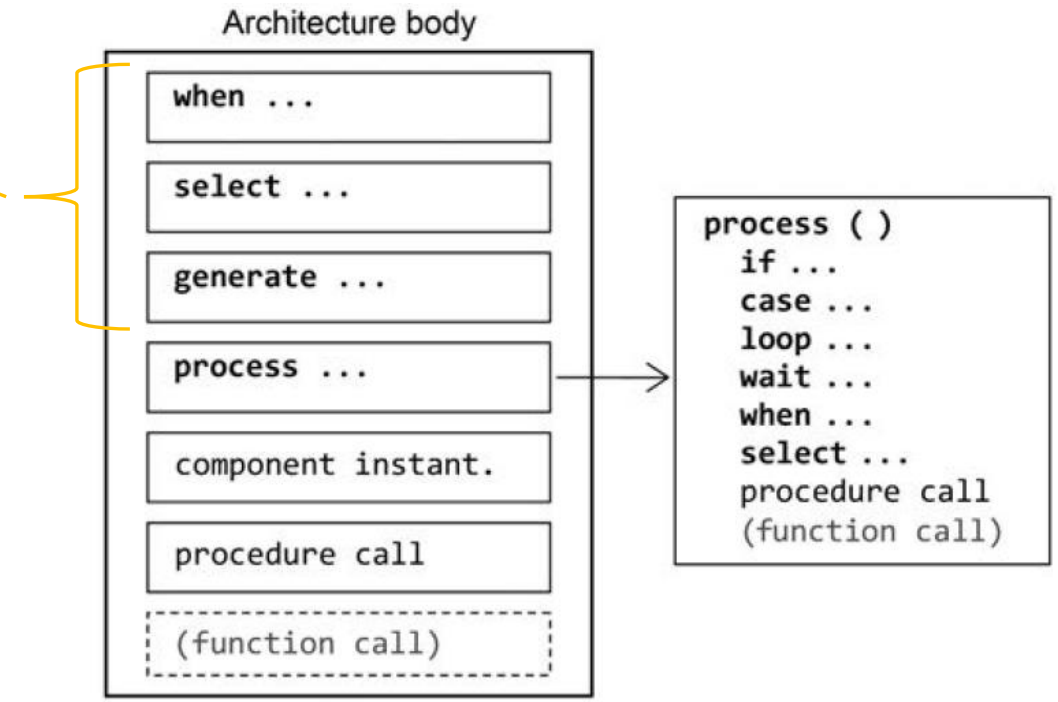
- Appropriate for implementing only combinational circuits

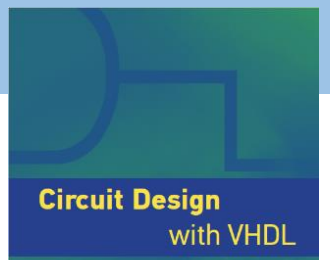
Circuit Design with VHDL

2. Concurrent code

- Appropriate for implementing only combinational circuits
- Fundamental concurrent statements (used outside processes and subprograms):

  - *when*
  - *select*
  - *generate*





2. Concurrent code

- Appropriate for implementing only combinational circuits
- Fundamental concurrent statements (used outside processes and subprograms):

  - *when*
  - *select*
  - *generate*

- Fundamental sequential statements (used inside processes and subprograms):

  (sequential versions of *when* and *select* were introduced in VHDL-2008)

  - *if*
  - *case*
  - *loop*
  - *wait*
  - *when*
  - *select*

```
process ( )
  if ...
  case ...
  loop ...
  wait ...
  when ...
  select ...
  procedure call
  (function call)
```

Circuit Design with VHDL

2. Concurrent code

- Appropriate for implementing only combinational circuits
- Fundamental concurrent statements (used outside processes and subprograms):

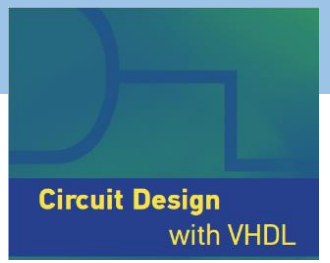
  - *when*
  - *select*
  - *generate*

- Fundamental sequential statements (used inside processes and subprograms):

  (sequential versions of *when* and *select* were introduced in VHDL-2008)

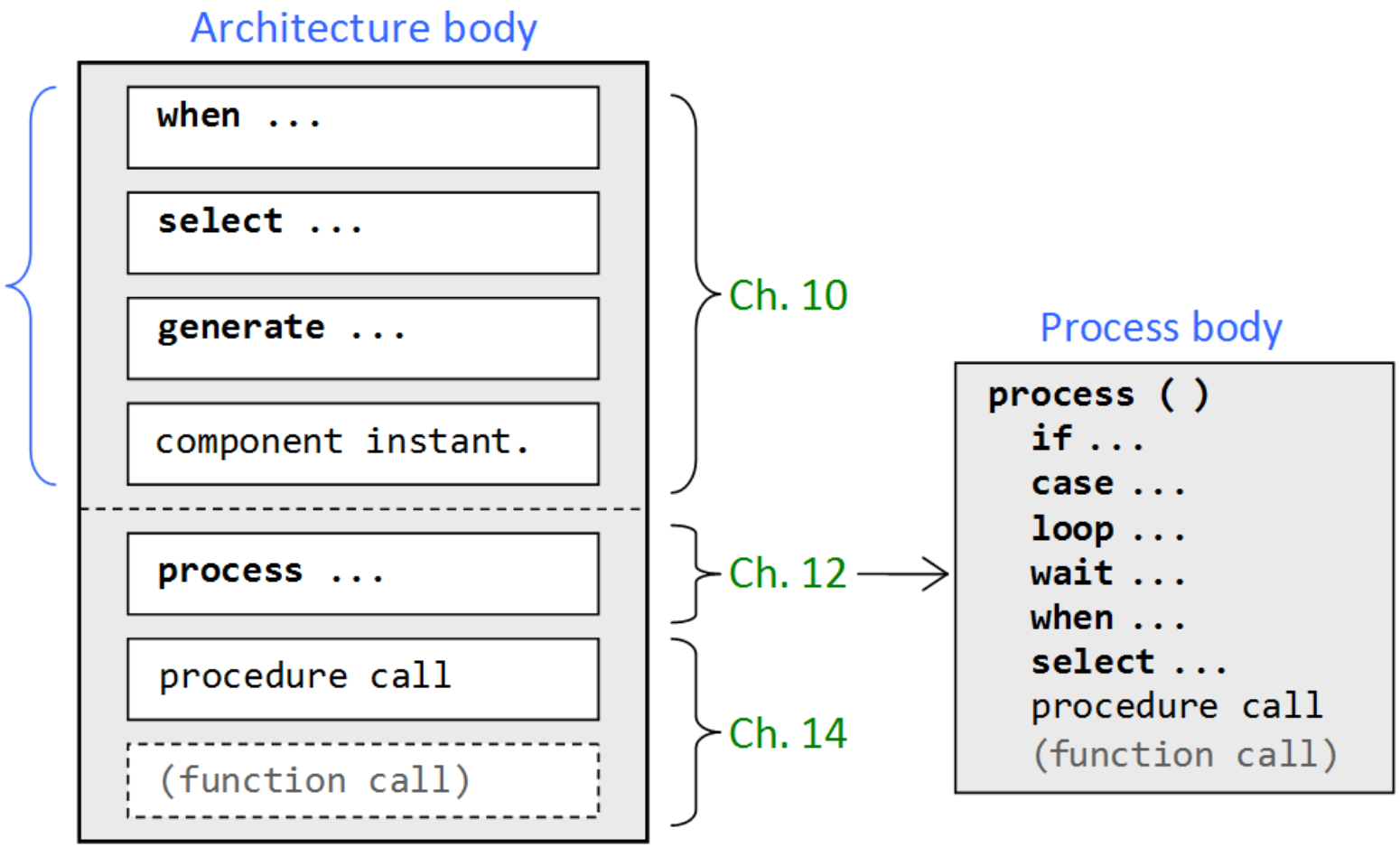
  - *if*
  - *case*
  - *loop*
  - *wait*
  - *when*
  - *select*

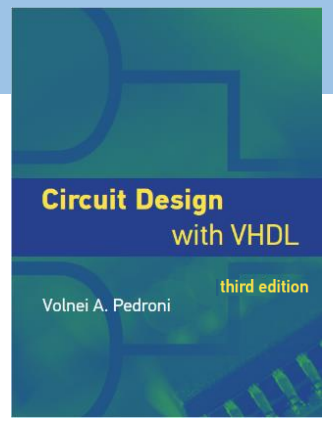
- Operators can be used anywhere (inside and outside processes and subprograms)



2. Concurrent code

Next:
- The *when* statement
- The *select* statement
- The *generate* statement
- Component instantiation

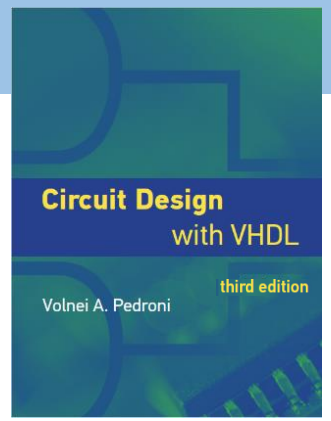




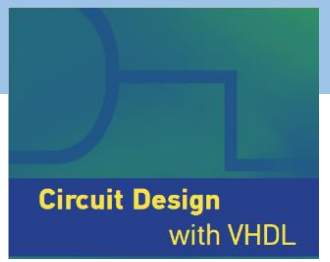
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL



3. The *when* statement



3. The *when* statement

- Also referred to as *conditional* statement
- Due to its priority-encoding nature, use it only when there are few conditions to test
- To enter truth tables in general, use *select* instead
- IMP.: Incomplete in-out coverage is accepted by the compiler, but it will infer latches

Circuit Design with VHDL

3. The *when* statement

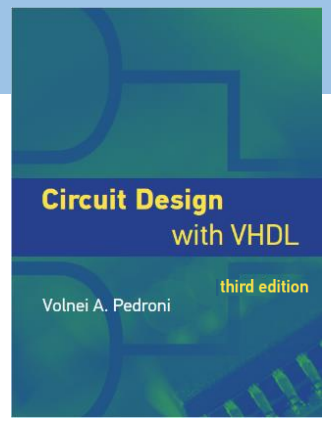
- Also referred to as *conditional* statement
- Due to its priority-encoding nature, use it only when there are few conditions to test
- To enter truth tables in general, use *select* instead
- IMP.: Incomplete in-out coverage is accepted by the compiler, but it will infer latches

Syntax:

```
target <= value when condition else
          value when condition else
          value;
```

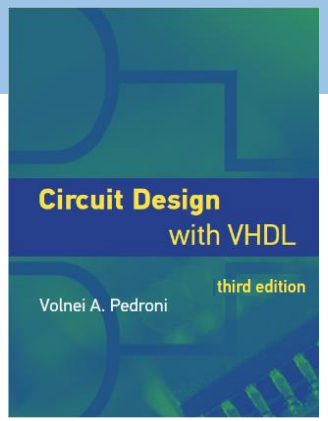
```
target <= value when condition else
          value when condition else
          value when condition;
```

- Ending with else value → Easy to cover all remaining values
- Ending with when condition → Clearer (shows all options explicitly, but rarely viable)



3. The *when* statement

Example:

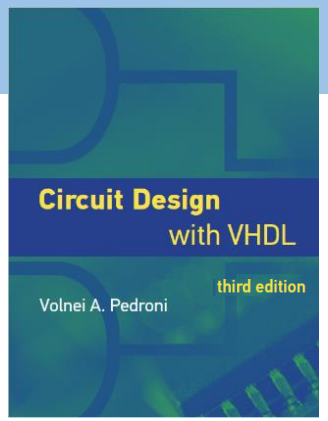


3. The *when* statement

Example:

```
outp <= inp when ena else (others => 'Z');  --tri-state buffer
```

- There is just one condition (signal *ena*) to test, so *when* is fine
- It ends in “else value”, fine too because if *ena* is of type SU or SL it is difficult to cover all possibilities



3. The *when* statement

Example:

```
outp <= inp when ena else (others => 'Z');  --tri-state buffer
```

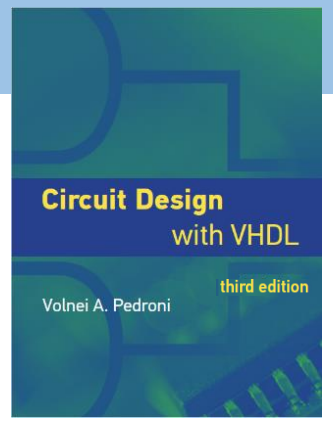
- There is just one condition (signal *ena*) to test, so *when* is fine
- It ends in "else value", fine too because if *ena* is of type SU or SL it is difficult to cover all possibilities

Example:

```
--Ending in else value:
y <= a when sel=0 else
     b when sel=1 else
     c when sel=2 else
     d;
```

```
--Ending in when condition:
y <= a when sel=0 else
     b when sel=1 else
     c when sel=2 else
     d when sel=3;
```

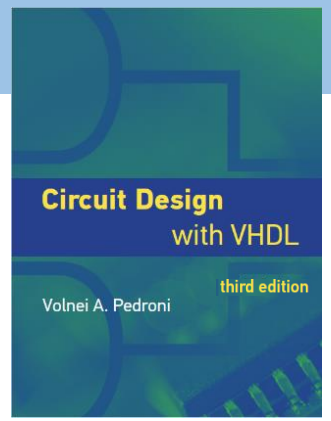
- A (rare) example where all conditions can be shown explicitly (*sel* is an integer, which is unusual)
- But even this small multiplexer is better tailored for *select* than for *when*



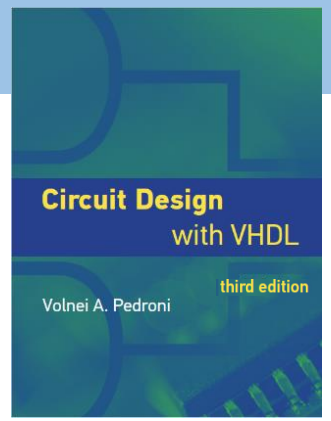
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL

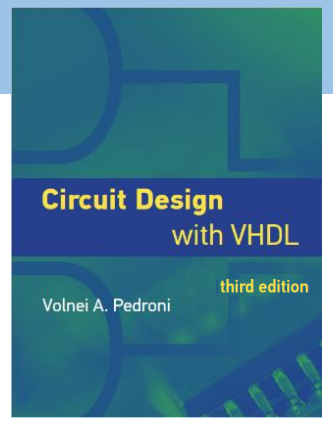


4. The *select* statement



4. The *select* statement

- Also referred to as *selected* statement
- Proper option for entering truth tables in general (in concurrent code)
- Incomplete in-out coverage not accepted by compiler (latch-free circuit)



4. The *select* statement

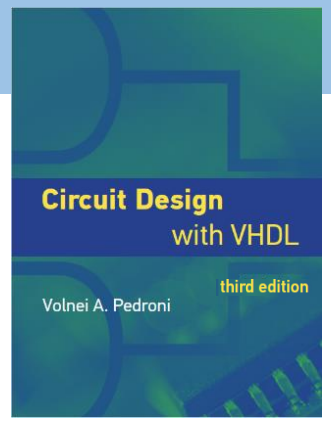
- Also referred to as *selected* statement
- Proper option for entering truth tables in general (in concurrent code)
- Incomplete in-out coverage not accepted by compiler (latch-free circuit)

Syntax:

```
with expression select
   target <= value when choice,
             value when choice,
             value when others;
```

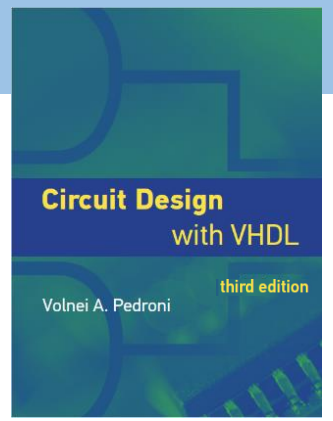
```
with expression select
   target <= value when choice,
             value when choice,
             value when choice;
```

- Ending with value when others → Easy to cover all remaining values
- Ending with value when choice → Clearer (shows all options explicitly, but rarely viable)



4. The *select* statement

Example:



4. The *select* statement

Example:

```
--Ending in value when others:
with sel select
   y <= a when 0,
        b when 1,
        c when 2,
        d when others;
```

```
--Ending in value when choice:
with sel select
   y <= a when 0,
        b when 1,
        c when 2,
        d when 3;
```

- This is the multiplexer of the previous example, where *sel* is an integer (unusual choice)
- Implemented with *select*, which is the right option for straightforward truth tables
- If *sel* were SUV or SLV (typical choice), the approach on the right would not be viable

4. The *select* statement

- Choice expressions can use *to*, *downto*, | (means *or*), and *others*

Circuit Design with VHDL

4. The *select* statement

- Choice expressions can use *to*, *downto*, | (means *or*), and *others*

Example:

```
with sel select
  x <= a when 0 | 7,
       b when 2 to 5,
       c when others;
```

Circuit Design with VHDL

4. The *select* statement

- Equations are allowed in the selection argument:

Circuit Design with VHDL

4. The *select* statement

- Equations are allowed in the selection argument:

Example:

```
with (a > b) select    --parentheses are optional
  y <= a when true,
       b when false;
```

Circuit Design with VHDL

4. The *select* statement

- Equations are allowed in the selection argument:

Example:

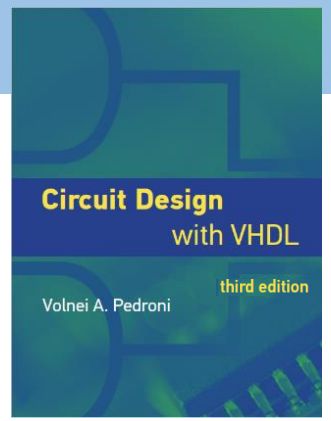
```
with (a > b) select    --parentheses are optional
  y <= a when true,
       b when false;
```

Example:

```
signal a, b: std_logic_vector(BITSab-1 downto 0);
signal x, y: std_logic_vector(BITSxy-1 downto 0);
...
with (a xor b) select
y <= x when (BITSxy-1 downto 0 => '1'),
     not x when others;
```

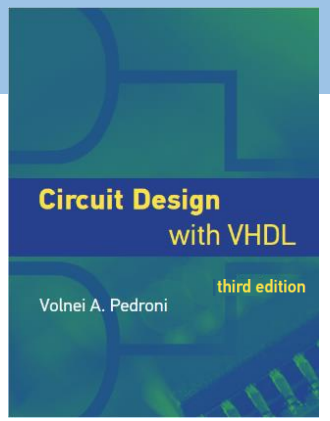
Equivalent:

```
x when (x'range => '1')
x when (y'range => '1')
```



4. The *select* statement

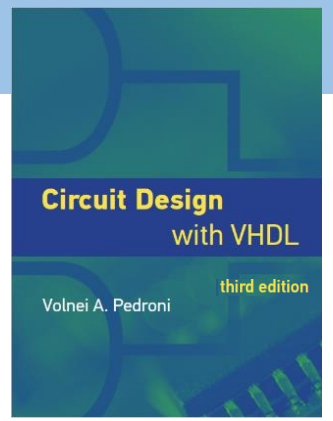
The matching *select?* version



4. The *select* statement

The matching ***select?*** version

- Uses the matching comparison operator ('0'='L', '1'='H', and '–' = any value)
- So it can be used for "don't care" values at the input

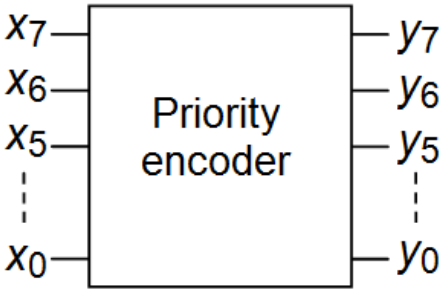


4. The *select* statement

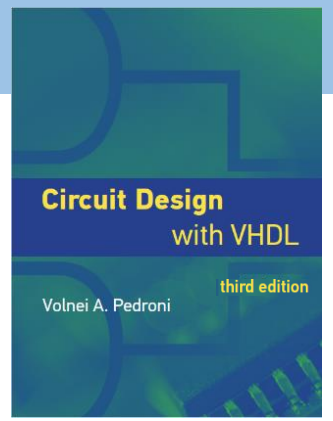
The matching ***select?*** version

- Uses the matching comparison operator ('0'='L', '1'='H', and '–' = any value)
- So it can be used for "don't care" values at the input

Example:



$x_7 \ldots x_0$	$y_7 \ldots y_0$
1xxxxxxx	10000000
01xxxxxx	01000000
001xxxxx	00100000
0001xxxx	00010000
00001xxx	00001000
000001xx	00000100
0000001x	00000010
00000001	00000001
00000000	00000000

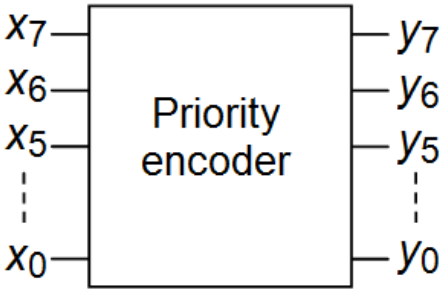


4. The *select* statement

The matching ***select?*** version

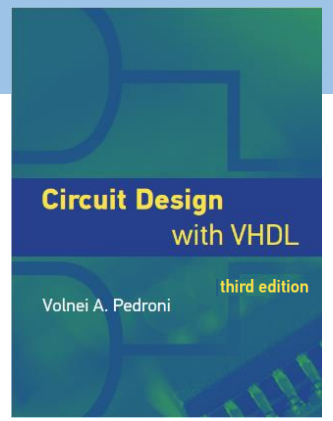
- Uses the matching comparison operator ('0'='L', '1'='H', and '–' = any value)
- So it can be used for "don't care" values at the input

Example:



$x_7 \ldots x_0$	$y_7 \ldots y_0$
1xxxxxxx	10000000
01xxxxxx	01000000
001xxxxx	00100000
0001xxxx	00010000
00001xxx	00001000
000001xx	00000100
0000001x	00000010
00000001	00000001
00000000	00000000

```
architecture lut of priority_encoder is
begin
  with x select?
    y <= "1000" when "1---",
         "0100" when "01--",
         "0010" when "001-",
         "0001" when "0001",
         "0000" when others;
end architecture;
```



Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL

5. The *generate* statement

5. The *generate* statement

- *generate* is used to build loops in concurrent code

5. The *generate* statement

- *generate* is used to build loops in concurrent code
- It has three forms: *for-generate*, *if-generate*, *case-generate*

Circuit Design with VHDL

5. The *generate* statement

- *generate* is used to build loops in concurrent code
- It has three forms: *for-generate*, *if-generate*, *case-generate*
- *for-generate* is by far the most common
- So only that will be seen here (check the others in section 10.4)

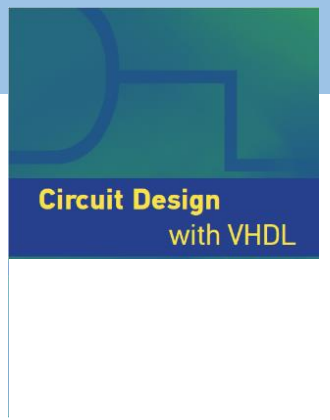
Circuit Design with VHDL

5. The *generate* statement

- *generate* is used to build loops in concurrent code
- It has three forms: *for-generate*, *if-generate*, *case-generate*
- *for-generate* is by far the most common
- So only that will be seen here (check the others in section 10.4)

for-generate:

```
label: for identifier in generate_range generate
   [generate_declarative_part
begin]
  concurrent_statements
end generate [label];
```



5. The *generate* statement

Example:

Circuit Design with VHDL

5. The *generate* statement

Example:

```
signal a, b, x: std_ulogic_vector(7 downto 0);
...
gen: for i in x'range generate
   x(i) <= a(i) xor b(b'left-i);
end generate;
```

What's the resulting circuit?

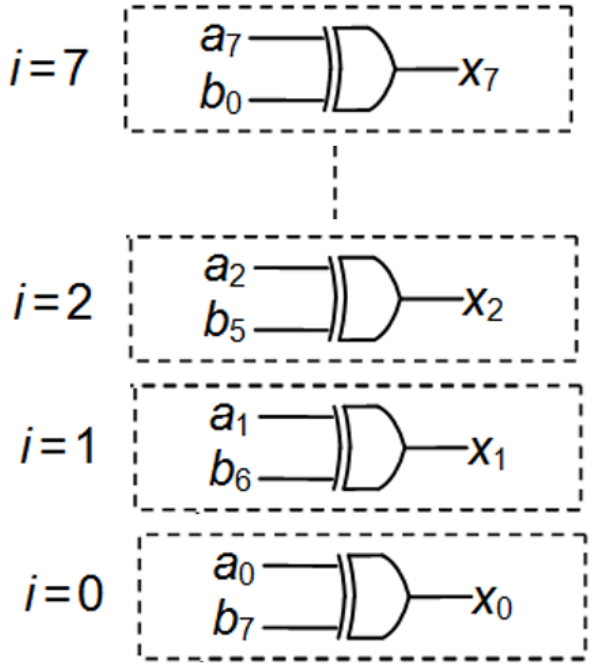
Circuit Design with VHDL

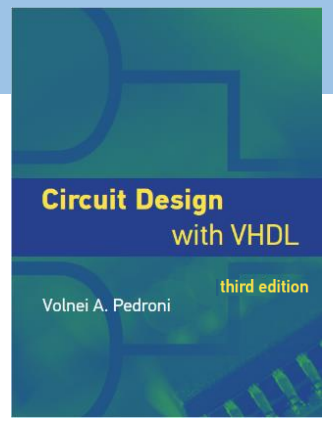
5. The *generate* statement

Example:

```
signal a, b, x: std_ulogic_vector(7 downto 0);
...
gen: for i in x'range generate
   x(i) <= a(i) xor b(b'left-i);
end generate;
```

What's the resulting circuit?

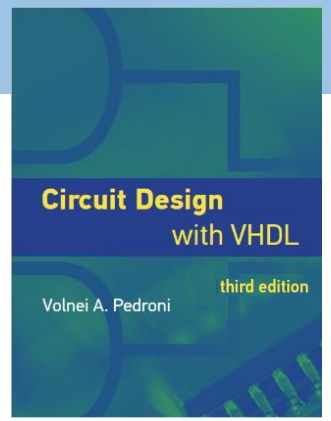




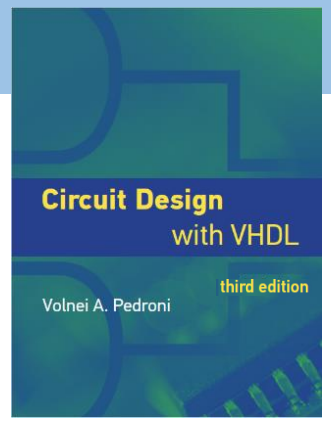
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL

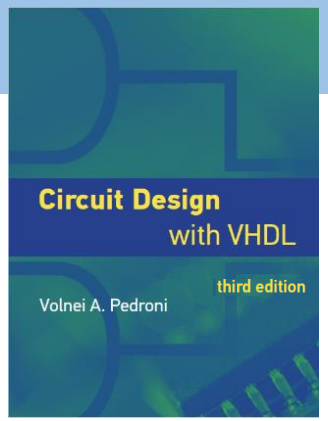


6. Component instantiation statements



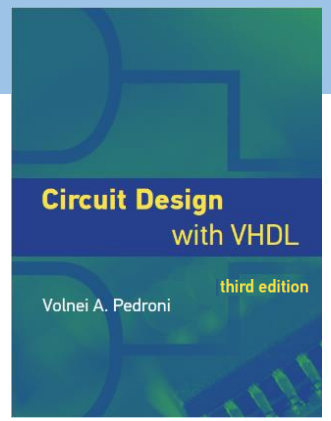
6. Component instantiation statements

- The circuit to be instantiated is a regular (previous) VHDL design



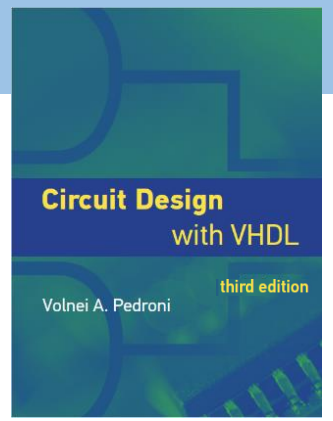
6. Component instantiation statements

- The circuit to be instantiated is a regular (previous) VHDL design
- It can be combinational or sequential



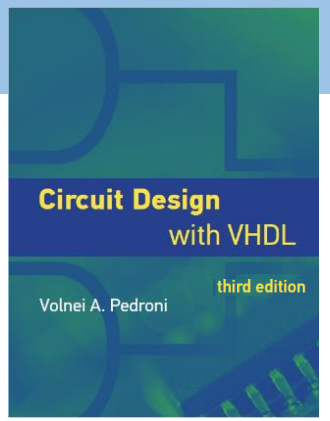
6. Component instantiation statements

- The circuit to be instantiated is a regular (previous) VHDL design
- It can be combinational or sequential
- But the instantiation can only be done in concurrent code



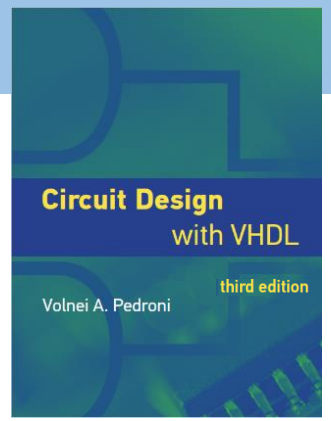
6. Component instantiation statements

- The circuit to be instantiated is a regular (previous) VHDL design
- It can be combinational or sequential
- But the instantiation can only be done in concurrent code
- The original design is associated to the new design using:
  port map for the circuit ports
  generic map for generic constants (if there are any)



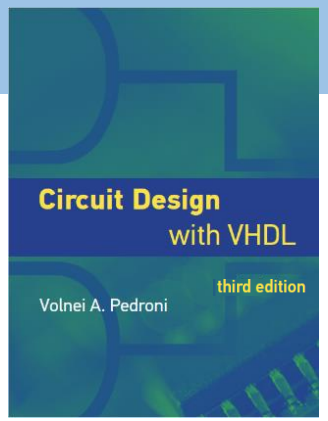
6. Component instantiation statements

- The circuit to be instantiated is a regular (previous) VHDL design
- It can be combinational or sequential
- But the instantiation can only be done in concurrent code
- The original design is associated to the new design using:
  port map for the circuit ports
  generic map for generic constants (if there are any)
- The instantiation can be done in two ways:
  a) Component instantiation (requires declaration + instantiation)
  b) Design entity instantiation (direct instantiation)



6. Component instantiation statements

a) Component instantiation



6. Component instantiation statements

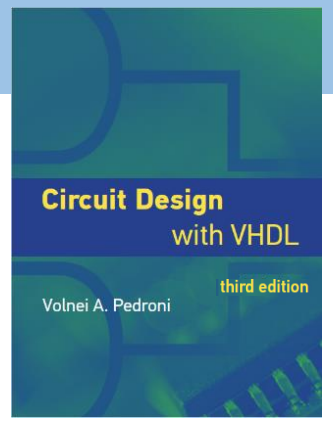
a) Component instantiation

Component declaration:

```
component component_name [is]
  [generic (...);]
  port (...);
end component [component_name];
```

Component instantiation:

```
label: [component] component_name
  [generic map (generic_association_list)]
  port map (port_association_list);
```



6. Component instantiation statements

a) Component instantiation

Component declaration:

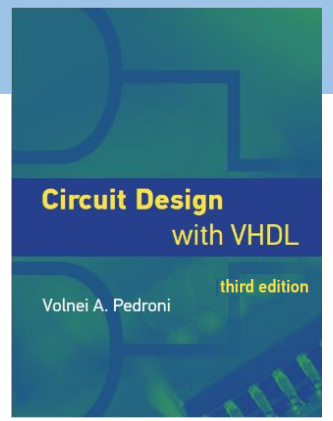
```
component component_name [is]
  [generic (...);]
  port (...);
end component [component_name];
```

Component instantiation:

```
label: [component] component_name
  [generic map (generic_association_list)]
  port map (port_association_list);
```

Declaration:

- It is a copy of the entity declaration, just with the word *entity* replaced with *component*
- A common place for this declaration is the architecture's declarative region



6. Component instantiation statements

a) Component instantiation

Component declaration:

```
component component_name [is]
  [generic (...);]
  port (...);
end component [component_name];
```

Component instantiation:

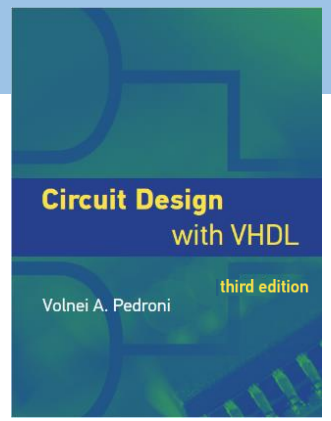
```
label: [component] component_name
  [generic map (generic_association_list)]
  port map (port_association_list);
```

Declaration:

- It is a copy of the entity declaration, just with the word *entity* replaced with *component*
- A common place for this declaration is the architecture's declarative region

Instantiation:

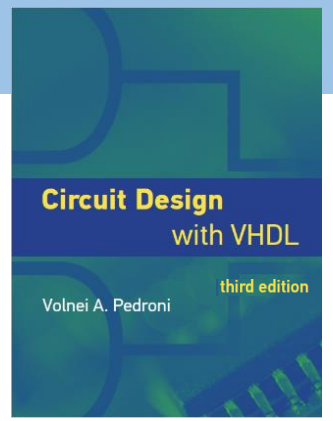
- Employs *port map* plus, if necessary, *generic map*
- The associations can be named or positional
- The keyword *open* is used to leave a port unconnected



6. Component instantiation statements

a) Component instantiation

Example:



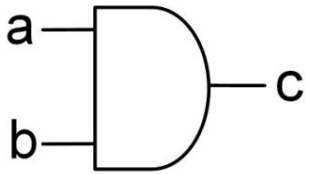
6. Component instantiation statements

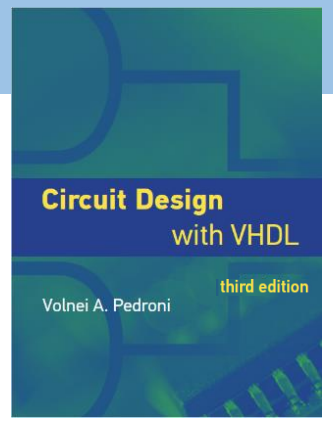
a) Component instantiation

Example:

```
--Component declaration:
component and_gate is
  port (
    a, b: in bit;
    c: out bit);
end component;
```

Original design:





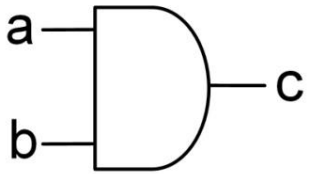
6. Component instantiation statements

a) Component instantiation

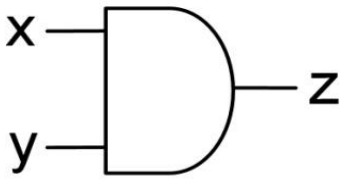
Example:

```
--Component declaration:
component and_gate is
  port (
    a, b: in bit;
    c: out bit);
end component;

--Component instantiation with named association:
and2x1: and_gate port map (a => x, b => y, c => z);

--Component instantiation with positional association:
and2x1: and_gate port map (x, y, z);
```

Original design:



As part of the new design:



6. Component instantiation statements

a) Component instantiation

A more detailed example:

6. Component instantiation statements

a) Component instantiation

A more detailed example:

A previous project:

```
entity brick is
  generic (
    NUM_BITS: positive);
  port (
    a, b: in ...;
    c: out ...);
end entity;

architecture ...
  ...
end architecture;
```

declaration

instantiation (named)

Current project:

```
entity wall is
  generic (
    WIDTH: positive := 32);
  port (
    x, y: in ...;
    z: out ...);
end entity wall;

architecture ...
  component brick is
    ...
  end component;
begin
  comp: brick generic map (NUM_BITS => WIDTH)
        port map (a => x, b => y, c => z);
   ...
end architecture;
```

6. Component instantiation statements

a) Component instantiation

A more detailed example:

A previous project:

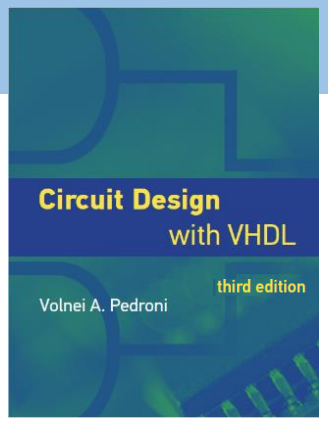
```
entity brick is
  generic (
    NUM_BITS: positive);
  port (
    a, b: in ...;
    c: out ...);
end entity;

architecture ...
  ...
end architecture;
```

declaration

instantiation
(positional)

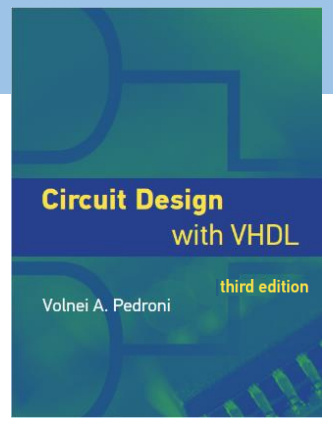
Current project:

```
entity wall is
  generic (
    WIDTH: positive := 32);
  port (
    x, y: in ...;
    z: out ...);
end entity wall;

architecture ...
  component brick is
    ...
  end component;
begin
  comp: brick generic map (WIDTH)
        port map (x, y, z);
   ...
end architecture;
```



6. Component instantiation statements

b) Design entity instantiation



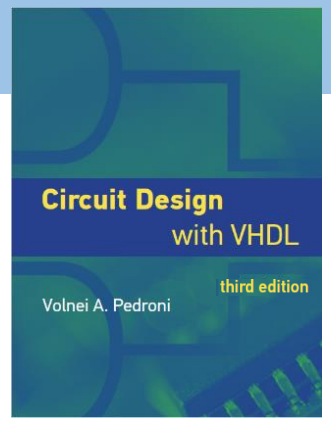
6. Component instantiation statements

b) Design entity instantiation

Instantiation:

```
label: entity work.entity_name [(architecture_name)]
  [generic map (generic_association_list)]
  port map (port_association_list);
```

Advantage: Declaration not needed



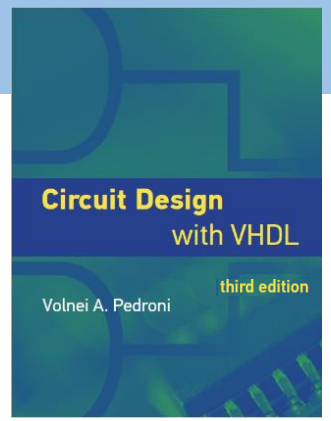
6. Component instantiation statements

b) Design entity instantiation

Example:

```
entity wall is
  generic (
    WIDTH: positive := 32);
  port (
    x, y: in ...;
    z: out ...);
end entity wall;

architecture ...
begin
  comp: entity work.brick generic map (NUM_BITS => WIDTH)
        port map (a => x, b => y, c => z);
   ...
end architecture;
```

instantiation
(named)



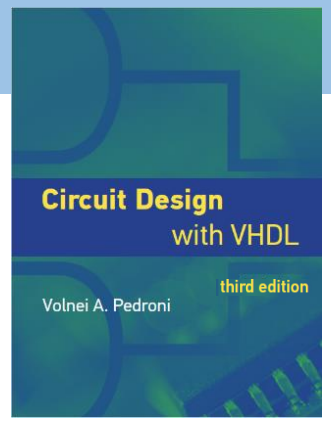
6. Component instantiation statements

b) Design entity instantiation

Example:

```
entity wall is
  generic (
    WIDTH: positive := 32);
  port (
    x, y: in ...;
    z: out ...);
end entity wall;

architecture ...
begin
  comp: entity work.brick generic map (NUM_BITS)
        port map (x, y, z);
   ...
end architecture;
```

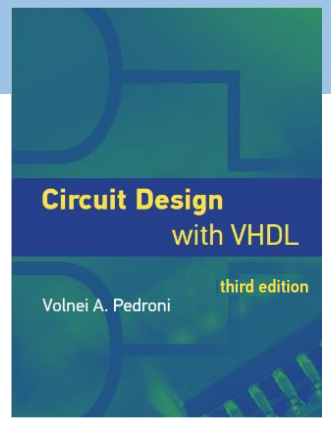
instantiation
(positional)



We close this chapter with two very special cases:

1) How to avoid assigning a value to a signal more than once

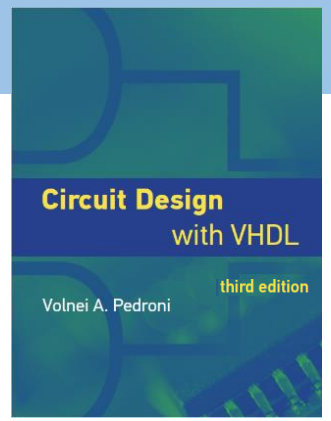
2) How to do math properly with VHDL



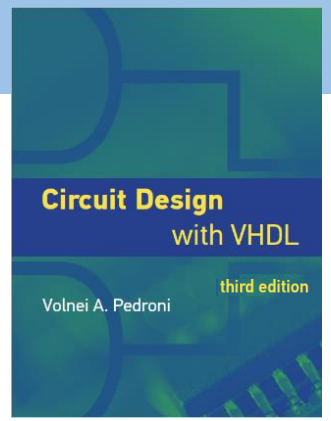
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL



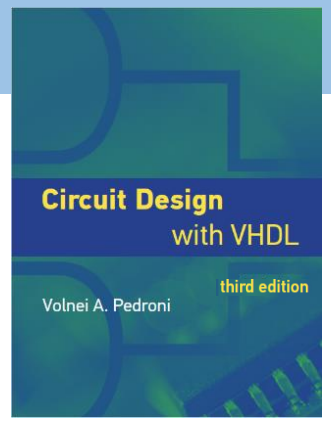
7. Avoiding multiple assignments to the same signal



7. Avoiding multiple assignments to the same signal

- Why can't we do this (outside a process)?

```
y <= 0;
...
y <= x + 1;
```

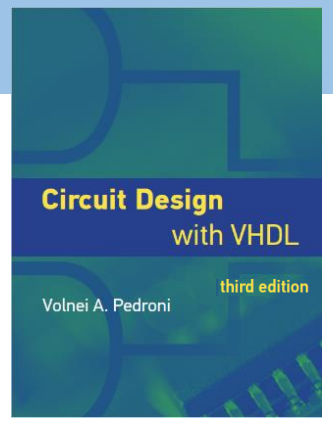


7. Avoiding multiple assignments to the same signal

- Why can't we do this (outside a process)?

```
y <= 0;
...
y <= x + 1;
```

- Because it is a concurrent code, so any order of the statements should produce the same result



7. Avoiding multiple assignments to the same signal

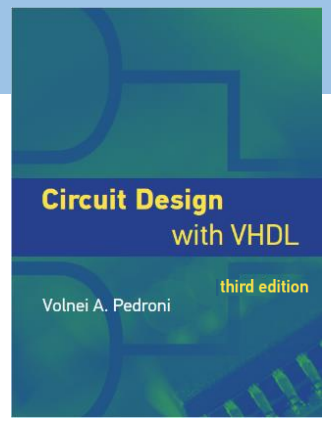
- Why can't we do this (outside a process)?

```
y <= 0;
...
y <= x + 1;
```

- Because it is a concurrent code, so any order of the statements should produce the same result

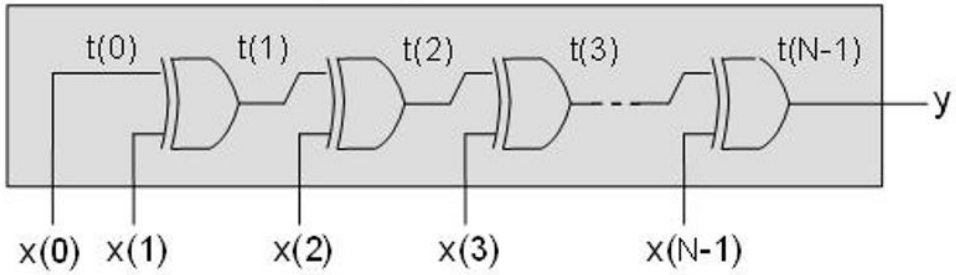
- Proposed solution:

1) Create an internal signal with extra dimension:
   Scalar → 1D
   1D → 1D×1D or 2D
   1D×1D → 1D×1D×1D or 3D, etc.
2) Compute the values for this signal
3) Pass its last value to the output signal

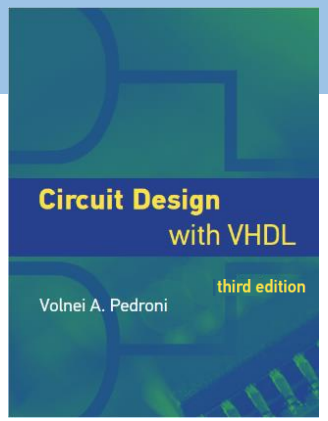


7. Avoiding multiple assignments to the same signal

Example: Chain-type parity detector (sec. 1.4.3)



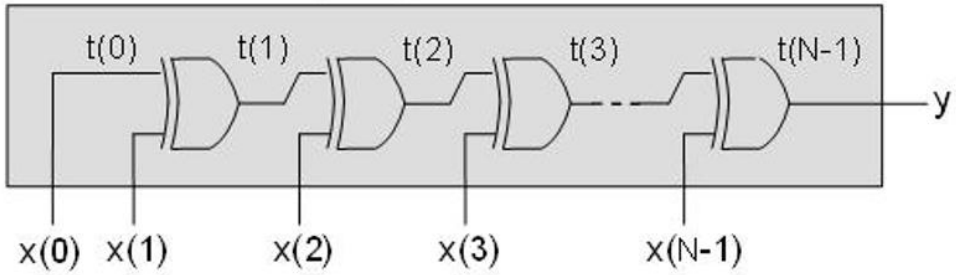
Note: Assume that there is no unary *xor* operator (`y <= xor x;`)



7. Avoiding multiple assignments to the same signal

Example:

Chain-type parity detector (sec. 1.4.3)

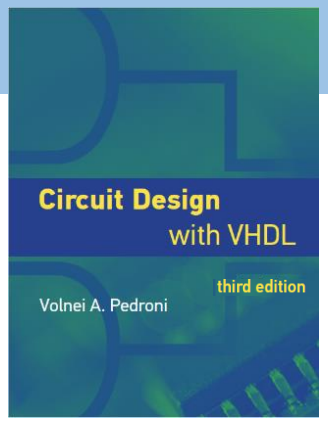


Note: Assume that there is no unary *xor* operator (`y <= xor x;`)

Solution 1:

```
-----------------------------------
architecture a1 of parity_detector is
begin
  y <= x(0);
  gen: for i in 1 to N-1 generate
    y <= y xor x(i);
  end generate;
end architecture;
-----------------------------------
```

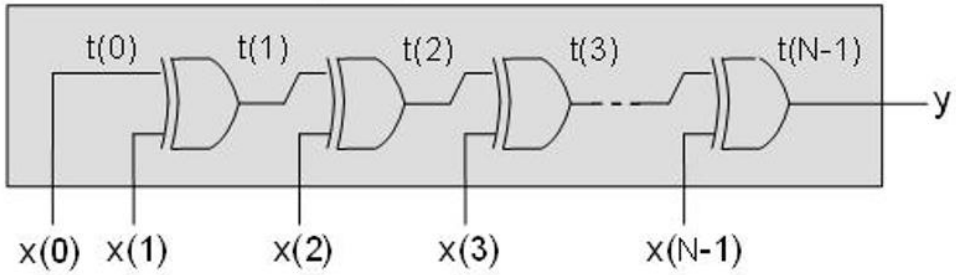
How many assignments to signal *y*? ___



7. Avoiding multiple assignments to the same signal

Example:

Chain-type parity detector (sec. 1.4.3)

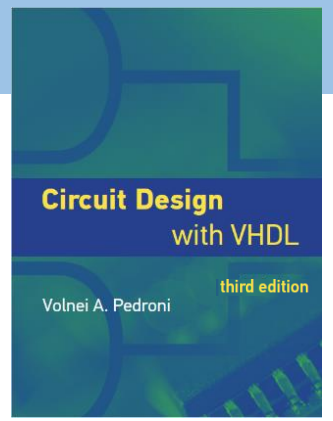


Note: Assume that there is no unary *xor* operator (`y <= xor x;`)

Solution 1:

```
-----------------------------------
architecture a1 of parity_detector is
begin
  y <= x(0);
  gen: for i in 1 to N-1 generate
    y <= y xor x(i);
  end generate;
end architecture;
-----------------------------------
```

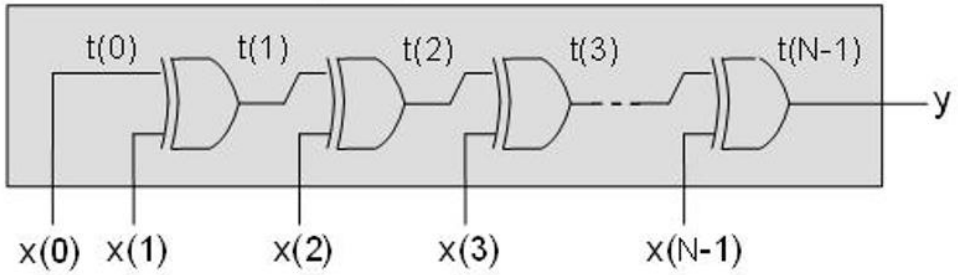
How many assignments to signal *y*? N



7. Avoiding multiple assignments to the same signal

Example:

Chain-type parity detector (sec. 1.4.3)



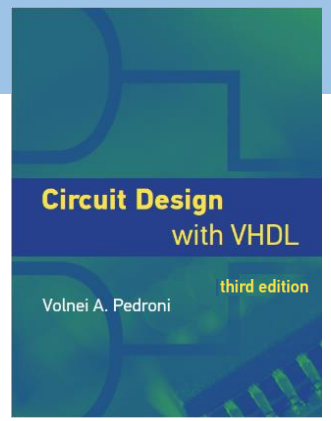
y is scalar (one bit)

Solution 2:

```
-----------------------------------
architecture a2 of parity_detector is
  signal t: bit_vector(N-1 downto 0);
begin
  t(0) <= x(0);
  gen: for i in 1 to N-1 generate
    t(i) <= t(i-1) xor x(i);
  end generate;
  y <= t(n-1);
end architecture;
-----------------------------------
```

t is 1D (vector of bits)

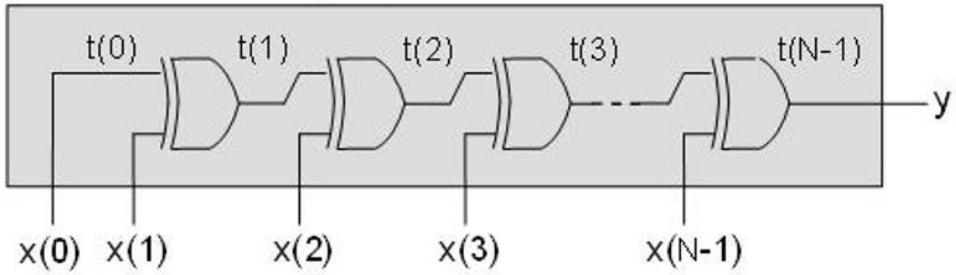
last value of *t* passed to *y* (single bit)



7. Avoiding multiple assignments to the same signal

Example:

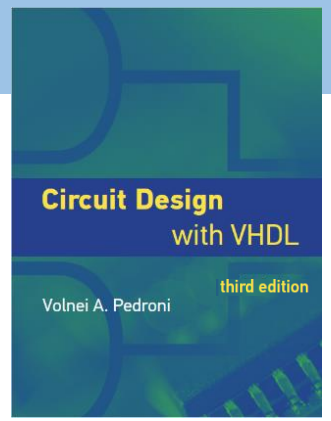
Chain-type parity detector (sec. 1.4.3)



Solution 2:

```
-----------------------------------
architecture a2 of parity_detector is
  signal t: bit_vector(N-1 downto 0);
begin
  t(0) <= x(0);
  gen: for i in 1 to N-1 generate
    t(i) <= t(i-1) xor x(i);
  end generate;
  y <= t(n-1);
end architecture;
-----------------------------------
```

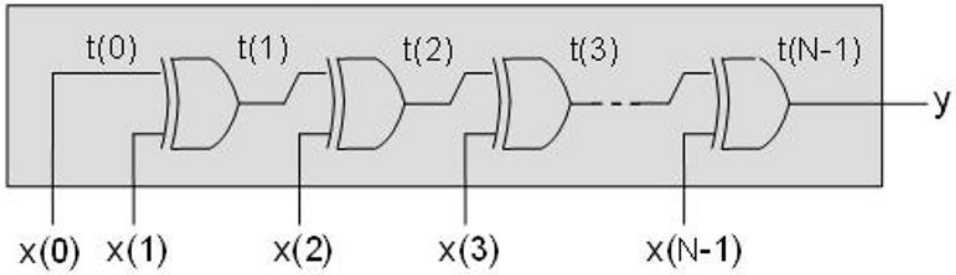
How many assignments to signal *y*? ___



7. Avoiding multiple assignments to the same signal

Example:

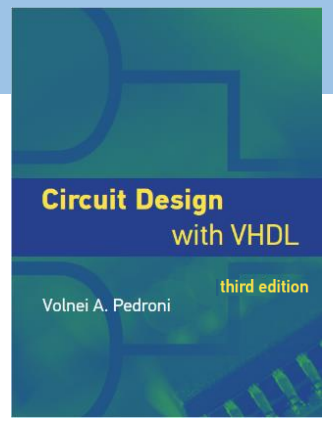
Chain-type parity detector (sec. 1.4.3)



Solution 2:

```
-----------------------------------
architecture a2 of parity_detector is
  signal t: bit_vector(N-1 downto 0);
begin
  t(0) <= x(0);
  gen: for i in 1 to N-1 generate
    t(i) <= t(i-1) xor x(i);
  end generate;
  y <= t(n-1);
end architecture;
-----------------------------------
```

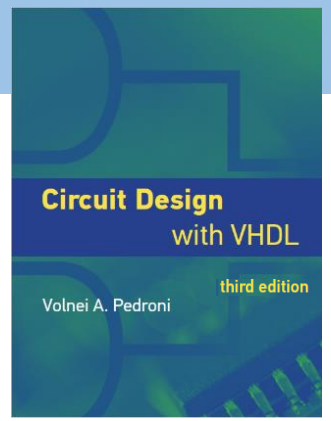
How many assignments to signal *y*? 1



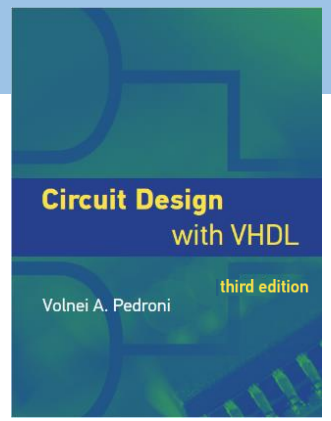
Chapters 10-11

Concurrent Code

1. Concurrent statements
2. Concurrent code
3. The *when* statement
4. The *select* statement
5. The *generate* statement
6. Component instantiation statements
7. Avoiding multiple assignments to the same signal
8. Doing math right with VHDL

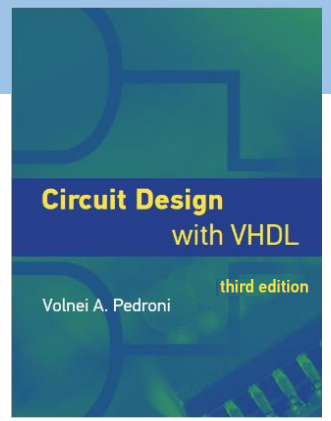


8. Doing math right with VHDL



8. Doing math right with VHDL

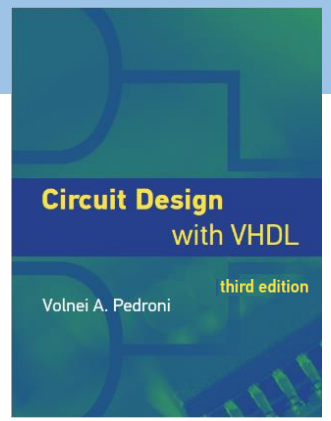
Note: Review the concepts below in section 1.6:
- Integer arithmetic
- Floating-point arithmetic
- Carry bit versus overflow flag
- Extension, truncation, rounding, and saturation



8. Doing math right with VHDL

Note: Review the concepts below in section 1.6:
- Integer arithmetic
- Floating-point arithmetic
- Carry bit versus overflow flag
- Extension, truncation, rounding, and saturation

- Consider the following as the "arithmetic" types:
  - For integers: `unsigned`, `signed`
  - For fixed-point: `ufixed`, `sfixed`
  - For floating-point: `float`, `float32`, `float64`, `float128`

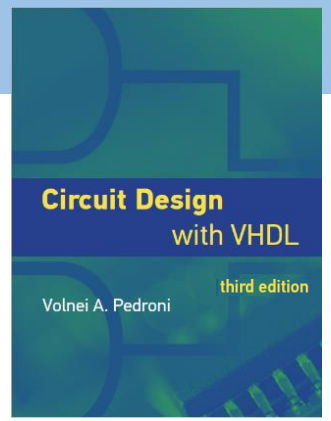


8. Doing math right with VHDL

Note: Review the concepts below in section 1.6:
- Integer arithmetic
- Floating-point arithmetic
- Carry bit versus overflow flag
- Extension, truncation, rounding, and saturation

- Consider the following as the "arithmetic" types:
  - For integers: `unsigned`, `signed`
  - For fixed-point: `ufixed`, `sfixed`
  - For floating-point: `float`, `float32`, `float64`, `float128`

- Do not use type `integer` to do math



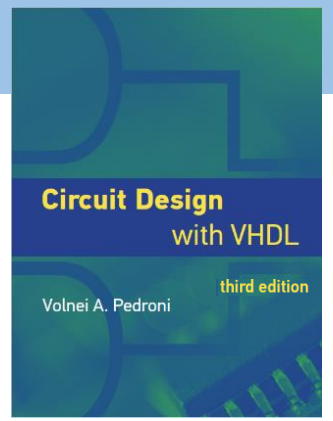
8. Doing math right with VHDL

Note: Review the concepts below in section 1.6:
- Integer arithmetic
- Floating-point arithmetic
- Carry bit versus overflow flag
- Extension, truncation, rounding, and saturation

- Consider the following as the "arithmetic" types:
  - For integers: `unsigned`, `signed`
  - For fixed-point: `ufixed`, `sfixed`
  - For floating-point: `float`, `float32`, `float64`, `float128`

- Do not use type `integer` to do math

- Avoid floating-point whenever possible (a lot of hardware, lower speed, extra power)



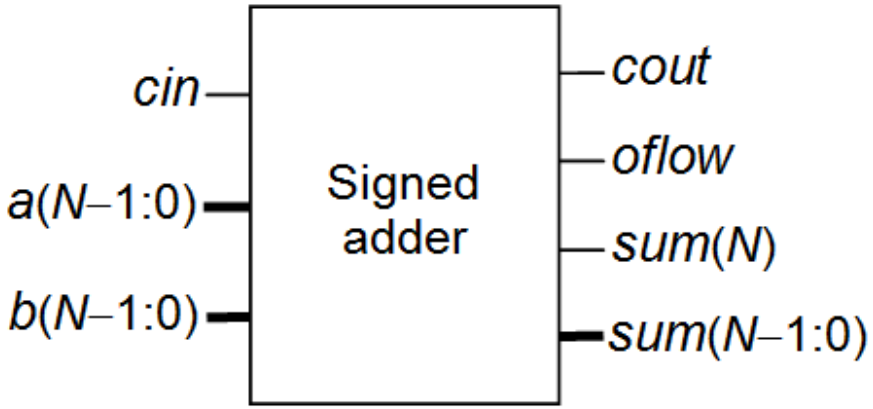
8. Doing math right with VHDL

Note: Review the concepts below in section 1.6:
- Integer arithmetic
- Floating-point arithmetic
- Carry bit versus overflow flag
- Extension, truncation, rounding, and saturation

- Consider the following as the "arithmetic" types:
  - For integers: `unsigned`, `signed`
  - For fixed-point: `ufixed`, `sfixed`
  - For floating-point: `float`, `float32`, `float64`, `float128`
- Do not use type `integer` to do math
- Avoid floating-point whenever possible (a lot of hardware, lower speed, extra power)
- As default, use standard-logic types for the circuit ports

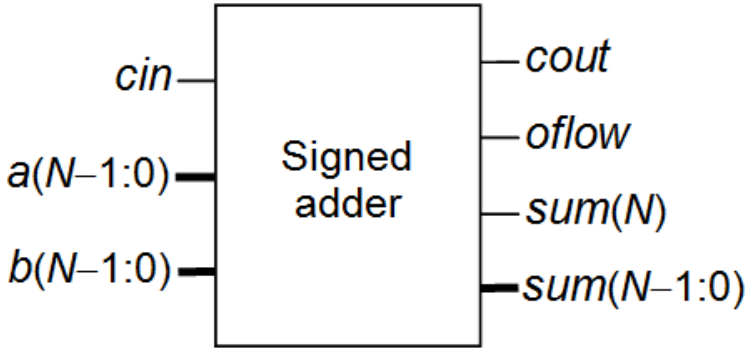
8. Doing math right with VHDL

Example:



8. Doing math right with VHDL

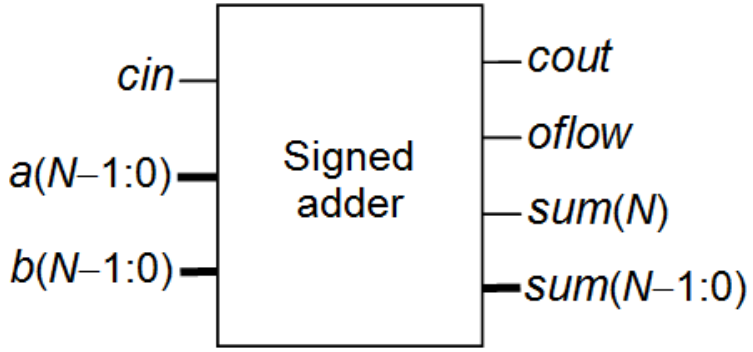
Example:



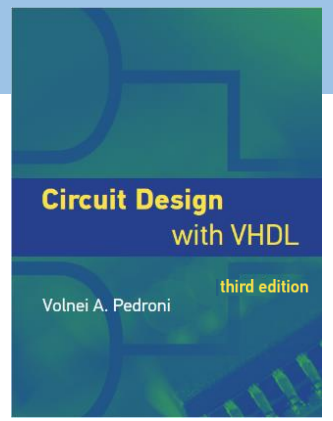

```
1  ----------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  entity adder_signed is
7     generic (
8        NUM_BITS: integer := 4);
9     port (
10       a, b: in std_logic_vector(NUM_BITS-1 downto 0);
11       cin: in std_logic;
12       sum: out std_logic_vector(NUM_BITS-1 downto 0);
13       sumMSB, cout, oflow: out std_logic);
14 end entity;
15
```

8. Doing math right with VHDL

Example:




```
16  architecture suggested of adder_signed is
17     signal sum_sig: signed(NUM_BITS downto 0);
18  begin
19
20     --Sign-extension, conversion to signed, and addition:
21     sum_sig <= signed(a(NUM_BITS-1) & a) + signed(b) + cin;
22     --sum_sig <= resize(signed(a), NUM_BITS+1) + signed(b) + cin;
23
24     --Conversion to std_logic_vector and final operations:
25     sum <= std_logic_vector(sum_sig(NUM_BITS-1 downto 0));
26     sumMSB <= sum_sig(NUM_BITS);
27     cout <= a(NUM_BITS-1) xor b(NUM_BITS-1) xor sumMSB;
28     oflow <= sum_sig(NUM_BITS) xor sum_sig(NUM_BITS-1);
29
30  end architecture;
31  ----------------------------------------------------------------
```



End of Chapter 10
(and 11)