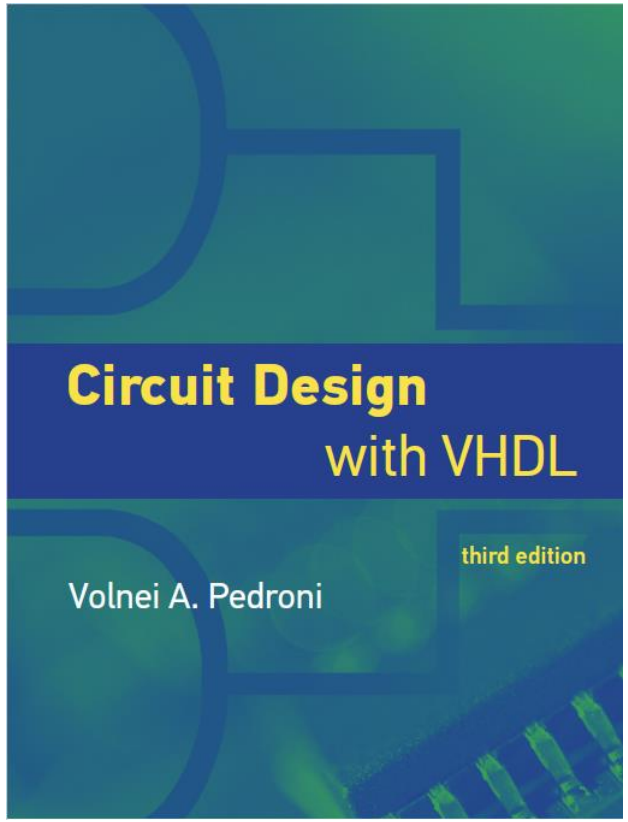


Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 9

Operators and Attributes

Revision 1

Book Contents

Part I: Digital Circuits Review

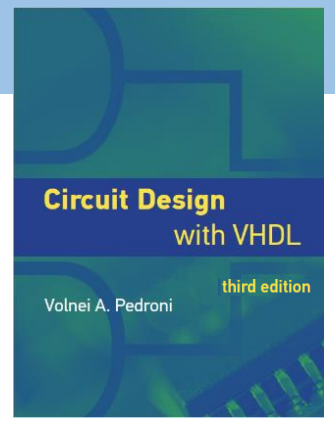
1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

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

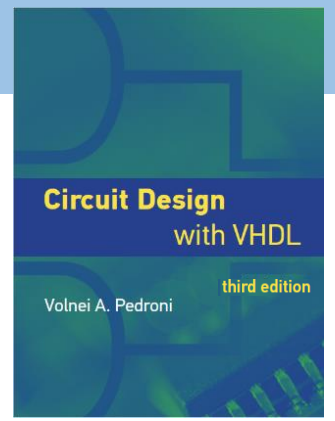
Appendices

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

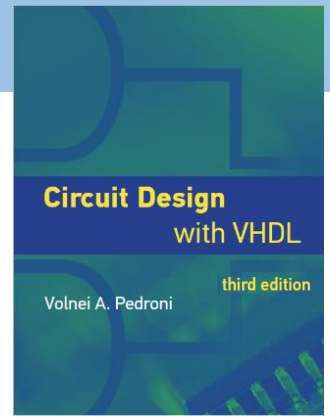
Chapter	Title
5	Introduction to VHDL
6	Code Structure and Composition
7	Predefined Data Types
8	User-Defined Data Types
9	Operators and Attributes
10	Concurrent Code
11	Concurrent Code – Practice
12	Sequential Code
13	Sequential Code – Practice
14	Packages and Subprograms



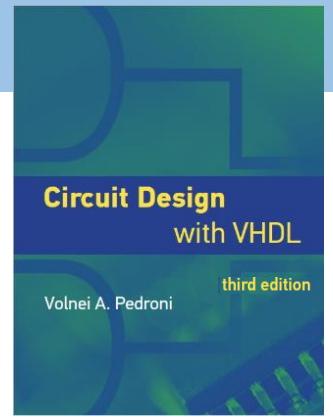
Chapter 9

Operators and Attributes

1. Operators
2. Overloaded operators
3. Attributes
4. Synthesis attributes
5. Alias



1. Operators



1. Operators

The predefined operators are divided into six categories:

a) Logical operators

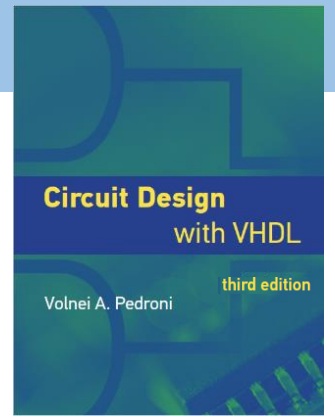
b) Arithmetic operators

c) Comparison (relational) operators

d) Shift operators

e) Concatenation operator

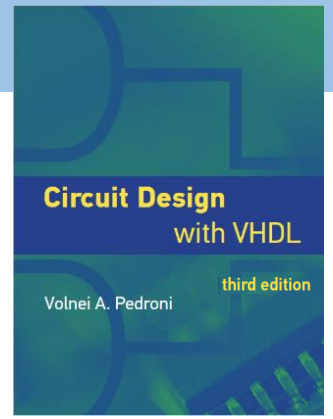
f) Condition operator



1. Operators

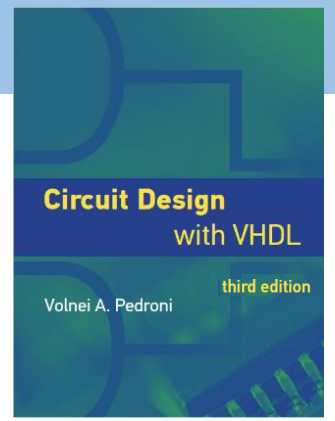
Table 9.1
Predefined VHDL operators

Category	Operators	Comments
Logical	**not**	The *not* operator
	and, nand, or, nor, xor, xnor	Regular and reduction logical operators
Arithmetic	–	Negation operator
	+, –, *, /, **, **mod, rem, abs**	Regular arithmetic operators
Comparison	=, /=, >, <, >=, <=	Regular comparison operators
	?=, ?/=, ?>, ?<, ?>=, ?<=	Matching comparison operators
	minimum, maximum	Additional comparison functions
Shift	**sll, srl, rol, ror**	Logical shift operators
	sla, sra	Arithmetic shift operators
Concatenation	&	Array-forming operator
Condition	??	Converts bit or std_ulogic to boolean



1. Operators

a) Logical operators



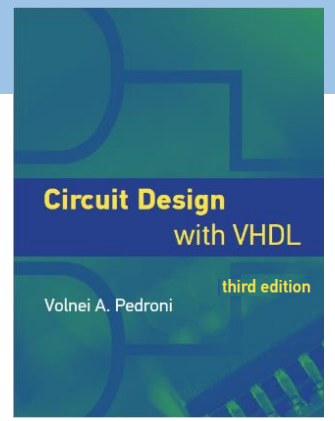
1. Operators

a) Logical operators

Binary (two-input) operators	
and	Regular *and* operator
nand	Regular *nand* operator
or	Regular *or* operator
nor	Regular *nor* operator
xor	Regular *xor* operator
xnor	Regular *xnor* operator

Unary (single-input) operators	
not	The *not* operator
and	Reduction *and* operator
nand	Reduction *nand* operator
or	Reduction *or* operator
nor	Reduction *nor* operator
xor	Reduction *xor* operator
xnor	Reduction *xnor* operator

Note: Unary operators have precedence over binary operators



1. Operators

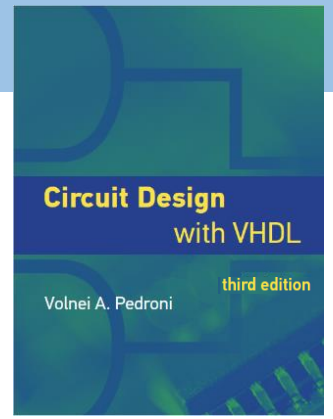
a) Logical operators

Table 9.2
Logical operators (including output sizes)

VHDL-2008:

Package	Binary operations			Unary operations					
	Regular `and, nand, or, nor, xor, xnor`			Reduction `and, nand, or, nor, xor, xnor`			`not` operator		
	Left, Right	Outp	Size	Inp	Outp	Size	Inp	Outp	Size
standard	B, B	B	(1)	BV	B	(3)	B	B	(4)
	BV, BV	BV		—	—		BV	BV	
	BV, B	BV	(2)	—	—		—	—	
	BO, BO	BO	(1)	BOV	BO		BO	BO	
	BOV, BOV	BOV		—	—		BOV	BOV	
	BOV, BO	BOV	(2)	—	—		—	—	
std_logic_1164	SU, SU	SU	(1)	SUV	SU	(3)	SU	SU	(4)
	SUV, SUV	SUV		—	—		SUV	SUV	

See complete table in section 9.1.1

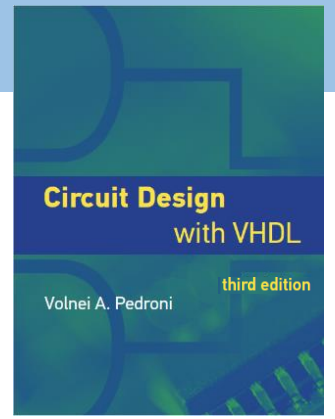


1. Operators

a) Logical operators

Examples of *unary* logical operations:

```
and "1110"  = '1' and '1' and '1' and '0' = '0'
nand "1110" = not (and "1110") = not ('0') = '1'
```



1. Operators

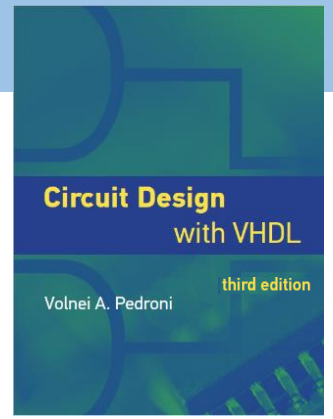
a) Logical operators

Examples of *unary* logical operations:

```
and "1110"  = '1' and '1' and '1' and '0' = '0'
nand "1110" = not (and "1110") = not ('0') = '1'
```

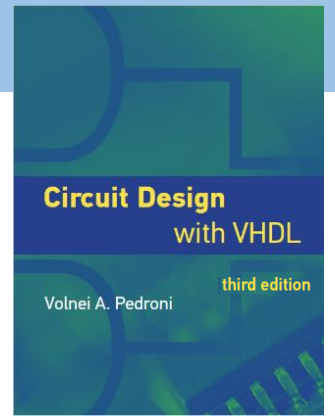
Examples of *binary* logical operations:

```
'1' and '1' = '1'
"0011" xor "0101" = "0110"
"0011" or not "0101" = "0011" or (not "0101") = "1011"
'1' and "0011" = ('1' and '0')('1' and '0')('1' and '1')('1' and '1') = "0011"
```



1. Operators

b) Arithmetic operators

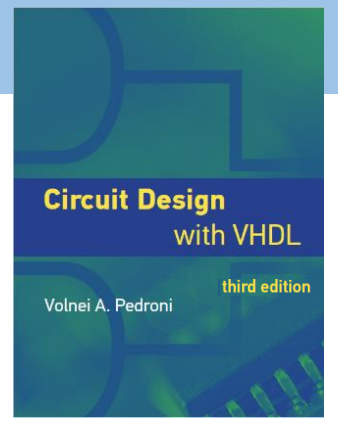


1. Operators

b) Arithmetic operators

Binary (two-input) operators	
+	Addition operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
**	Exponentiation operator
rem	Remainder operator
mod	Modulo operator

Unary (single-input) operators	
-	Negation operator
abs	Absolute-value operator



1. Operators

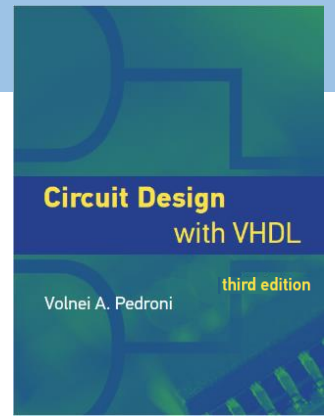
b) Arithmetic operators

Table 9.4
Arithmetic operators (including output sizes or ranges)

VHDL-2008:

Binary operations							Unary operations		
Left, Right	Outp	+, -	*	/	**	rem, mod	Inp	Outp	-, abs
INT, INT	INT	(1)	(2)	(3)	(4) (5)	(6)	INT	INT	(7)
UNS, UNS	UNS	(8)	(11)	(14)	–	(15)	—	—	—
UNS, NAT		(9)	(12)	(9)	–	(9)	—	—	—
UNS, SU		(9)	–	–	–	–	—	—	—
SIG, SIG	SIG	(8)	(11)	(14)	–	(15)	SIG	SIG	(10)
SIG, INT		(10)	(13)	(10)	–	(10)	—	—	—
SIG, SU		(10)	–	–	–	–	—	—	—
UFIX, UFIX	UFIX	(16)	(18)	(20)	–	(24)	—	—	—

See complete table in section 9.1.2

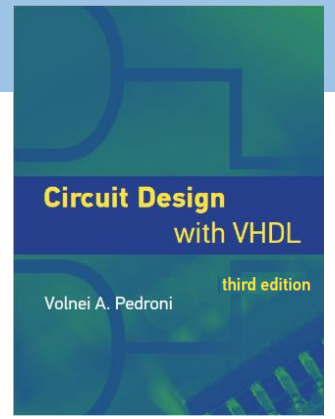


1. Operators

b) Arithmetic operators

Rules can be complex (see complete set in table 9.4):

```
For integer (virtually any size is legal; below are the minimum overflow-free sizes):
(1) Outp'length = maximum(L'length, R'length) + 1)
(2) Outp'length = L'length + R'length
...
For unsigned and signed (ranges normalized to "… downto 0"):
(8) maximum(L'length, R'length)-1 downto 0 (output length = largest input length)
(9) UNS'length-1 downto 0 (output length = length of unsigned input)
...
For float:
(28) maximum(L'left, R'left) downto minimum(L'right, R'right)
(29) FLO'range (range of float input)
```



1. Operators

b) Arithmetic operators

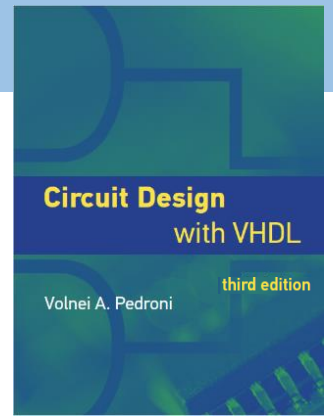
Rules can be complex (see complete set in table 9.4):

```
For integer (virtually any size is legal; below are the minimum overflow-free sizes):
(1) Outp'length = maximum(L'length, R'length) + 1)
(2) Outp'length = L'length + R'length
...
For unsigned and signed (ranges normalized to "… downto 0"):
(8) maximum(L'length, R'length)-1 downto 0 (output length = largest input length)
(9) UNS'length-1 downto 0 (output length = length of unsigned input)
...
For float:
(28) maximum(L'left, R'left) downto minimum(L'right, R'right)
(29) FLO'range (range of float input)
```

Examples:

For UNS and SIG, length of sum or subtraction must be the same as that of longest input

For UNS and SIG, length of multiplication must be equal to the sum of the input lengths

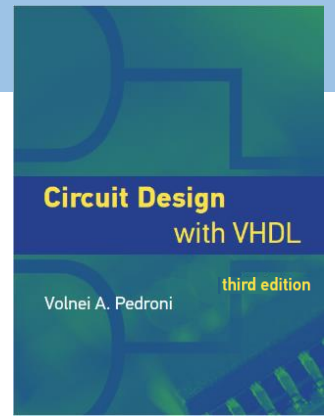


1. Operators

b) Arithmetic operators

Examples involving types (all rules are in table 9.4):

Examples involving ranges (all rules are in table 9.4):



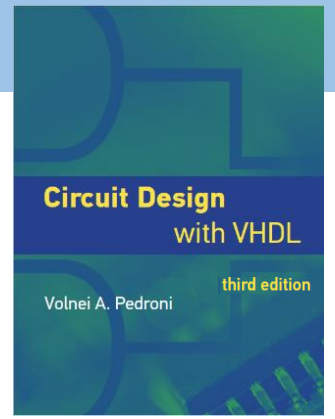
1. Operators

b) Arithmetic operators

Examples involving types (all rules are in table 9.4):

```
int <= int + int;      --Legal (INT+INT=INT available in table 9.4)
sig <= sig - int;      --Legal (SIG-INT=SIG available in table 9.4)
uns <= uns - sig;      --Illegal (UNS-SIG=UNS not available in table 9.4)
flo <= flo - sfix;     --Illegal (FLO-SFIX=FLO not available in table 9.4)
```

Examples involving ranges (all rules are in table 9.4):



1. Operators

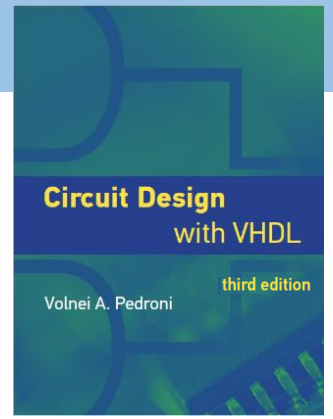
b) Arithmetic operators

Examples involving types (all rules are in table 9.4):

```
int <= int + int;      --Legal (INT+INT=INT available in table 9.4)
sig <= sig - int;      --Legal (SIG-INT=SIG available in table 9.4)
uns <= uns - sig;      --Illegal (UNS-SIG=UNS not available in table 9.4)
flo <= flo - sfix;     --Illegal (FLO-SFIX=FLO not available in table 9.4)
```

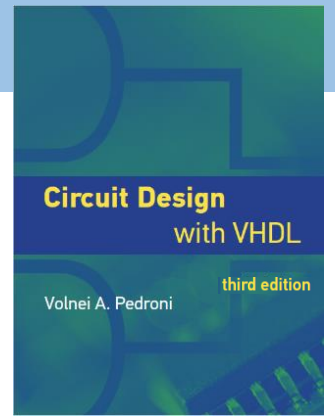
Examples involving ranges (all rules are in table 9.4):

```
sig(6:0) <= sig(5:0) + su + su + su;   --Illegal (output should be sig(5:0))
ufix(5:-4) <= ufix(5:-1) - ufix(3:-3); --illegal (output should be ufix(6:-3))
flo(9:-7) <= flo(5:-5) + flo(4:-2);    --illegal (output should be flo(5:-5))
```



1. Operators

c) Comparison operators



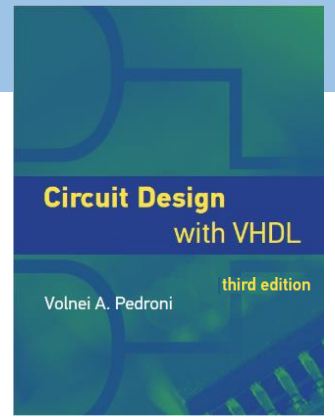
1. Operators

c) Comparison operators

Regular comparison operators	
=	Equality
/=	Inequality
<	Smaller than
<=	Smaller than or equal to
>	Greater than
>=	Greater than or equal to
minimum(), maximum()	

Matching comparison operators	
?=	Equality
?/=	Inequality
?<	Smaller than
?<=	Smaller than or equal to
?>	Greater than
?>=	Greater than or equal to

- Regular comparison: All nine SU values are considered to be different
- Matching comparison: Considers `'0'='L'`, `'1'='H'`, `'–'` = any other value



1. Operators

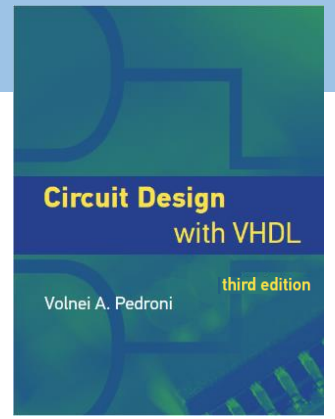
c) Comparison operators

Regular comparison operators	
=	Equality
/=	Inequality
<	Smaller than
<=	Smaller than or equal to
>	Greater than
>=	Greater than or equal to
minimum(), maximum()	

Matching comparison operators	
?=	Equality
?/=	Inequality
?<	Smaller than
?<=	Smaller than or equal to
?>	Greater than
?>=	Greater than or equal to

Allowed compositions:

```
x <= '1' when a=b else '0';   --with regular comparison
x <= a ?= b;                  --with matching comparison
```



1. Operators

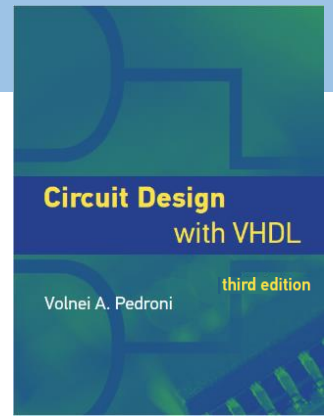
c) Comparison operators

Table 9.5
Comparison operators (main options)

VHDL-2008:

Package	Regular comparison		Matching comparison		min/max functions	
	Left, Right	Outp	Left, Right	Outp	Left, Right	Outp
standard	B, B	BO	B, B	B	B, B	B
	BV, BV	BO	BV, BV[1]	B	BV, BV	BV
	BO, BO	BO	—	—	BO, BO	BO
	BOV, BOV	BO	—	—	BOV, BOV	BOV
	INT, INT	BO	—	—	INT, INT	INT
	INTV, INTV	BO	—	—	INTV	INT
	CHAR, CHAR	BO	—	—	CHAR, CHAR	CHAR
	STR, STR	BO	—	—	STR	CHAR
std_logic_1164[2]	SU, SU	BO	SU, SU	SU	—	—
	SUV, SUV	BO	SUV, SUV1	SU	—	—
	UNS, UNS	BO	UNS, UNS	SU	UNS, UNS	UNS

See complete table in section 9.1.2



1. Operators

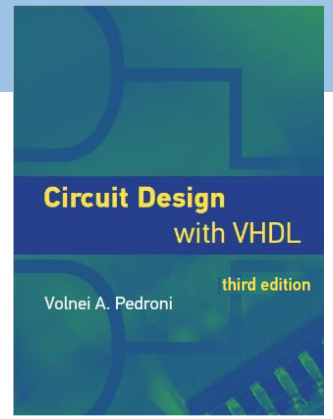
c) Comparison operators

How comparisons are made:

- For INT, UNS/SIG, UFIX/SFIX, FLO, … : The “usual” way
- For BV and SUV/SLV: “Unusual” way

  Equality test: Vectors must have same length

  Other tests: Rightmost bits of longer vector are discarded; if then the vectors become equal, the longer vector is considered larger

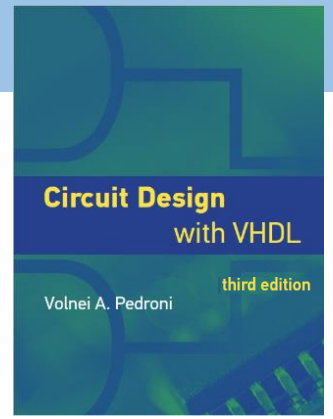


1. Operators

c) Comparison operators

Examples with UNS:

```
"1111" > "011110" ?
"1100" = "001100" ?
```

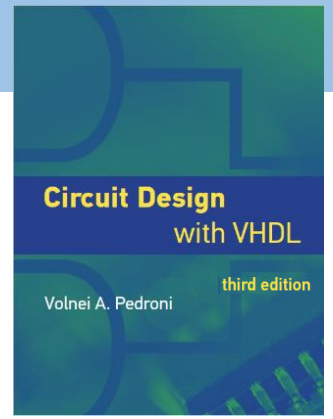


1. Operators

c) Comparison operators

Examples with UNS:

```
"1111" > "011110" false (15 < 30)
"1100" = "001100" true  (12 = 12)
```



1. Operators

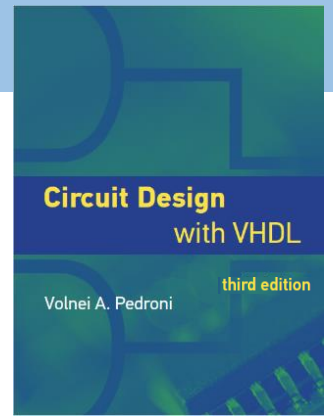
c) Comparison operators

Examples with UNS:

```
"1111" > "011110" false (15 < 30)
"1100" = "001100" true  (12 = 12)
```

Examples with SIG:

```
"1000" < "000011" ?
"1100" = "001100" ?
```



1. Operators

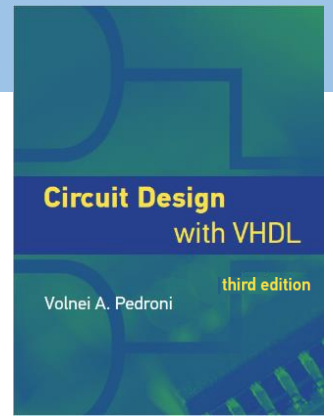
c) Comparison operators

Examples with UNS:

```
"1111" > "011110" false (15 < 30)
"1100" = "001100" true  (12 = 12)
```

Examples with SIG:

```
"1000" < "000011" true (–8 < 3)
"1100" = "001100" false (–4 < 12)
```



1. Operators

c) Comparison operators

Examples with UNS:

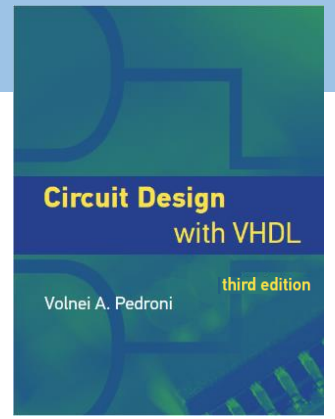
```
"1111" > "011110" false (15 < 30)
"1100" = "001100" true  (12 = 12)
```

Examples with SIG:

```
"1000" < "000011" true (–8 < 3)
"1100" = "001100" false (–4 < 12)
```

Examples with BV or SUV/SLV:

```
"1000" > "100000"  ?
"1111" > "011111"  ?
"01000" = "001000" ?
```



1. Operators

c) Comparison operators

Examples with UNS:

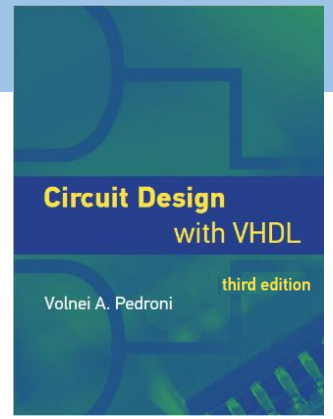
```
"1111" > "011110" false (15 < 30)
"1100" = "001100" true  (12 = 12)
```

Examples with SIG:

```
"1000" < "000011" true (–8 < 3)
"1100" = "001100" false (–4 < 12)
```

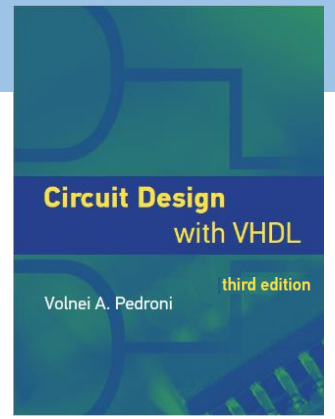
Examples with BV or SUV/SLV:

```
"1000" > "100000"  false (truncation: "1000" < "1000"; so longer vector wins)
"1111" > "011111"  true (truncation: "1111" > "0111"; so left vector wins)
"01000" = "001000" false (never equal when sizes are different)
```



1. Operators

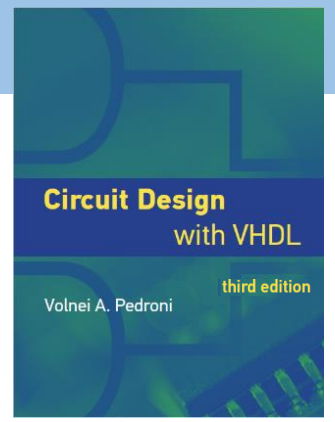
d) Shift operators



1. Operators

d) Shift operators

`sll`	Shift left logical (positions on the right are filled with zeros)
`srl`	Shift right logical (positions on the left are filled with zeros)
`sla`	Shift left arithmetic (rightmost bit is replicated on the right)
`sra`	Shift right arithmetic (leftmost bit is replicated on the left)
`rol`	Rotate left (circular shift to the left, reentering on the right)
`ror`	Rotate right (circular shift to the right, reentering on the left)



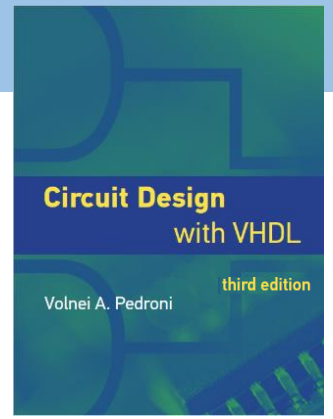
1. Operators

d) Shift operators

Table 9.6
Shift operators

VHDL-2008:

Package	Type	`sll, srl, rol, ror`		`sla, sra`	
		Left, Right	Outp	Left, Right	Outp
standard	BV	BV, INT	BV	BV, INT	BV
	BOV	BOV, INT	BOV	BOV, INT	BOV
std_logic_1164	SUV	SUV, INT	SUV	—	—
numeric_std	UNS	UNS, INT	UNS	UNS, INT	UNS
	SIG	SIG, INT	SIG	SIG, INT	SIG
fixed_generic_pkg	UFIX	UFIX, INT	UFIX	UFIX, INT	UFIX
	SFIX	SFIX, INT	SFIX	SFIX, INT	SFIX

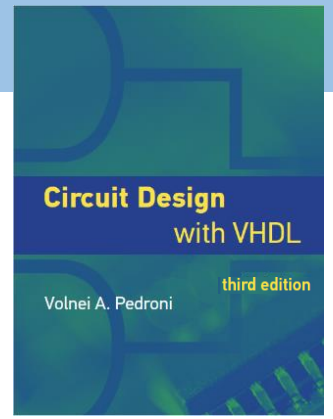


1. Operators

d) Shift operators

Examples:

```
"100011" sll 2  = ?
"100100" sra 3  = ?
"100110" ror -2 = ?
```

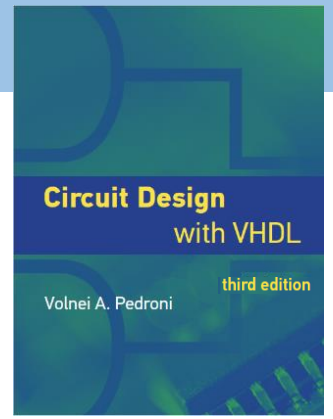


1. Operators

d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = ?
"100110" ror -2 = ?
```

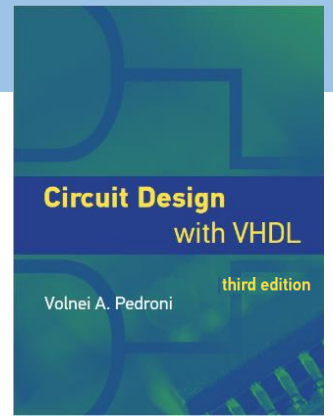


1. Operators

d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = "111100"
"100110" ror -2 = ?
```

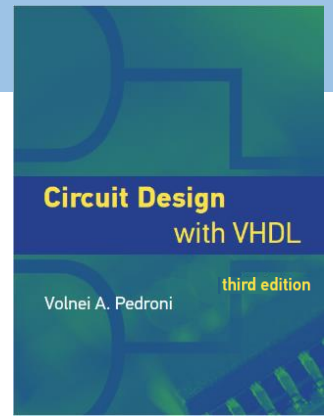


1. Operators

d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = "111100"
"100110" ror -2 = "011010"
```



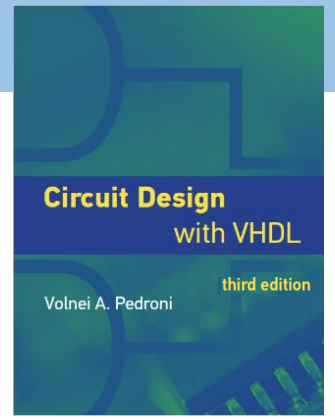
1. Operators

d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = "111100"
"100110" ror -2 = "011010"
```

IMPORTANT: Shift can be done with the concatenation operator (&):



1. Operators

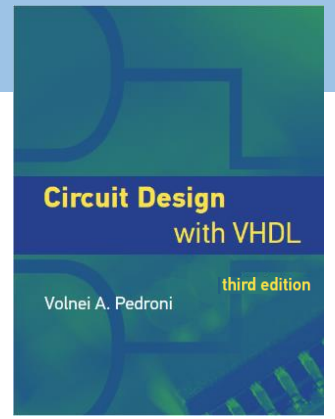
d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = "111100"
"100110" ror -2 = "011010"
```

IMPORTANT: Shift can be done with the concatenation operator (&):

```
signal x, y, z: std_logic_vector(5 downto 0);
...
y <= x sll 2;   →   y <= ?
z <= x sra 2;   →   z <= ?
```



1. Operators

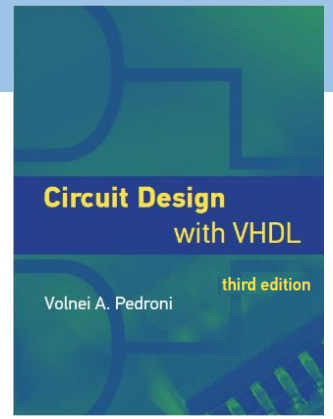
d) Shift operators

Examples:

```
"100011" sll 2  = "001100"
"100100" sra 3  = "111100"
"100110" ror -2 = "011010"
```

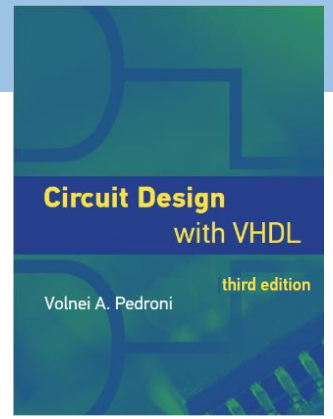
IMPORTANT: Shift can be done with the concatenation operator (&):

```
signal x, y, z: std_logic_vector(5 downto 0);
...
y <= x sll 2;   →   y <= x(3 downto 0) & "00";
z <= x sra 2;   →   z <= x(5) & x(5) & x(5 downto 2);
```



1. Operators

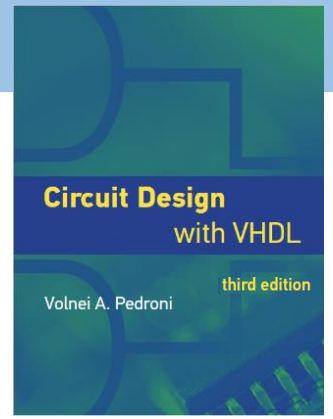
e) Concatenation operator (&)



1. Operators

e) Concatenation operator (&)

- Review section 7.9.2 and its many examples

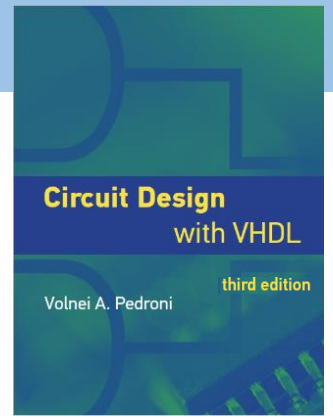


1. Operators

e) Concatenation operator (&)

- Review section 7.9.2 and its many examples

f) Condition operator (??)



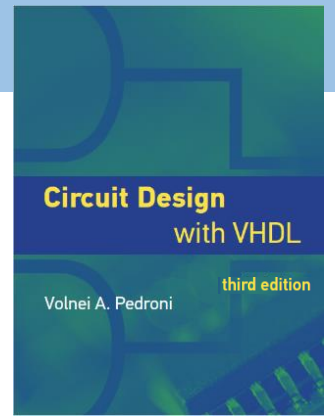
1. Operators

e) Concatenation operator (&)

- Review section 7.9.2 and its many examples

f) Condition operator (??)

- Converts a `bit` or `std_ulogic/std_logic` value to `boolean`
- Can be used implicitly (recommended)



1. Operators

e) Concatenation operator (&)

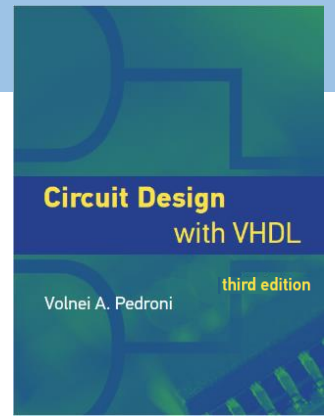
- Review section 7.9.2 and its many examples

f) Condition operator (??)

- Converts a `bit` or `std_ulogic/std_logic` value to `boolean`
- Can be used implicitly (recommended)

Example:

```
if x='1' then ...     --before VHDL 2008
if ?? x then ...      --after VHDL 2008, explicit
if x then ...         --after VHDL 2008, implicit (recommended)
```



1. Operators

e) Concatenation operator (&)

- Review section 7.9.2 and its many examples

f) Condition operator (??)

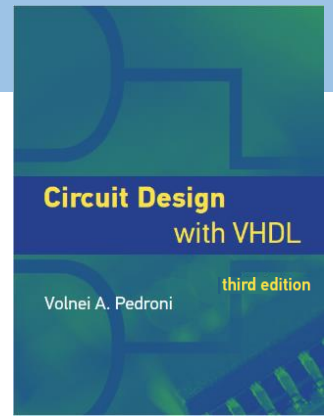
- Converts a `bit` or `std_ulogic/std_logic` value to `boolean`
- Can be used implicitly (recommended)

Example:

```
if x='1' then ...     --before VHDL-2008
if ?? x then ...      --after VHDL-2008, explicit
if x then ...         --after VHDL-2008, implicit (recommended)
```

Another example:

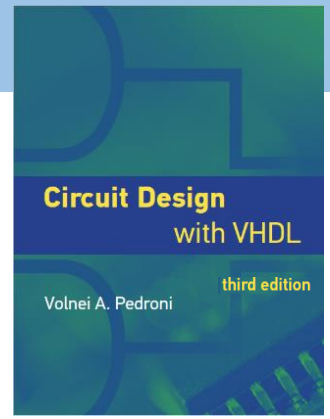
```
y <= x when a='1' and b'=1' and c='0' else '0';  --before VHDL-2008
y <= x when a and b and not c else '0';          --after VHDL-2008, implicit
```



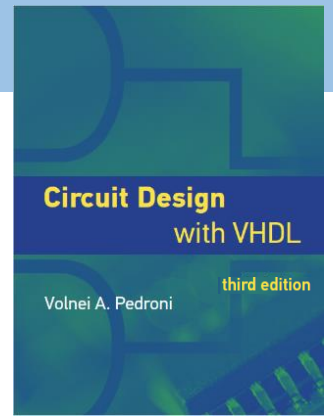
Chapter 9

Operators and Attributes

1. Operators
2. Overloaded operators
3. Attributes
4. Synthesis attributes
5. Alias



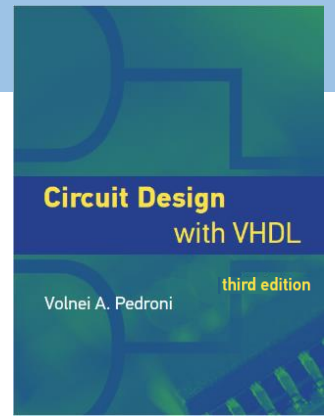
2. Overloaded operators



2. Overloaded operators

- An operator for which more than one construction exists
- Nearly all predefined operators are overloaded, like the following:

```
function "+" (L, R: integer) return integer;
function "+" (L, R: unsigned) return unsigned;
...
```

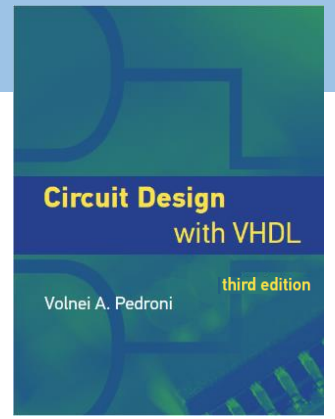


2. Overloaded operators

- An operator for which more than one construction exists
- Nearly all predefined operators are overloaded, like the following:

```
function "+" (L, R: integer) return integer;
function "+" (L, R: unsigned) return unsigned;
...
```

Example: There is no "+" operator for BIT+INT and INT+BIT. This operator is further overloaded by the functions below.



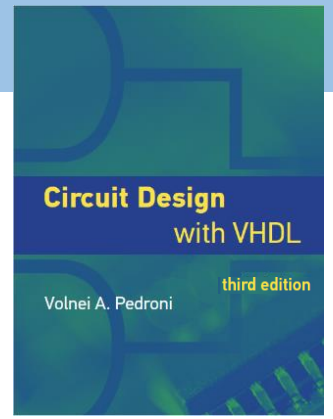
2. Overloaded operators

- An operator for which more than one construction exists
- Nearly all predefined operators are overloaded, like the following:

```
function "+" (L, R: integer) return integer;
function "+" (L, R: unsigned) return unsigned;
...
```

Example: There is no "+" operator for BIT+INT and INT+BIT. This operator is further overloaded by the functions below.

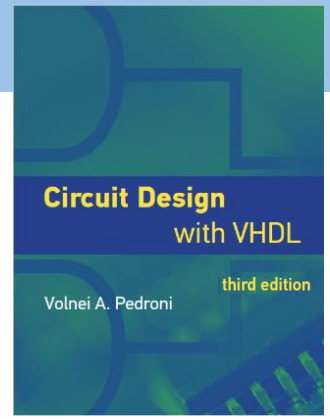
```
function "+" (L: integer; R: bit) return integer is
begin
   if R then return L+1;
   else return L;
   end if;
end function "+";

function "+" (L: bit; R: integer) return integer is
  ...
end function "+";
```



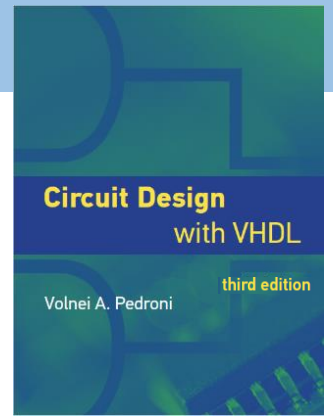
Chapter 9

Operators and Attributes

1. Operators
2. Overloaded operators
3. Attributes
4. Synthesis attributes
5. Alias

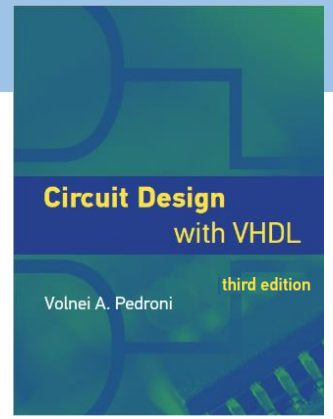


3. Attributes



3. Attributes

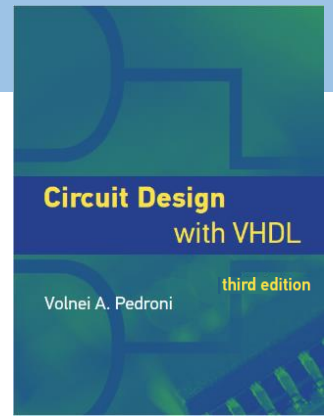
- The predefined attributes are divided into four categories:
  a) Attributes of scalar types
  b) Attributes of array types and objects
  c) Attributes of signals
  d) Attributes of named entities



3. Attributes

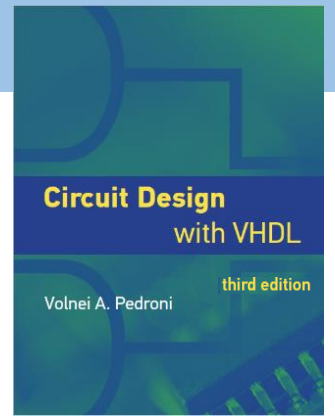
- The predefined attributes are divided into four categories:
  a) Attributes of scalar types
  b) Attributes of array types and objects
  c) Attributes of signals
  d) Attributes of named entities

- Some are rarely used or not even implemented in synthesis tools



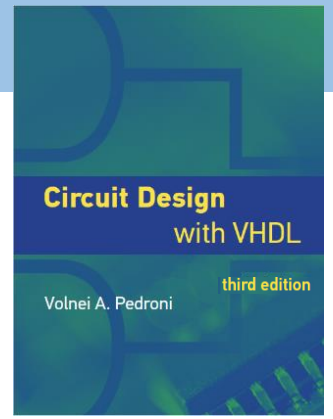
3. Attributes

- The predefined attributes are divided into four categories:
  a) Attributes of scalar types
  b) Attributes of array types and objects
  c) Attributes of signals
  d) Attributes of named entities
- Some are rarely used or not even implemented in synthesis tools
- Examples for only the truly needed will be shown next (full details are in sec. 9.3)



3. Attributes

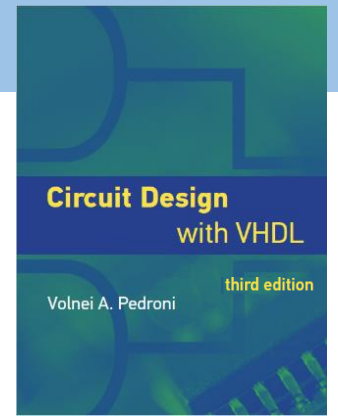
- The predefined attributes are divided into four categories:
  a) Attributes of scalar types
  b) Attributes of array types and objects
  c) Attributes of signals
  d) Attributes of named entities
- Some are rarely used or not even implemented in synthesis tools
- Examples for only the truly needed will be shown next (full details are in sec. 9.3)
- These attributes are always preceded by the "tick" symbol (')



3. Attributes

Example of very common attributes for array (or scalar) types

```
signal x: std_logic_vector(M downto N);
x'high → M
x'low → N
x'left → M
x'right → N
x'length → M–N+1
x'range → M downto N
x'reverse_range → N to M
x'ascending → false
```



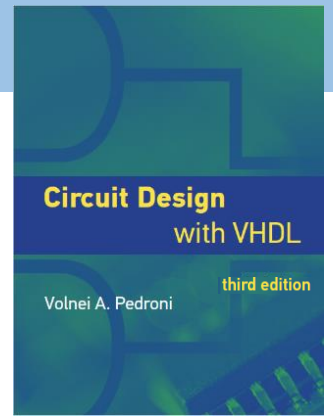
3. Attributes

Example of very common attributes for array (or scalar) types

```
signal x: std_logic_vector(M downto N);
x'high → M
x'low → N
x'left → M
x'right → N
x'length → M–N+1
x'range → M downto N
x'reverse_range → N to M
x'ascending → false
```

Related usage examples:

```
signal x: std_logic_vector(7 downto 0);
for i in x'range loop ...          --same as “for i in 7 downto 0 loop ...”
for i in x'low to x'high loop ...  --same as “for i in 0 to 7 loop ...”
if x'length /= y'length then ...   --comparison of two vector lengths
if x=(x'range => '0') then ...     --checks whether x="000…0" (sec. 8.4.2)
```



3. Attributes

Most common attribute for signals:

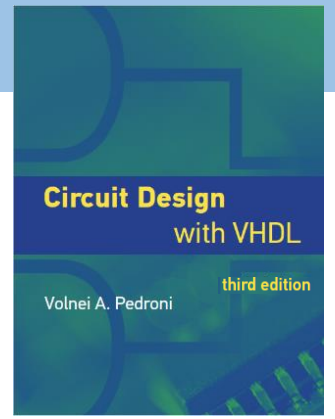
`'event` → informs that an event (change of level) has occurred

Also synthesizable, but not really needed:

`'stable` → informs that no event has occurred

For simulation (chapter 18):

`'quiet` → useful for detecting glitches



3. Attributes

Most common attribute for signals:

`'event` → informs that an event (change of level) has occurred

Also synthesizable, but not really needed:

`'stable` → informs that no event has occurred

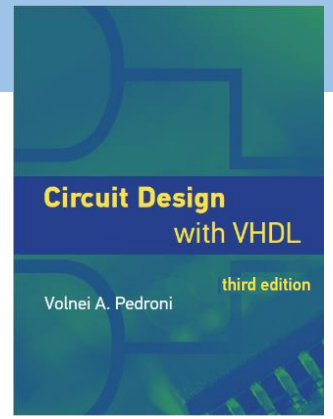
For simulation (chapter 18):

`'quiet` → useful for detecting glitches

Related usage examples:

```
signal clk: std_logic;
...
if clk'event and clk='1' then ...    --a rising-edge of clock
if clk'event and clk='0' then ...    --a falling-edge of clock
```

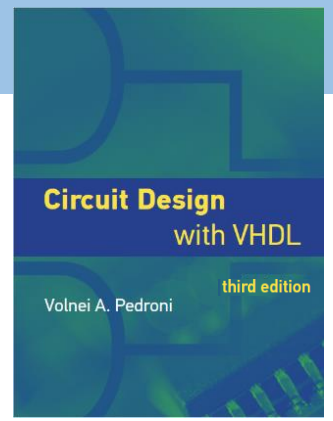
Note: Check functions *rising_edge* and *falling_edge* in section 12.2



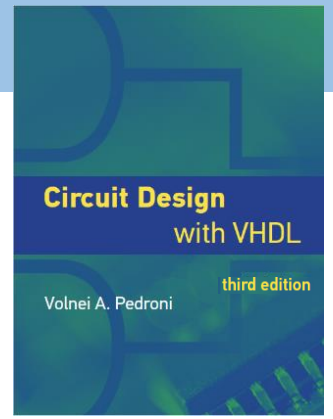
Chapter 9

Operators and Attributes

1. Operators
2. Overloaded operators
3. Attributes
4. Synthesis attributes
5. Alias

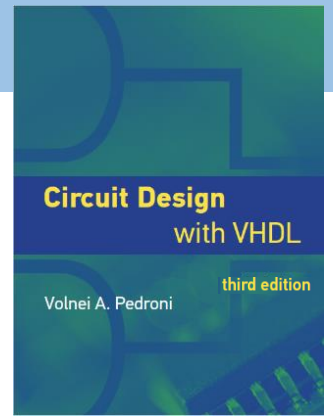


4. Synthesis attributes



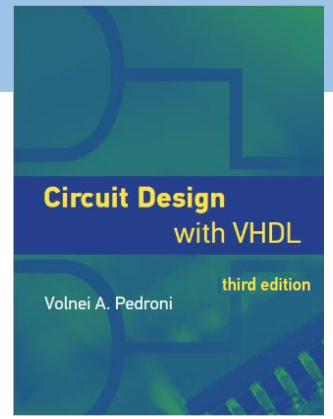
4. Synthesis attributes

- To communicate with the compiler (overrides its setup)
- Tick not used
- Only some of synthesis attributes are standardized



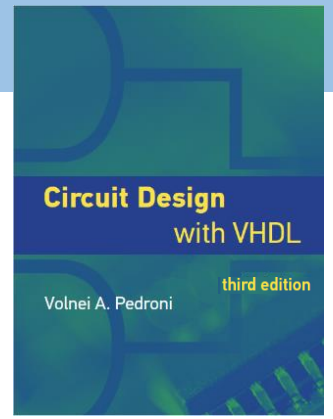
4. Synthesis attributes

- To communicate with the compiler (overrides its setup)
- Tick not used
- Only some of synthesis attributes are standardized
- Two examples will be shown here (see others in section 9.5):
  a) The "state machine encoding" attribute
  b) The "keep logic" (do not simplify it) attribute



4. Synthesis attributes

a) The “state machine encoding” attribute



4. Synthesis attributes

a) The “state machine encoding” attribute

- In Vivado

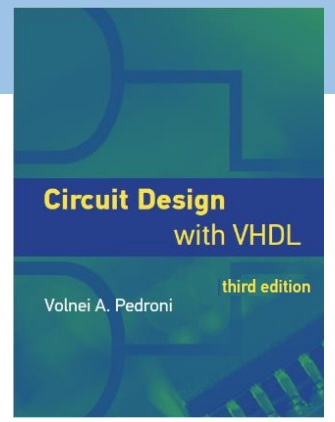
  Name: *fsm_encoding*

  Options: `"sequential"`, `"one_hot"`, `"gray"`, `"johnson"`, `"auto"` `(default)`

- In Quartus Prime

  Name: *syn_encoding*

  Options: `same as above plus "compact" (minimal bits) and user-encoded`



4. Synthesis attributes

a) The "state machine encoding" attribute

- In Vivado

  Name: *fsm_encoding*

  Options: `"sequential"`, `"one_hot"`, `"gray"`, `"johnson"`, `"auto"` `(default)`

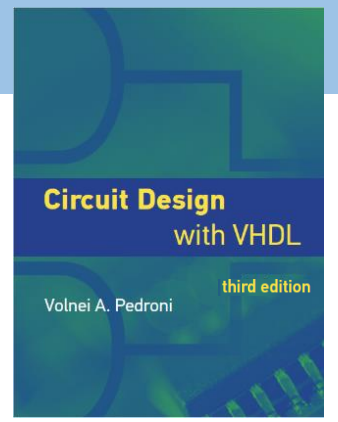
  Example:

```
type state_type is (A, B, C, D, E);
signal pr_state, nx_state: state_type;
attribute fsm_encoding: string;
attribute fsm_encoding of pr_state, nx_state: signal is "one_hot";
```

- In Quartus Prime

  Name: *syn_encoding*

  Options: `same as above plus "compact" (minimal bits) and user-encoded`



4. Synthesis attributes

a) The "state machine encoding" attribute

- In Vivado

  Name: *fsm_encoding*

  Options: `"sequential"`, `"one_hot"`, `"gray"`, `"johnson"`, `"auto"` (default)

  Example:

```
type state_type is (A, B, C, D, E);
signal pr_state, nx_state: state_type;
attribute fsm_encoding: string;
attribute fsm_encoding of pr_state, nx_state: signal is "one_hot";
```

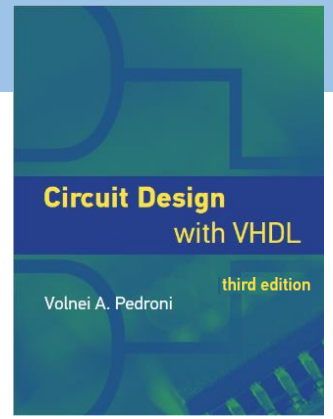
- In Quartus Prime

  Name: *syn_encoding*

  Options: same as above plus `"compact"` (minimal bits) and user-encoded

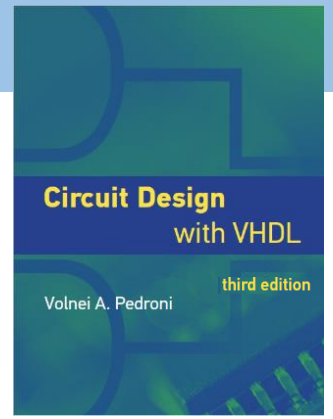
  Example:

```
type state_type is (A, B, C, D, E);
signal pr_state, nx_state: state_type;
attribute syn_encoding: string;
attribute syn_encoding of state_type: type is "0000 1111 0011 1100 0110";
```



4. Synthesis attributes

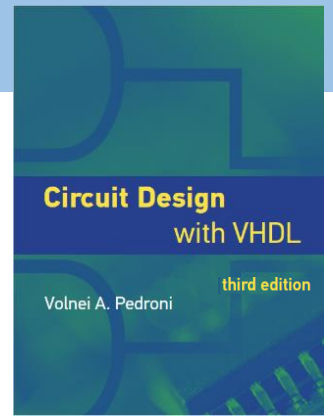
b) The *keep* attribute



4. Synthesis attributes

b) The *keep* attribute

- Prevents the simplification of combinational logic
- Syntax is the same in Vivado and Quartus Prime

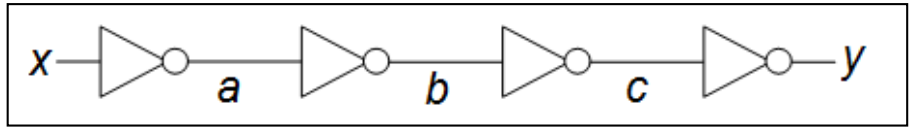


4. Synthesis attributes

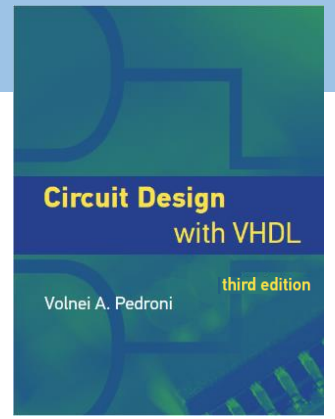
b) The *keep* attribute

- Prevents the simplification of combinational logic
- Syntax is the same in Vivado and Quartus Prime

Example: Delay line



Desired circuit

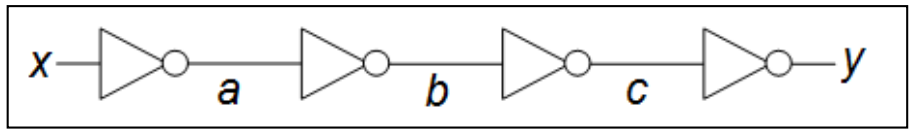


4. Synthesis attributes

b) The *keep* attribute

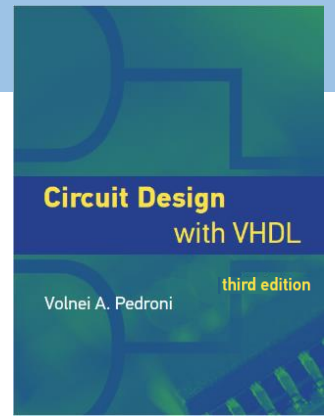
- Prevents the simplification of combinational logic
- Syntax is the same in Vivado and Quartus Prime

Example: Delay line



Desired circuit

```
architecture with_keep of delay_line is
  signal a, b, c: bit;
  attribute keep: boolean;
  attribute keep of a, b, c: signal is true;
begin
  a <= not x;
  b <= not a;
  c <= not b;
  y <= not c;
end architecture;
```

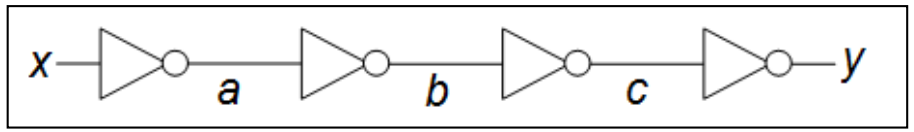


4. Synthesis attributes

b) The *keep* attribute

- Prevents the simplification of combinational logic
- Syntax is the same in Vivado and Quartus Prime

Example: Delay line



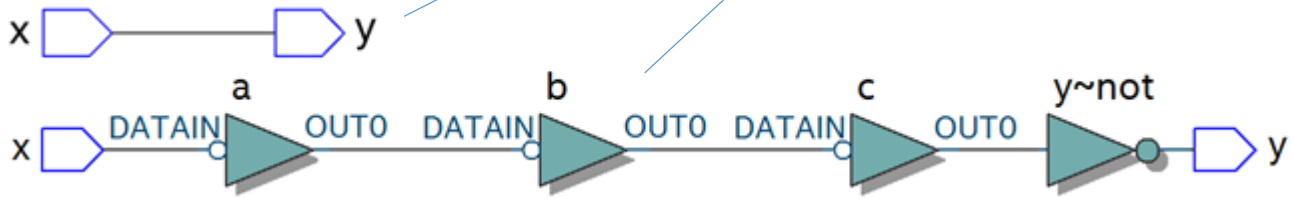

```
architecture with_keep of delay_line is
  signal a, b, c: bit;
  attribute keep: boolean;
  attribute keep of a, b, c: signal is true;
begin
  a <= not x;
  b <= not a;
  c <= not b;
  y <= not c;
end architecture;
```

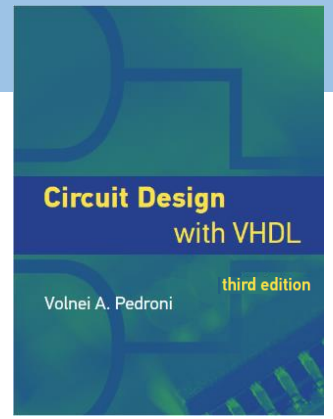
Desired circuit

Synthesis result

without *keep*

with *keep*

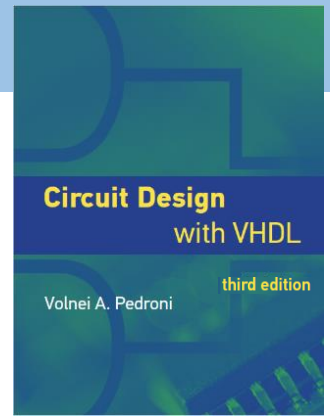




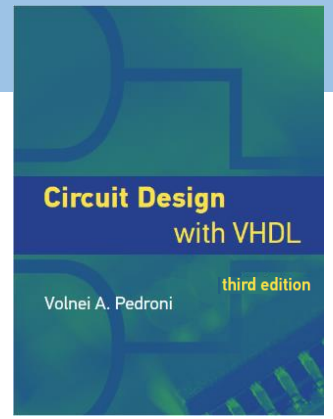
Chapter 9

Operators and Attributes

1. Operators
2. Overloaded operators
3. Attributes
4. Synthesis attributes
5. Alias

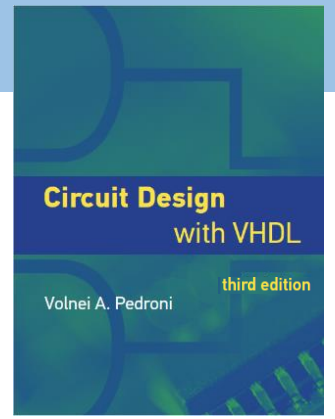


5. Alias



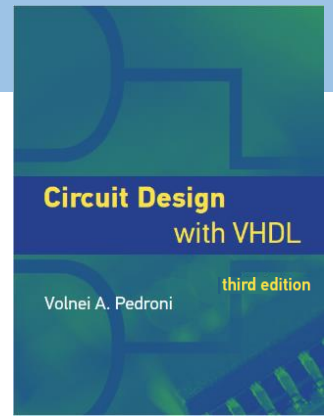
5. Alias

- Provides an alternative name to any named entity (signal, variable, …)
- Must obey the VHDL naming criteria (chapter 5)
- Usually done in the declarative part of a subprogram or of an architecture



5. Alias

- Provides an alternative name to any named entity (signal, variable, …)
- Must obey the VHDL naming criteria (chapter 5)
- Usually done in the declarative part of a subprogram or of an architecture
- There are three forms of aliasing:
  a) Simple object aliasing
  b) Object aliasing with type modification
  c) Non-object aliasing
- Cases (a) and (c) are described next; see case (b) in section 9.7



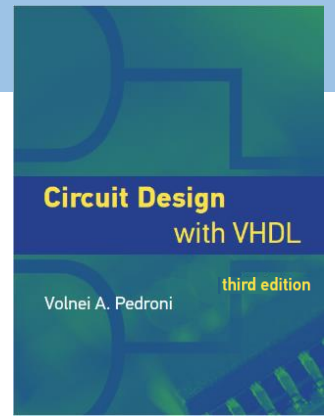
5. Alias

Simple object aliasing:

```
alias new_name is original_name;
```

Non-object aliasing:

```
alias new_name is original_name [signature];
```



5. Alias

Simple object aliasing:

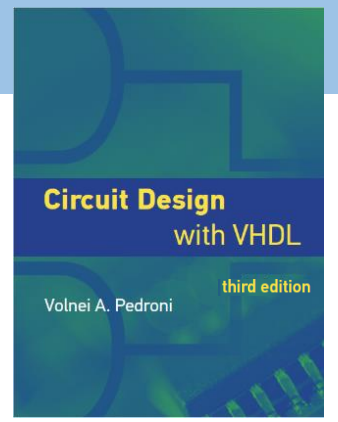
```
alias new_name is original_name;
```

Example:

```
signal floating_point_input: float32;
alias sign_bit is floating_point_input(8);
alias exponent is floating_point_input(7 downto 0);
alias fraction is floating_point_input(-1 downto -23);
```

Non-object aliasing:

```
alias new_name is original_name [signature];
```



5. Alias

Simple object aliasing:

```
alias new_name is original_name;
```

Example:

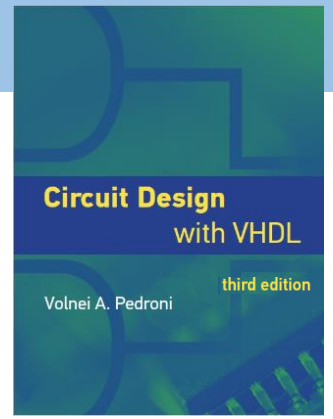
```
signal floating_point_input: float32;
alias sign_bit is floating_point_input(8);
alias exponent is floating_point_input(7 downto 0);
alias fraction is floating_point_input(-1 downto -23);
```

Non-object aliasing:

```
alias new_name is original_name [signature];
```

Example:

```
procedure sort (a, b: in integer; x, y: out integer) is ...
procedure sort (a, b: in unsigned; x, y: out unsigned) is ...
alias sort_int is sort [integer, integer, integer, integer];
alias sort_uns is sort [unsigned, unsigned, unsigned, unsigned];
```



End of Chapter 9