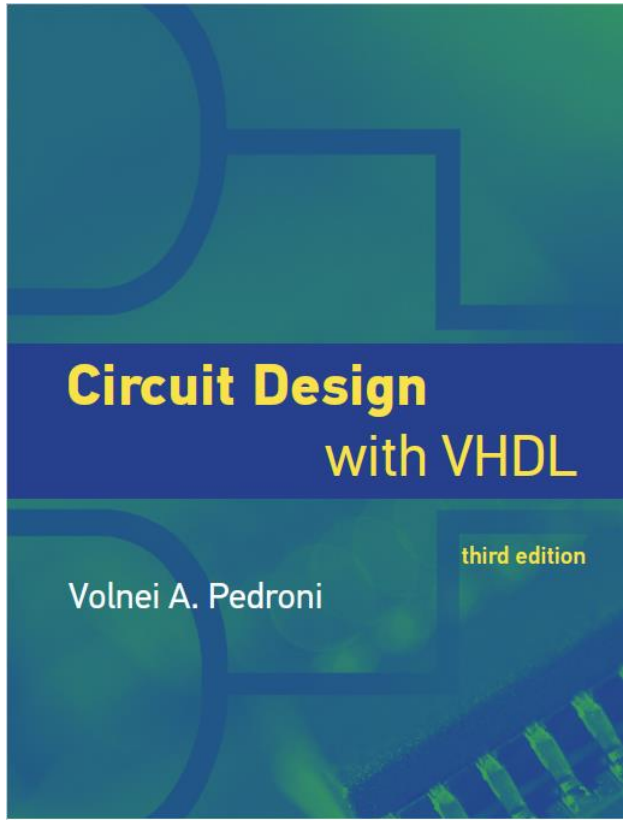


Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 7

Predefined Data Types

Revision 1

Book Contents

Part I: Digital Circuits Review

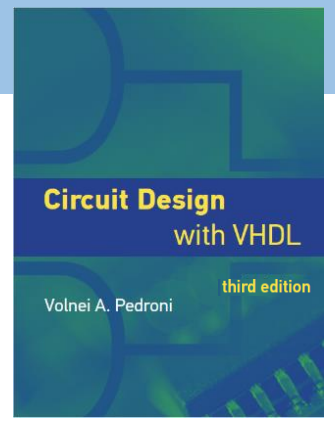
1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

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

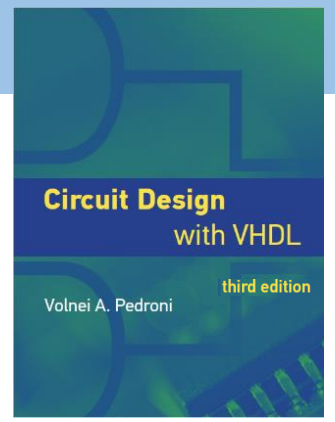
Appendices

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

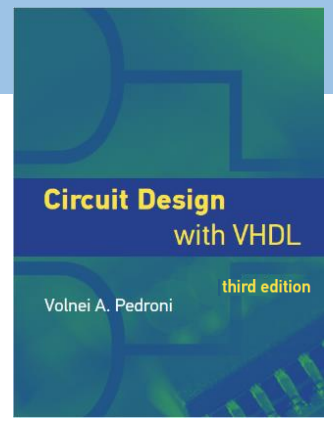
Chapter	Title
5	Introduction to VHDL
6	Code Structure and Composition
7	Predefined Data Types
8	User-Defined Data Types
9	Operators and Attributes
10	Concurrent Code
11	Concurrent Code – Practice
12	Sequential Code
13	Sequential Code – Practice
14	Packages and Subprograms



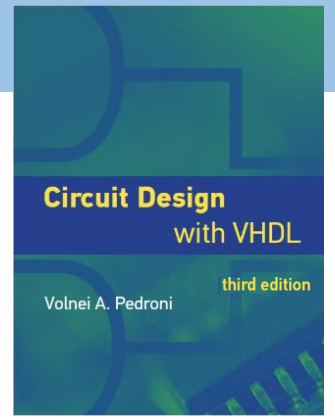
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

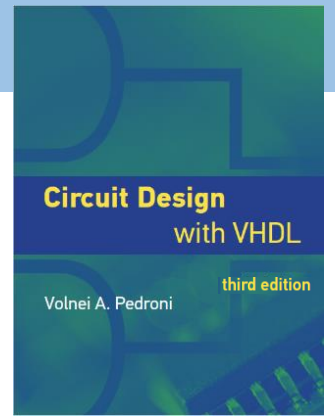


1. Main *synthesizable* types



1. Main *synthesizable* types

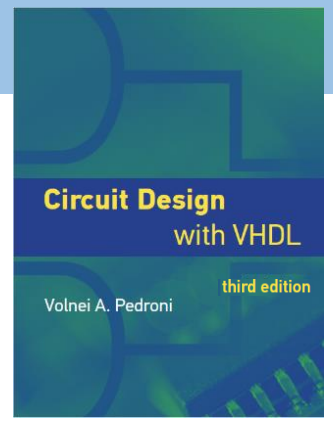
Category	Types	Packages
Standard types	bit, bit_vector boolean, boolean_vector integer, natural, positive, integer_vector character, string	*std*
Standard-logic types	std_ulogic, std_ulogic_vector std_logic, std_logic_vector	*std_logic_1164*
Unsigned/signed types	unsigned, signed	*numeric_std*
Fixed-point types	ufixed, sfixed	*fixed_generic_pkg* (*fixed_pkg*)
Floating-point type	float	*float_generic_pkg* (*float_pkg*)



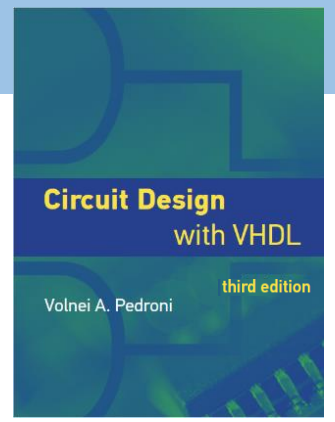
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions



2. Type declarations (how types are created)



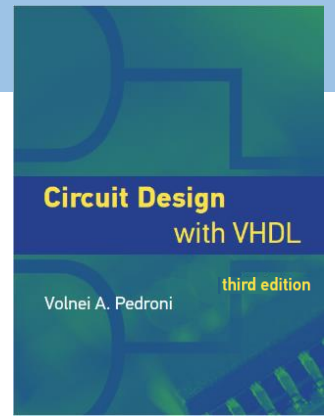
2. Type declarations (how types are created)

For scalar types:

```
type type_name is type_definition;
```

Examples (of original VHDL declarations):

```
type bit is ('0', '1');
type integer is range –2147483648 to 2147483647;
```



2. Type declarations (how types are created)

For scalar types:

```
type type_name is type_definition;
```

Examples (of original VHDL declarations):

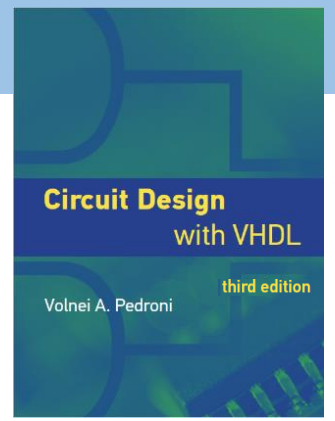
```
type bit is ('0', '1');
type integer is range –2147483648 to 2147483647;
```

For array types:

```
type type_name is array (range_spec) of base_type_name [range_spec]
```

Examples (of original VHDL declarations):

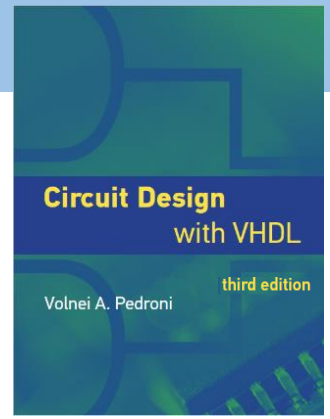
```
type bit_vector is array (natural range <>) of bit;
type integer_vector is array (natural range <>) of integer;
```



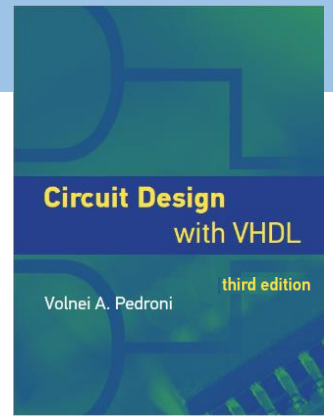
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

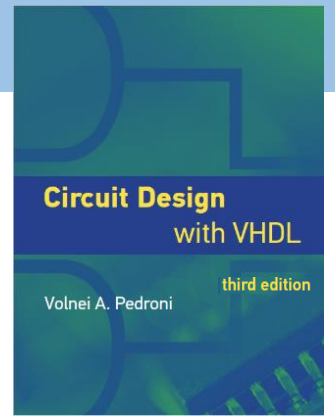


3. Subtypes



3. Subtypes

- Subtype = A type with a constraint
- Advantage (over a new type): Inherits all properties (operators and other functions) of parent type

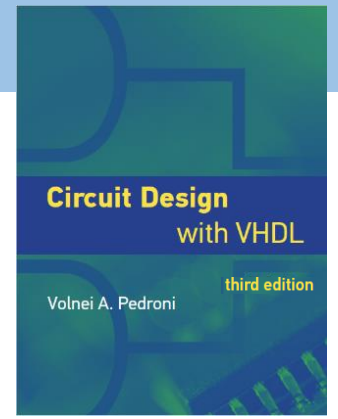


3. Subtypes

- Subtype = A type with a constraint
- Advantage (over a new type): Inherits all properties (operators and other functions) of parent type

Subtype declaration:

```
subtype subtype_name is [resolution_function] type_name [range specification];
```



3. Subtypes

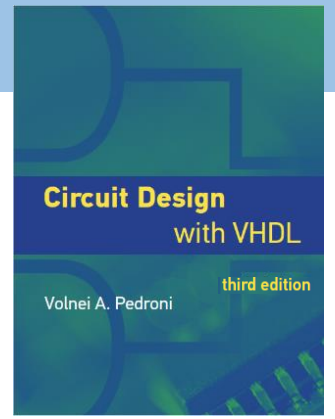
- Subtype = A type with a constraint
- Advantage (over a new type): Inherits all properties (operators and other functions) of parent type

Subtype declaration:

```
subtype subtype_name is [resolution_function] type_name [range specification];
```

Example (of original VHDL declaration):

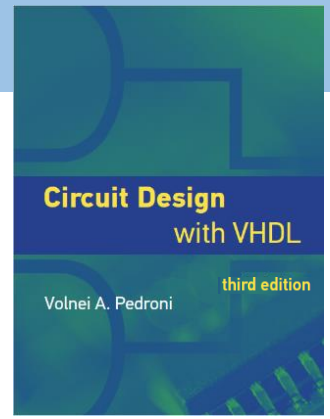
```
type integer is range –2147483648 to 2147483647;     --range –2^31 to 2^31–1
subtype natural is integer range 0 to integer'high; --range 0 to 2^31–1
```



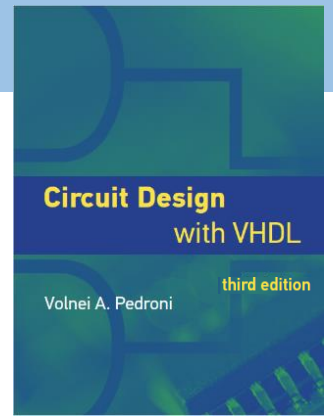
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

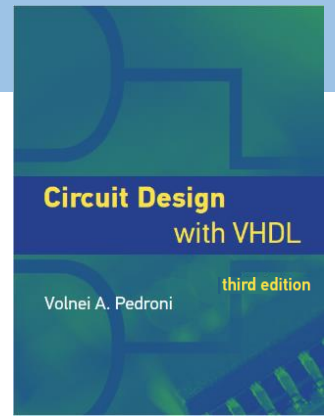


4. Record types



4. Record types

- Can have multiple members
- Elements can be of different types
- Useful to manipulate several objects together
- Particularly helpful in simulation (e.g. example 18.2)

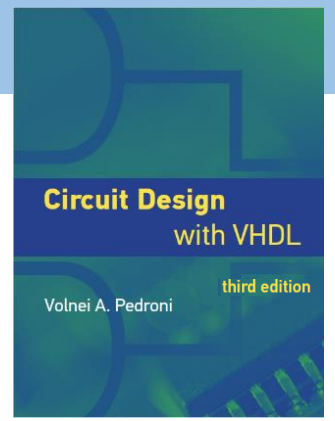


4. Record types

- Can have multiple members
- Elements can be of different types
- Useful to manipulate several objects together
- Particularly helpful in simulation (e.g. example 18.2)

Example:

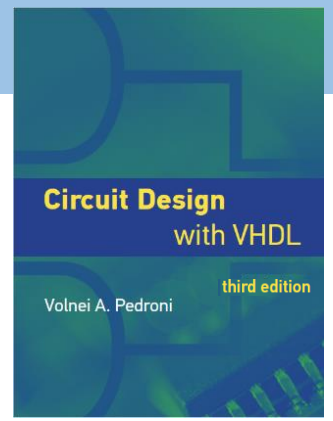
```
--Record type declaration:
type memory_access is record
   word_addr: natural range 0 to 4095;
   data: std_logic_vector(15 downto 0);
end record;

--Record type usage:
signal ram_interface: memory_access;
...
if ram_interface.word_addr=4095 then ...
```



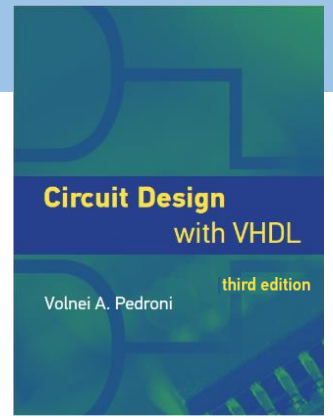
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

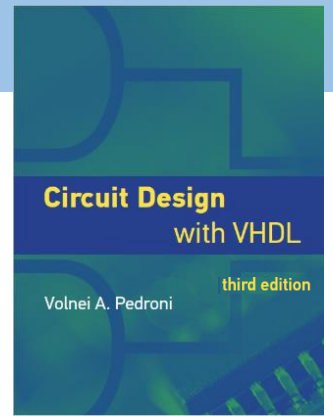


5. Description of predefined types, one-by-one



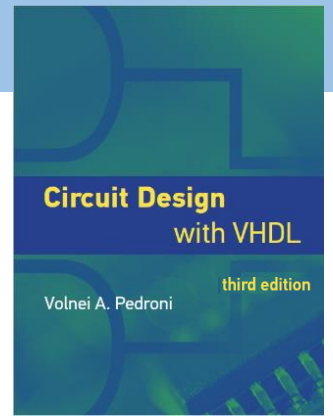
5. Description of predefined types, one-by-one

Category	Types	Abbreviations
Standard types	bit, bit_vector boolean, boolean_vector integer, natural, positive, integer_vector character, string	B, BV BO, BOV INT, NAT, POS, INTV CHAR, STR
Standard-logic types	std_ulogic, std_ulogic_vector std_logic, std_logic_vector	SU, SUV SL, SLV
Unsigned/signed types	unsigned signed	UNS SIG
Fixed-point types	ufixed sfixed	UFIX SFIX
Floating-point type	float	FLO



5. Description of predefined types, one-by-one

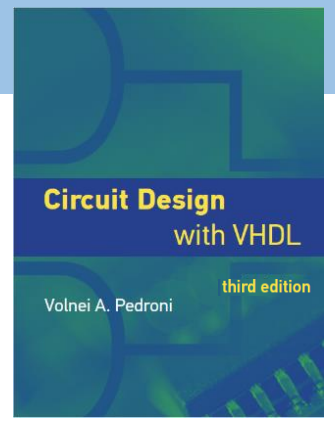
a) Standard types



5. Description of predefined types, one-by-one

a) Standard types

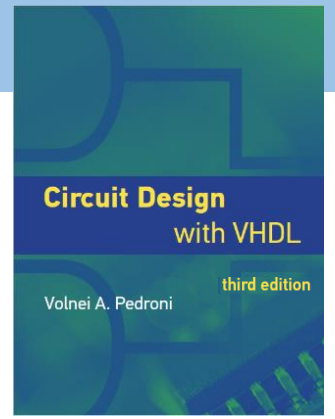
```
-----------------------------------------------
bit (2 values)          x <= '0'; y <= '1';
bit_vector              x <= "1101"; y <= b"101_0000";
-----------------------------------------------
```



5. Description of predefined types, one-by-one

a) Standard types

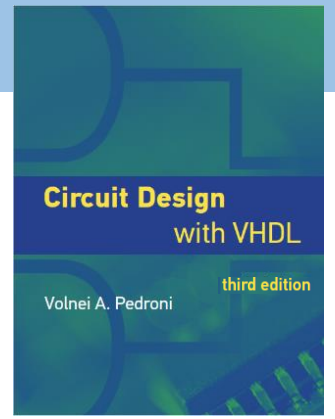
```
-----------------------------------------------
bit (2 values)          x <= '0'; y <= '1';
bit_vector              x <= "1101"; y <= b"101_0000";
-----------------------------------------------
boolean (2 values)      x <= false; y <= true;
boolean_vector          x <= (false, true, false);
-----------------------------------------------
```



5. Description of predefined types, one-by-one

a) Standard types

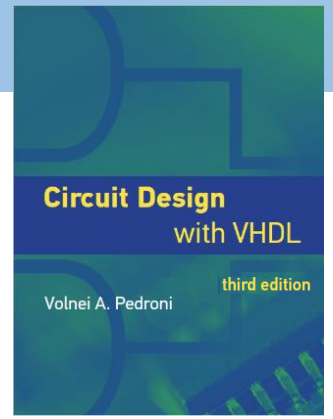
```
-----------------------------------------------
bit (2 values)          x <= '0'; y <= '1';
bit_vector              x <= "1101"; y <= b"101_0000";
-----------------------------------------------
boolean (2 values)      x <= false; y <= true;
boolean_vector          x <= (false, true, false);
-----------------------------------------------
integer (32 bits)       x <= -255; y <= 1111;
natural                 x <= 0;
positive                x <= 255; fclk <= 50_000_000;
integer_vector          x <= (0, 1, 2, -3);
-----------------------------------------------
```



5. Description of predefined types, one-by-one

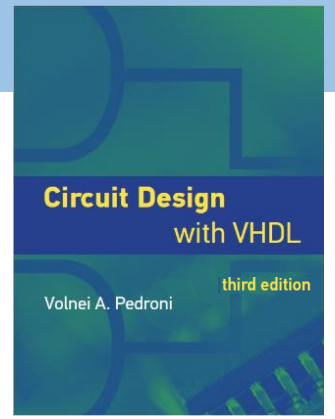
a) Standard types

```
-------------------------------------------------
bit (2 values)          x <= '0'; y <= '1';
bit_vector              x <= "1101"; y <= b"101_0000";
-------------------------------------------------
boolean (2 values)      x <= false; y <= true;
boolean_vector          x <= (false, true, false);
-------------------------------------------------
integer (32 bits)       x <= -255; y <= 1111;
natural                 x <= 0;
positive                x <= 255; fclk <= 50_000_000;
integer_vector          x <= (0, 1, 2, -3);
-------------------------------------------------
character (8 bits)      x <= 'a';
string                  x <= "VHDL";
-------------------------------------------------
```



5. Description of predefined types, one-by-one

b) Standard-logic types



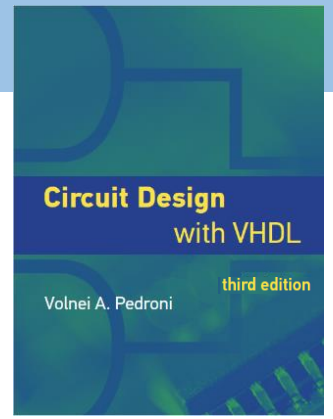
5. Description of predefined types, one-by-one

b) Standard-logic types

Type std_ulogic	
'U'	Uninitialized
'X'	Forcing unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High impedance
'W'	Weak unknown
'L'	Weak 0
'H'	Weak 1
'–'	Don't care

```
-----------------------------
std_ulogic (type)
std_logic (subtype)

std_ulogic_vector (type)
std_ulogic_vector (subtype)
-----------------------------
```

- Subtypes above = resolved versions of types above
- Resolution table: see section 7.6.2
- These are the most common types in the industry

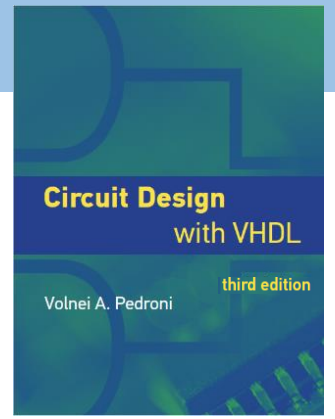


5. Description of predefined types, one-by-one

b) Standard-logic types

Examples with `std_ulogic` or `std_logic`:

```
x <= '0';
y <= '-';
z <= 'Z';
```



5. Description of predefined types, one-by-one

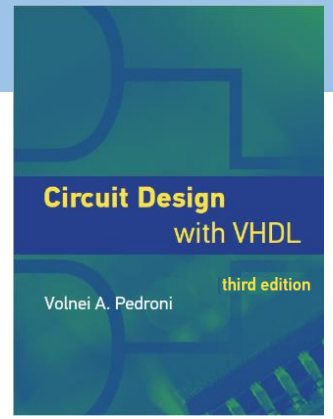
b) Standard-logic types

Examples with `std_ulogic` or `std_logic`:

```
x <= '0';
y <= '-';
z <= 'Z';
```

Examples with `std_ulogic_vector` or `std_logic_vector`:

```
v <= "1101";
x <= b"1101_----";
y <= "011-1-0";
z <= (others => 'Z');
```

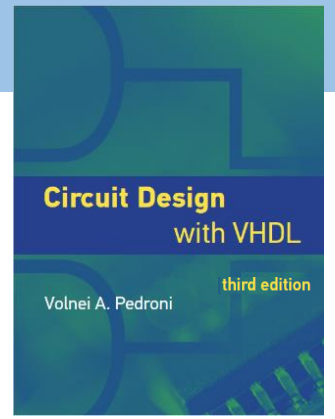


5. Description of predefined types, one-by-one

b) Standard-logic types

Synthesis tools:

```
'L' = '0'
'H' = '1'
'X' = '–'
```



5. Description of predefined types, one-by-one

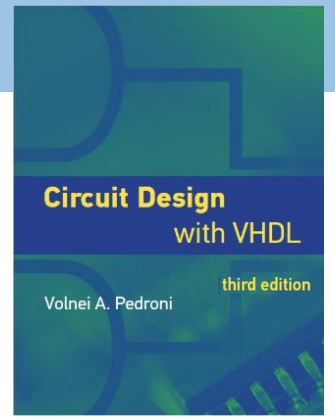
b) Standard-logic types

Synthesis tools:

```
'L' = '0'
'H' = '1'
'X' = '–'
```

Good design practice:

1) Use only `...???...` for inputs
2) Use only `...???...` for outputs
3) Use only `...???...` for arithmetic circuits (inputs and outputs)



5. Description of predefined types, one-by-one

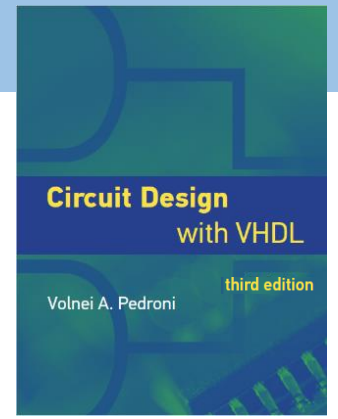
b) Standard-logic types

Synthesis tools:

```
'L' = '0'
'H' = '1'
'X' = '–'
```

Good design practice:

1) Use only `'0'`, `'1'`, `'–'` for inputs
2) Use only `'0'`, `'1'`, `'–'`, `'Z'` for outputs
3) Use only `'0'`, `'1'` for arithmetic circuits (inputs and outputs)



5. Description of predefined types, one-by-one

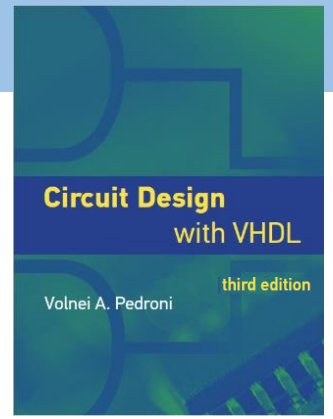
b) Standard-logic types

Synthesis tools:

```
'L' = '0'
'H' = '1'
'X' = '–'
```

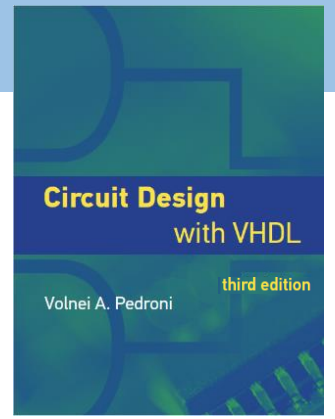
Good design practice:

1) Use only `'0'`, `'1'`, `'–'` for inputs
2) Use only `'0'`, `'1'`, `'–'`, `'Z'` for outputs
3) Use only `'0'`, `'1'` for arithmetic circuits (inputs and outputs)
4) But for arithmetic, use SUV/SLV only for ports; for computations, adopt:
   - For integers: UNS/SIG
   - For fixed-point: UFIX/SFIX
   - For floating-point: FLOAT



5. Description of predefined types, one-by-one

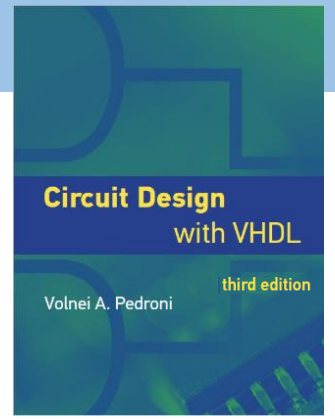
c) Unsigned and signed types



5. Description of predefined types, one-by-one

c) Unsigned and signed types

- `unsigned(L downto R)` or `unsigned(L to R)`
- `signed(L downto R)` or `signed(L to R)`
- Represent integers
- Based on `std_ulogic`
- `signed` employs two's complement for negative values



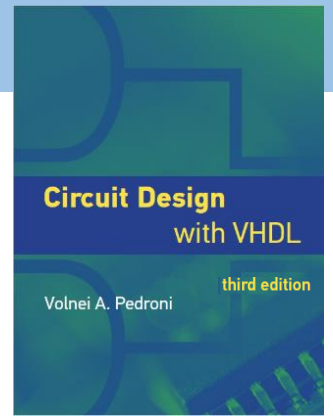
5. Description of predefined types, one-by-one

c) Unsigned and signed types

- `unsigned(L downto R)` or `unsigned(L to R)`
- `signed(L downto R)` or `signed(L to R)`
- Represent integers
- Based on `std_ulogic`
- `signed` employs two's complement for negative values

Recommendations:

- Use these types for implementing integer arithmetic circuits
- Use only values `'0'` and `'1'`
- Use only R=0

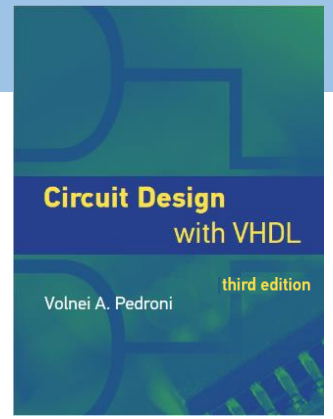


5. Description of predefined types, one-by-one

c) Unsigned and signed types

Examples:

```
signal x: unsigned(5 downto 0);
...
x <= "100000";          -- ?
x <= "10_0000";         -- ?
x <= b"10_0000";        -- ?
x <= (others => '1');   -- ?
```

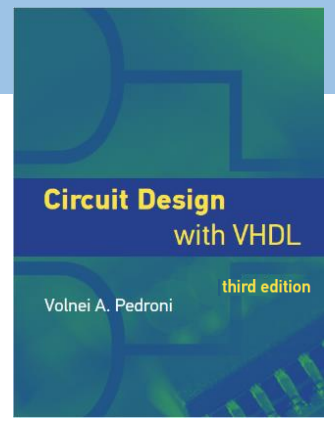


5. Description of predefined types, one-by-one

c) Unsigned and signed types

Examples:

```
signal x: unsigned(5 downto 0);
...
x <= "100000";          -- 32
x <= "10_0000";         -- illegal
x <= b"10_0000";        -- 32
x <= (others => '1');   -- 63
```

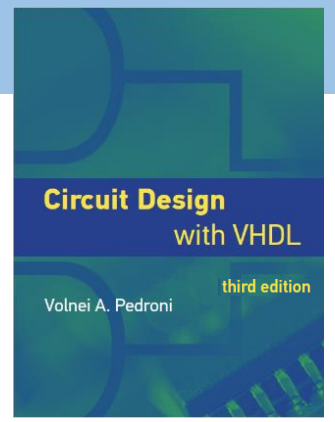


5. Description of predefined types, one-by-one

c) Unsigned and signed types

Examples:

```
signal x: unsigned(5 downto 0);
...
x <= "100000";          -- 32
x <= "10_0000";         -- illegal
x <= b"10_0000";        -- 32
x <= (others => '1');   -- 63

signal y: signed(5 downto 0);
...
y <= "100000";          -- ?
y <= b"10_0000";        -- ?
y <= (others => '1');   -- ?
```

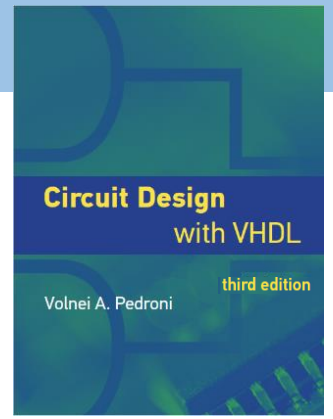


5. Description of predefined types, one-by-one

c) Unsigned and signed types

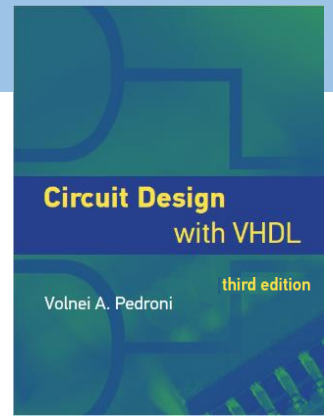
Examples:

```
signal x: unsigned(5 downto 0);
...
x <= "100000";          -- 32
x <= "10_0000";         -- illegal
x <= b"10_0000";        -- 32
x <= (others => '1');   -- 63

signal y: signed(5 downto 0);
...
y <= "100000";          -- -32
y <= b"10_0000";        -- -32
y <= (others => '1');   -- -1
```



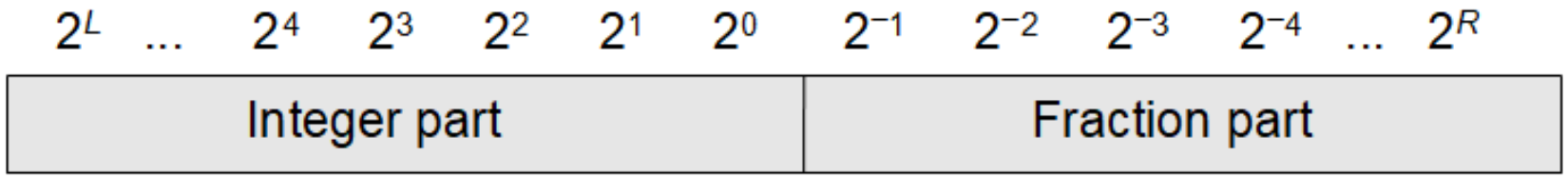
5. Description of predefined types, one-by-one

d) Fixed-point types

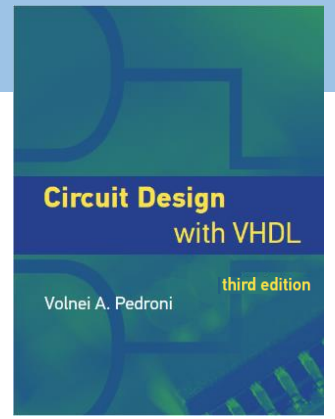


5. Description of predefined types, one-by-one

d) Fixed-point types

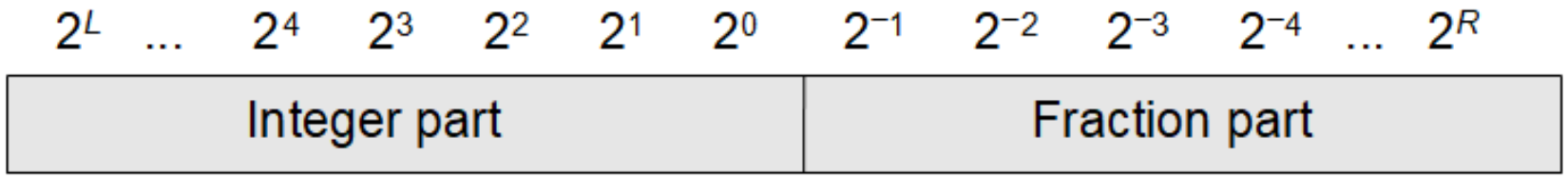


- `ufixed(L downto R)`
- `sfixed(L downto R)`
- Based on `std_ulogic`
- Employs two's complement for negative values



5. Description of predefined types, one-by-one

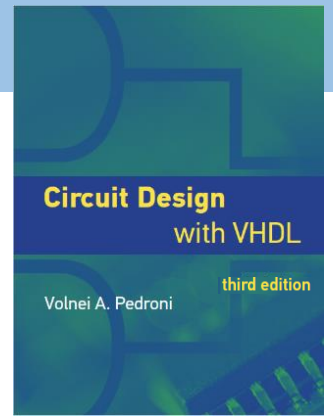
d) Fixed-point types



- `ufixed(L downto R)`
- `sfixed(L downto R)`
- Based on `std_ulogic`
- Employs two's complement for negative values

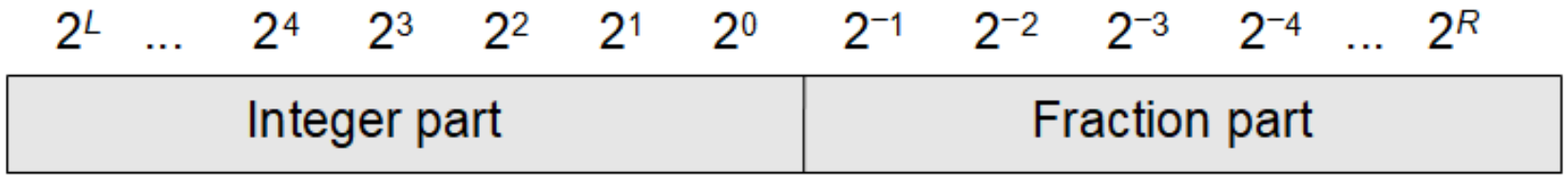
Recommendations:

- Use these types for implementing fixed-point arithmetic circuits
- Use only values `'0'` and `'1'`
- Remember that here L > R and usually R < 0

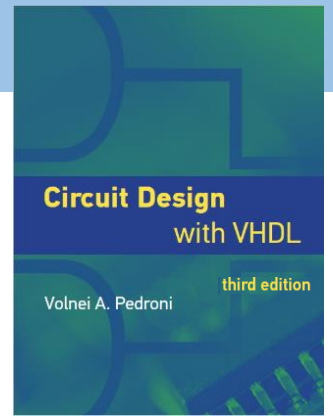


5. Description of predefined types, one-by-one

d) Fixed-point types

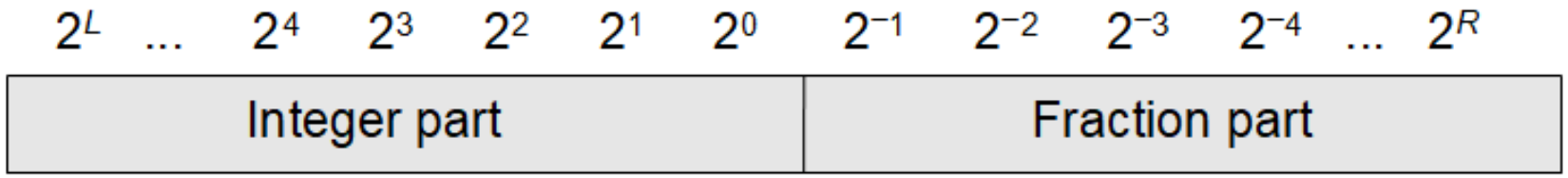


Examples (with L=3 and R=–2):



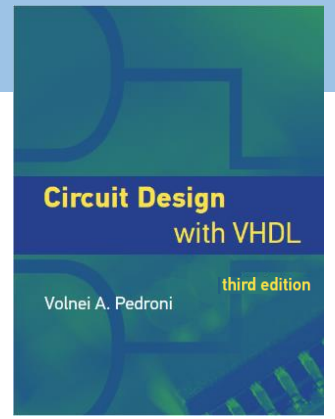
5. Description of predefined types, one-by-one

d) Fixed-point types



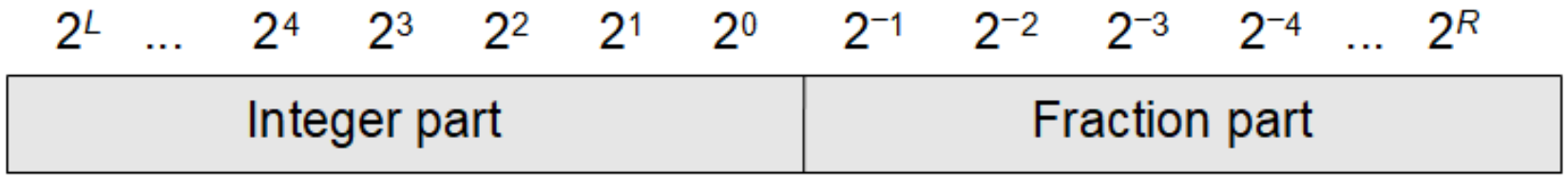
Examples (with L=3 and R=–2):

```
signal x: ufixed(3 downto –2);
x <= "100101";      -- 2³+2⁰+2⁻²=9.25 or uns/2^-R=37/4=9.25
x <= b"1001_01";    -- same
```



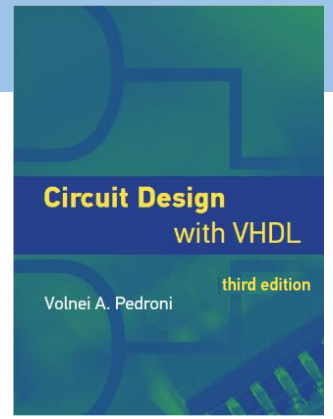
5. Description of predefined types, one-by-one

d) Fixed-point types



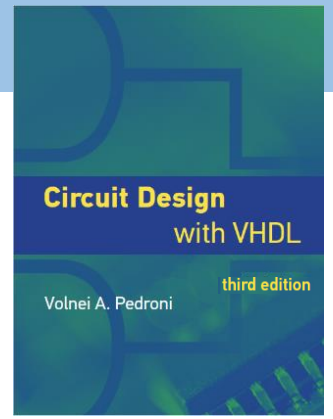
Examples (with L=3 and R=–2):

```
signal x: ufixed(3 downto –2);
x <= "100101";      -- 2^3+2^0+2^–2=9.25 or uns/2^-R=37/4=9.25
x <= b"1001_01";    -- same

signal y: sfixed(3 downto –2);
y <= "100101";      -- ufix–2^(L+1)=9.25–16=–6.75 or (uns-2^BITS)/2^-R=–6.75
y <= b"1001_01";    -- same
```



5. Description of predefined types, one-by-one

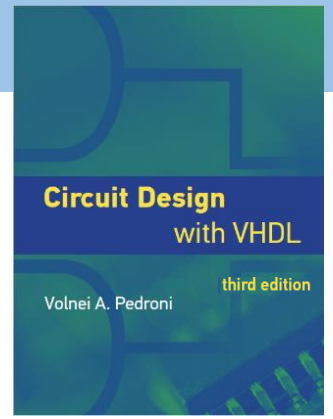
e) Floating-point types



5. Description of predefined types, one-by-one

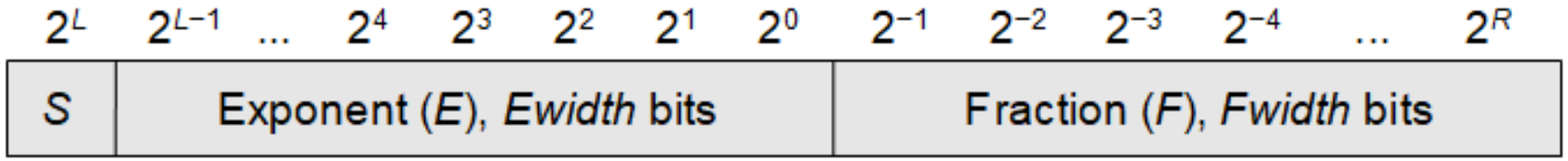
e) Floating-point types

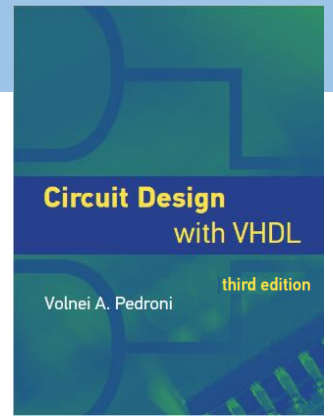
Review section 1.6.5 Floating-Point Arithmetic:
- IEEE standard
- Legal values
- Corresponding decimal value
- Rounding options
- Overflow, saturation, …



5. Description of predefined types, one-by-one

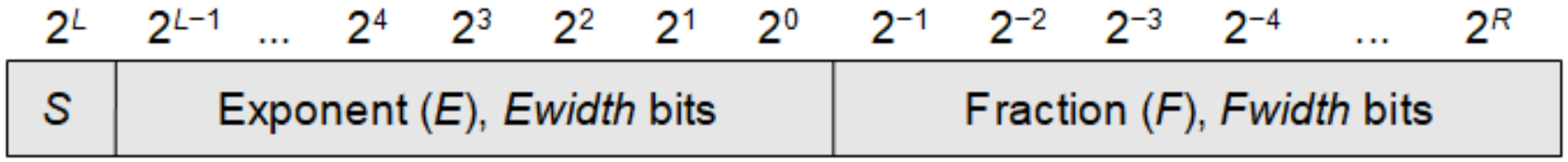
e) Floating-point types



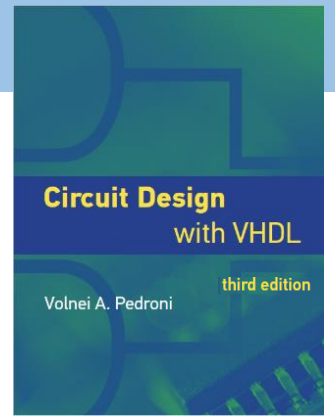


5. Description of predefined types, one-by-one

e) Floating-point types

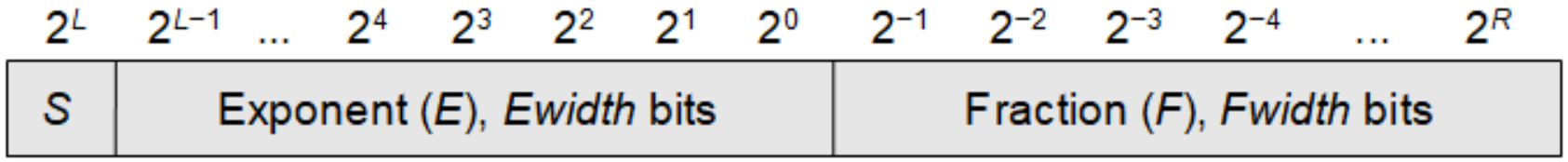


- `float32`
- `float64`
- `float128`
- `float(L downto R)`
- Based on `std_ulogic`
- Employs sign bit (no two's complement here)



5. Description of predefined types, one-by-one

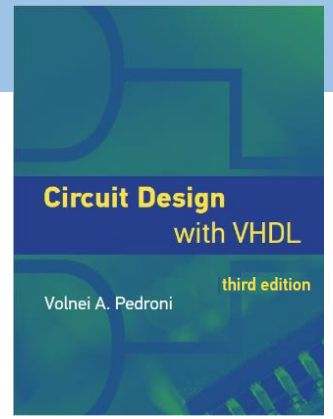
e) Floating-point types



- `float32`
- `float64`
- `float128`
- `float(L downto R)`
- Based on `std_ulogic`
- Employs sign bit (no two's complement here)

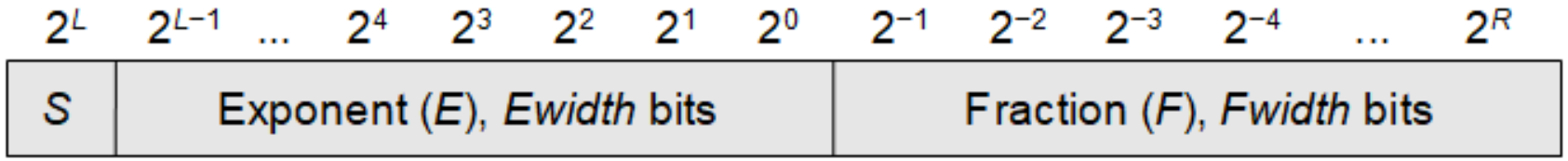
Recommendations:

- Use these types for implementing floating-point arithmetic circuits
- Use only values `'0'` and `'1'`
- Remember that here L > R and R < 0



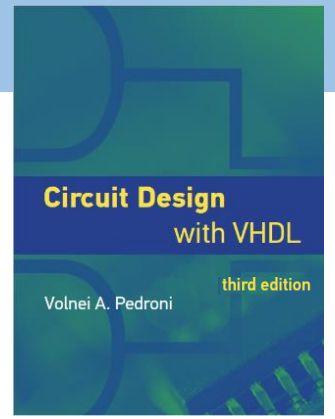
5. Description of predefined types, one-by-one

e) Floating-point types



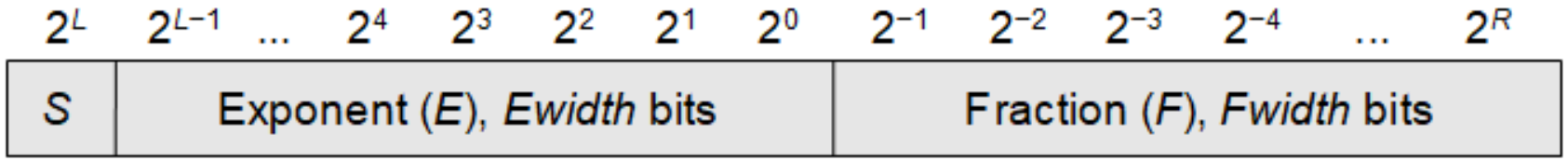
Decimal value: $dec = (-1)^S(1+F)2^{E-BIAS}$

Where: $BIAS = 2^{Ewidth-1} - 1$



5. Description of predefined types, one-by-one

e) Floating-point types



Decimal value: $dec = (-1)^S(1+F)2^{E-BIAS}$

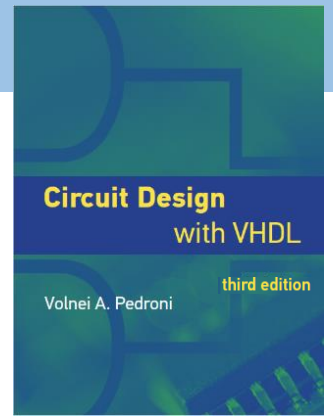
Where: $BIAS = 2^{Ewidth-1} - 1$

Example (with L=3 and R=–4):

```
signal x: float(3 downto –4);
x <= "10100110";
```

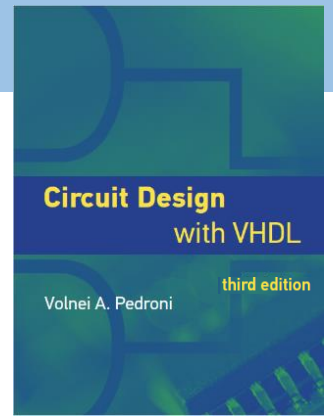
Decimal value of *x*:

$x = (S=1)(E=010)(F=0110) = -(1+0.375)2^{2-3} = -0.6875$



5. Description of predefined types, one-by-one

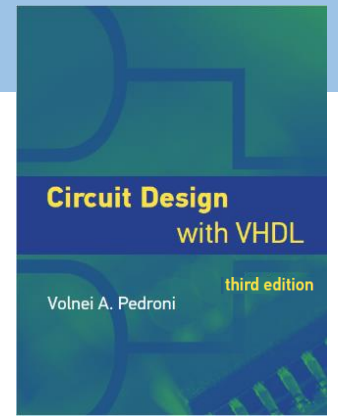
f) Type *real*



5. Description of predefined types, one-by-one

f) Type *real*

- Not synthesizable in general
- But can be used as part of expressions whose output is synthesizable
- Requires package *math_real*



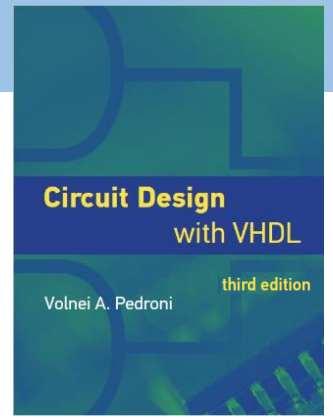
5. Description of predefined types, one-by-one

f) Type *real*

- Not synthesizable in general
- But can be used as part of expressions whose output is synthesizable
- Requires package *math_real*

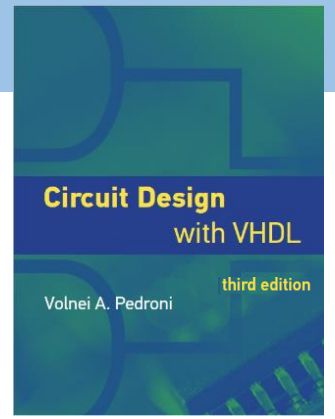
Example: Number of bits needed to represent an integer

```
library ieee;
use ieee.math_real.all;
...
signal count: natural range 0 to MAX;
constant NUM_BITS: natural := integer(ceil(log2(real(MAX+1))));
```



5. Description of predefined types, one-by-one

Closing example…

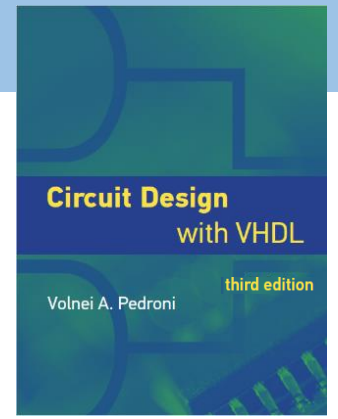


5. Description of predefined types, one-by-one

Closing example…

First, recall the main synthesizable types:

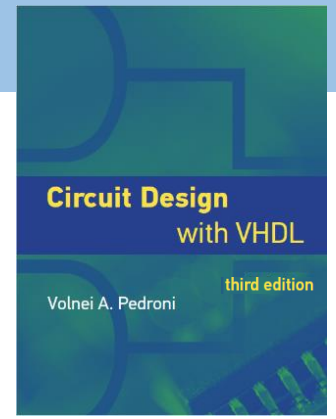
Category	Types	Abbreviations
Standard types	bit, bit_vector boolean, boolean_vector integer, natural, positive, integer_vector character, string	B, BV BO, BOV INT, NAT, POS, INTV CHAR, STR
Standard-logic types	std_ulogic, std_ulogic_vector std_logic, std_logic_vector	SU, SUV SL, SLV
Unsigned/signed types	unsigned signed	UNS SIG
Fixed-point types	ufixed sfixed	UFIX SFIX
Floating-point type	float	FLO



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

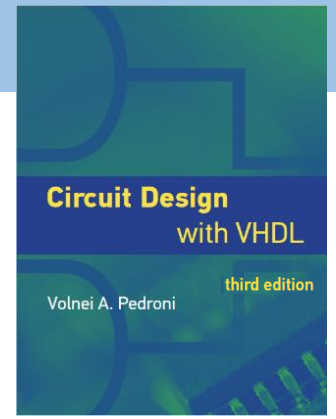
```
x1 <= '1';
x2 <= false;
x3 <= "false";
x4 <= "0-00000-";
x5 <= 011011;
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

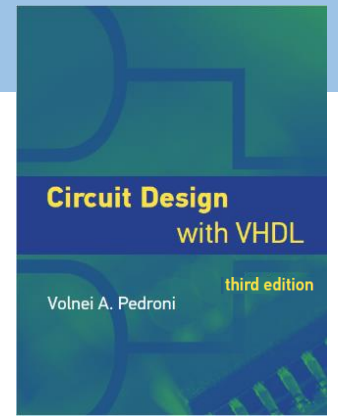
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false;
x3 <= "false";
x4 <= "0-00000-";
x5 <= 011011;
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

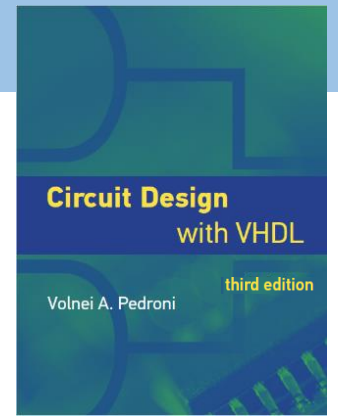
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false";
x4 <= "0-00000-";
x5 <= 011011;
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

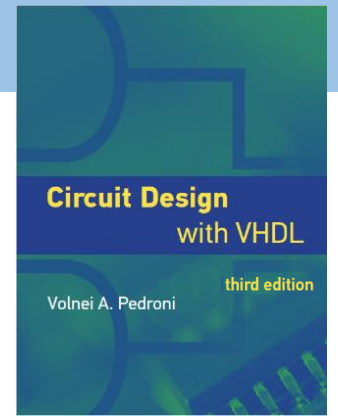
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-";
x5 <= 011011;
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

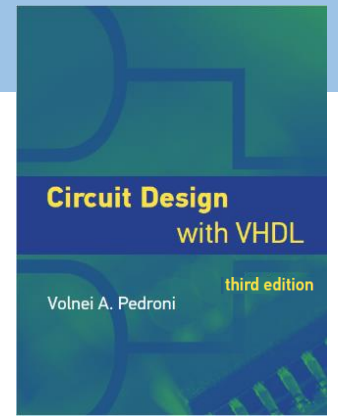
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011;
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

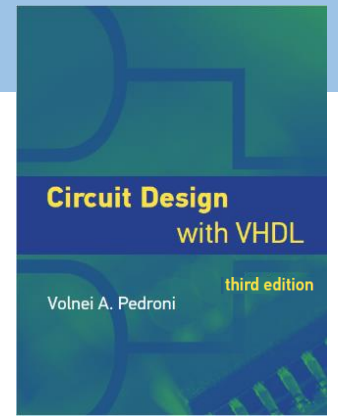
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000;
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

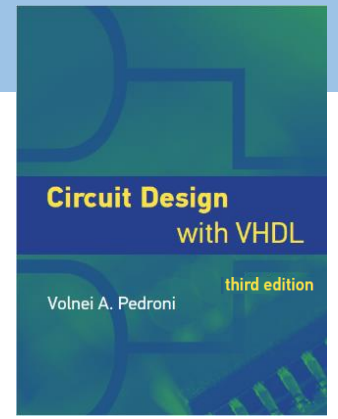
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z');
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

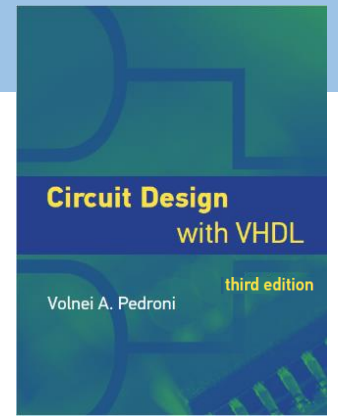
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0');
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

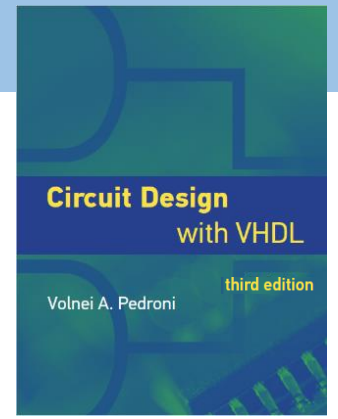
```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0'); --same as x4 plus BV
x9 <= "11" & "010";
```



5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0'); --same as x4 plus BV
x9 <= "11" & "010"; --same as x8
```

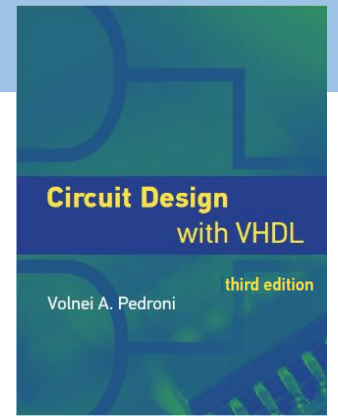


5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0'); --same as x4 plus BV
x9 <= "11" & "010"; --same as x8
```

Which options should be eliminated and why?

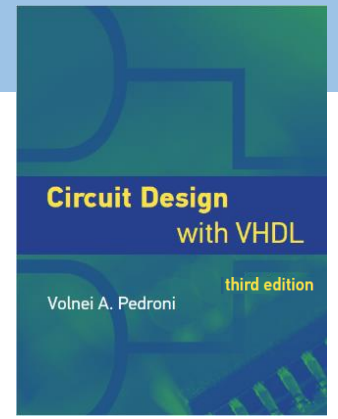


5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0'); --same as x4 plus BV
x9 <= "11" & "010"; --same as x8
```

Which options should be eliminated and why?

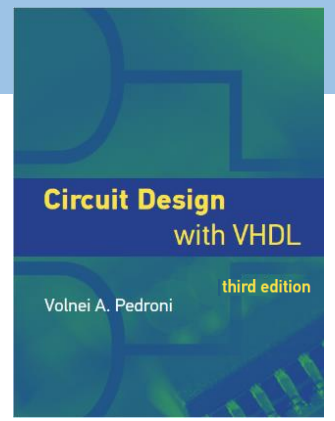


5. Description of predefined types, one-by-one

Closing example… Which are the possible types of the data below?

```
x1 <= '1'; --B, SU/SL, CHAR
x2 <= false; --BO
x3 <= "false"; --STR
x4 <= "0-00000-"; --STR, SUV/SLV, UNS/SIG, UFIX/SFIX, FLOAT
x5 <= 011011; --INT, NAT, POS
x6 <= 50_000_000; --same as x5
x7 <= (others => 'Z'); --same as x4
x8 <= ('1', '1', '0', '1', '0'); --same as x4 plus BV
x9 <= "11" & "010"; --same as x8
```

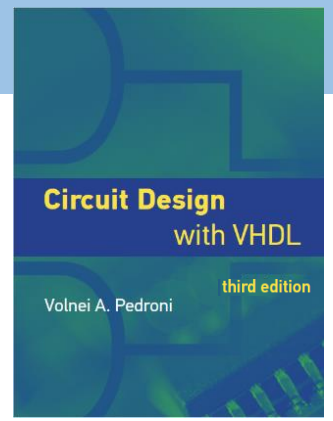
Which options should be eliminated and why? Use only '0' and '1' for arithmetic!



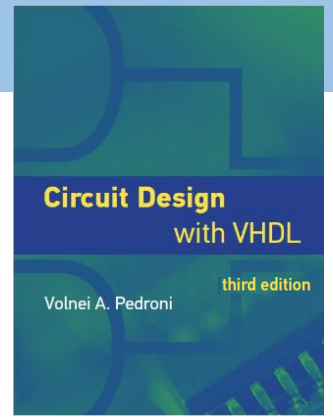
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

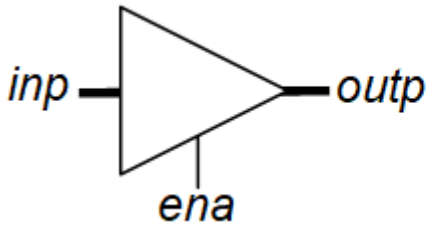


6. Application examples for *standard-logic* types

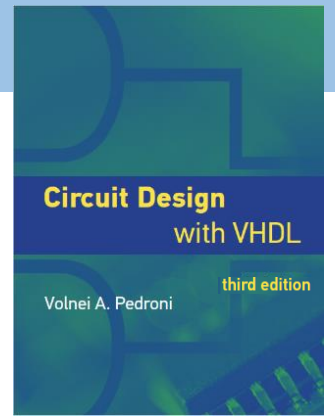


6. Application examples for *standard-logic* types

1) Why does this circuit need a standard-logic type?

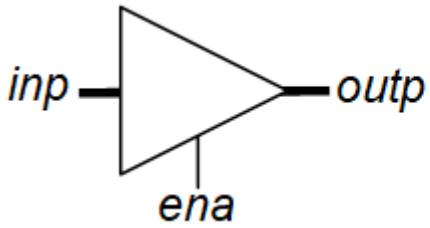


ena	*outp*
1	*inp*
0	ZZ…Z



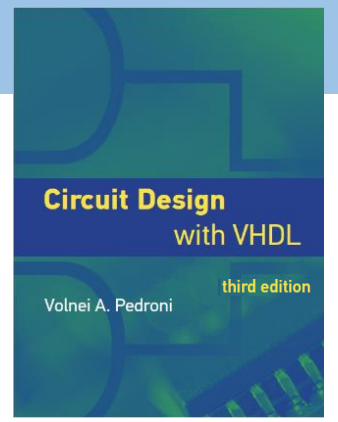
6. Application examples for *standard-logic* types

1) Why does this circuit need a standard-logic type?



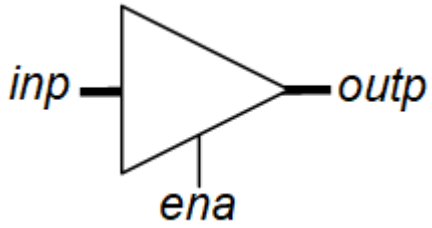
ena	*outp*
1	*inp*
0	ZZ…Z

Because of the high-impedance state (`'Z'`)

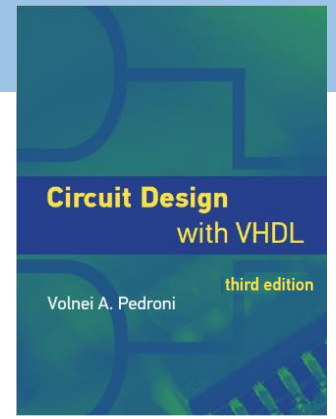


6. Application examples for *standard-logic* types

1) Why does this circuit need a standard-logic type?

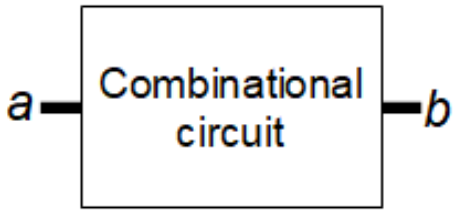



```
1  ------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity tri_state_buffer is
6     generic (
7        NUM_BITS: natural := 16);
8     port (
9        inp: in std_logic_vector(NUM_BITS-1 downto 0);
10       ena: in std_logic;
11       outp: out std_logic_vector(NUM_BITS-1 downto 0));
12 end entity;
13
14 architecture tri_state of tri_state_buffer is
15 begin
16    outp <= inp when ena else (others => 'Z');
17 end architecture;
18 ------------------------------------------------
```

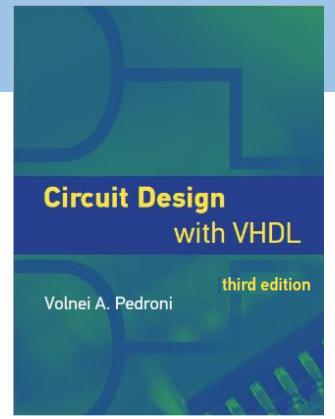


6. Application examples for *standard-logic* types

2) Why does this circuit need a standard-logic type?

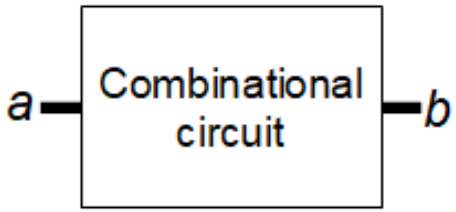


$a_2\ a_1\ a_0$	$b_1\ b_0$
0 0 0	– 0
0 0 1	1 0
0 1 0	– 1
0 1 1	– –
1 0 0	0 –
1 0 1	0 0
1 1 0	0 –
1 1 1	0 1



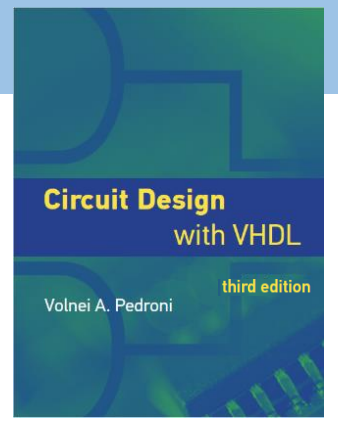
6. Application examples for *standard-logic* types

2) Why does this circuit need a standard-logic type?



$a_2\ a_1\ a_0$	$b_1\ b_0$
0 0 0	– 0
0 0 1	1 0
0 1 0	– 1
0 1 1	– –
1 0 0	0 –
1 0 1	0 0
1 1 0	0 –
1 1 1	0 1

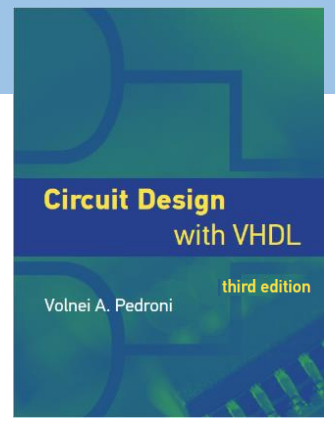
Because of the don't care state ('–')



6. Application examples for *standard-logic* types

2) Why does this circuit need a standard-logic type?

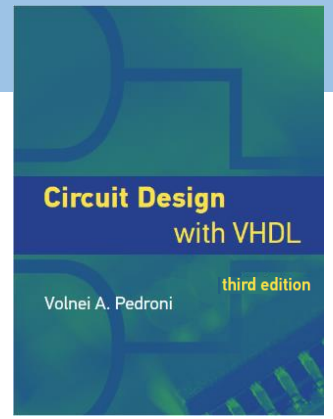
```
1  ----------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity circuit_with_dontcare is
6     port (
7        a: in std_logic_vector(2 downto 0);
8        b: out std_logic_vector(1 downto 0));
9  end entity;
10
11 architecture truth_table of circuit_with_dontcare is
12 begin
13    with a select
14       b <= "-0" when "000",
15            "10" when "001",
16            "-1" when "010",
17            "--" when "011",
18            "0-" when "100" | "110",
19            "00" when "101",
20            "01" when others;
21 end architecture;
22 ----------------------------------------------------
```



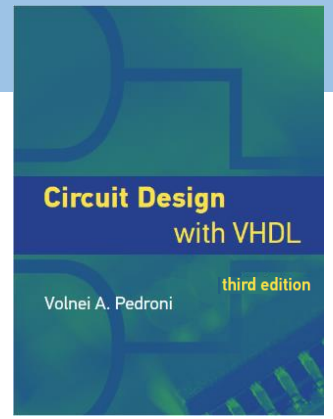
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions



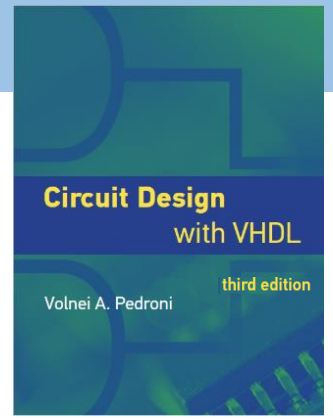
7. Type conversion



7. Type conversion

Three options:

- Automatic Conversion
- Type Cast
- Type-Conversion Functions



7. Type conversion

Three options:

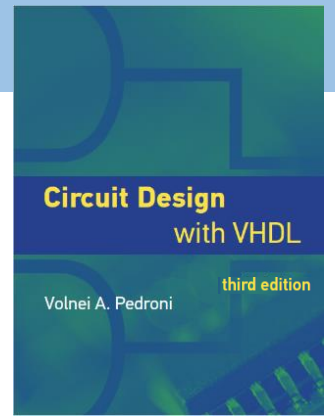
- Automatic Conversion
- Type Cast
- Type-Conversion Functions

a) Automatic Conversion

Occurs when dealing directly with the base type

Examples:

```
bv(0) <= b;      --single element of type bit on both sides
sl <= slv(7);    --single element of type std_logic on both sides
```



7. Type conversion

Three options:

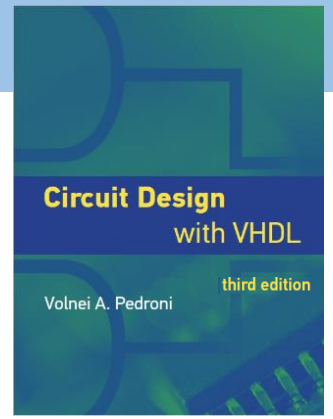
- Automatic Conversion
- Type Cast
- Type-Conversion Functions

b) Type cast

- Between `integer` and `real`
- Between `std_(u)logic_vector`, `unsigned`, and `signed`

Examples:

```
slv <= std_logic_vector(uns); --type cast from UNS to SLV
uns <= unsigned(slv);         --type cast from SLV to UNS
sig <= signed(uns);           --type cast from UNS to SIG
int <= integer(re);           --type cast from RE to INT
```



7. Type conversion

Three options:

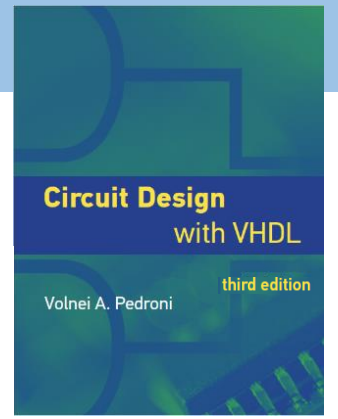
- Automatic Conversion
- Type Cast
- Type-Conversion Functions

c) With a type-conversion function

- See table next

Examples:

```
uns <= to_unsigned(int, 8);      --INT to UNS, with 8 bits
int <= to_integer(uns);           --UNS to INT, with same # of bits
sfix <= to_sfixed(int, 3, -4);   --INT to SFIX, with L=3 and R=-4
```



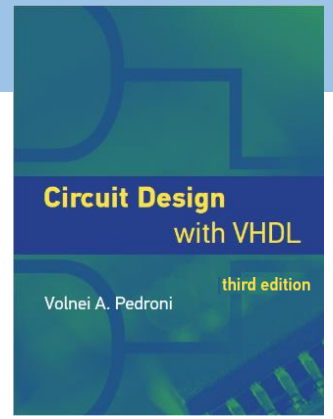
7. Type conversion

Table 7.10

Predefined type-conversion functions for synthesizable types and subtypes

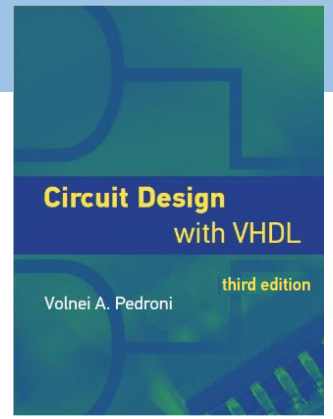
From type	To type	Type-conversion function		Package of origin
integer, natural, positive	std_ulogic_vector	1	*to_std_ulogic_vector(arg, width)*	*numeric_std_unsigned*
	std_logic_vector	2	*to_std_logic_vector(arg, width)*	*numeric_std_unsigned*
	unsigned	3	*to_unsigned(arg, width)*	*numeric_std*
	signed	4	*to_signed(arg, width)*	*numeric_std*
	ufixed	5	*to_ufixed(arg, L, R)*	*fixed_generic_pkg*
	sfixed	6	*to_sfixed(arg, L, R)*	*fixed_generic_pkg*
	float	7	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*
	real	8	Type cast: *real(arg)*	*math_real*
bit	std_ulogic, std_logic	9	*to_stdulogic(arg)*	*std_logic_1164*
bit_vector	std_ulogic_vector	10	*to_stdulogicvector(arg)*	*std_logic_1164*
	std_logic_vector	11	*to_stdlogicvector(arg)*	*std_logic_1164*
std_ulogic,	bit	12	*to_bit(arg)*	*std_logic_1164*

See complete table in section 7.10.3



7. Type conversion

A special case: How to convert INT ↔ SLV/SUV?

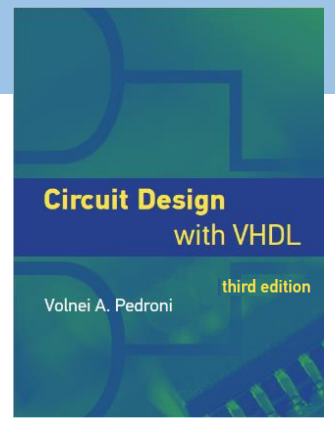


7. Type conversion

A special case: How to convert INT ↔ SLV/SUV?

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
slv <= std_logic_vector(to_signed(int, 8));   --from INT to SLV
int <= to_integer(signed(slv));               --from SLV to INT
```

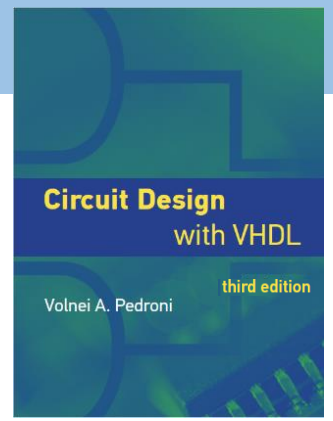
(assumed that package *numeric_std_unsigned* is not supported)



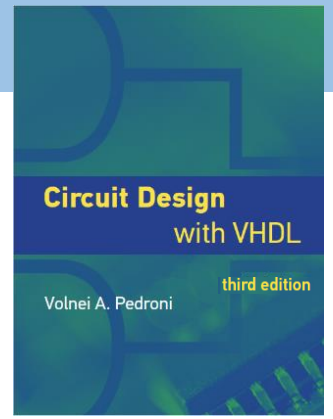
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions

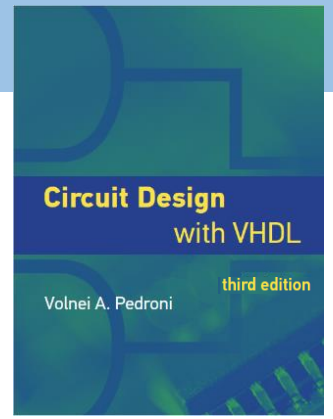


8. Aggregation, concatenation, and resizing



8. Aggregation, concatenation, and resizing

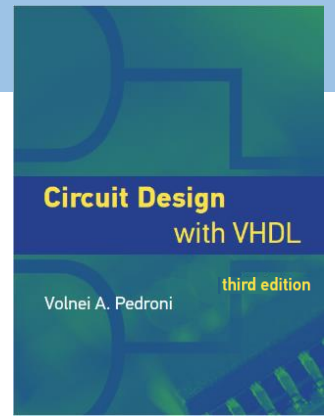
a) Aggregation



8. Aggregation, concatenation, and resizing

a) Aggregation

- Single-values, written between parentheses
- Separated by comma
- Keyword `others` often helpful
- Multi-value pieces allowed in VHDL 2008



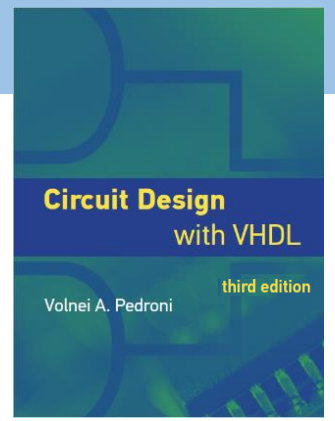
8. Aggregation, concatenation, and resizing

a) Aggregation

- Single-values, written between parentheses
- Separated by comma
- Keyword *others* often helpful
- Multi-value pieces allowed in VHDL 2008

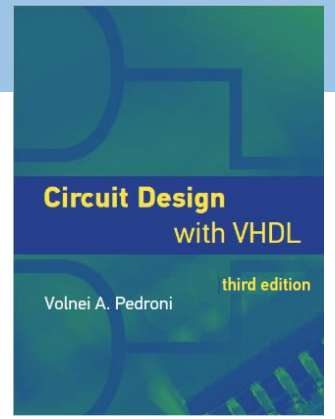
Examples (with *positional* and *named* associations):

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(5 downto 0);
...
s1 <= ('1', '0', '0', '0', '1', '0');          --s1="100010"
s2 <= ('1', '0', '0', others => 'Z');          --s2="100ZZZ"
s3 <= ('0', '-', others => s1(5));             --s3="0-1111"
s4 <= (1 => '1', 5 downto 2 => '0', 0 =>'-');  --s4="00001-"
s5 <= (5|1 => '1', others => '0');             --s5="100010"
s6 <= ('1', "000", "11");                      --s6="100011" (VHDL 2008)
```



8. Aggregation, concatenation, and resizing

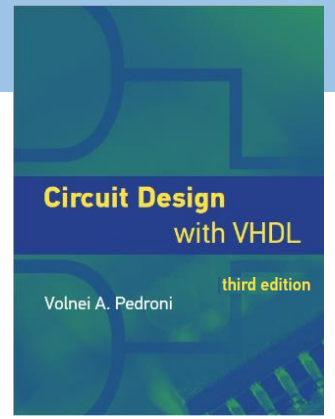
b) Concatenation



8. Aggregation, concatenation, and resizing

b) Concatenation

- Uses "&" operator
- Parentheses are optional (might improve readability)
- Keyword *others* not allowed
- Concatenation of aggregates is legal



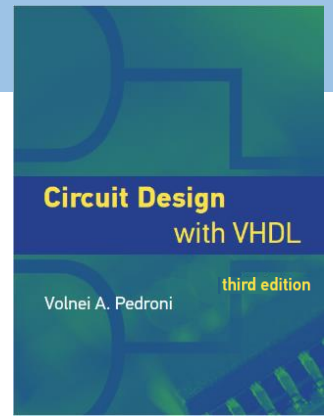
8. Aggregation, concatenation, and resizing

b) Concatenation

- Uses "&" operator
- Parentheses are optional (might improve readability)
- Keyword *others* not allowed
- Concatenation of aggregates is legal

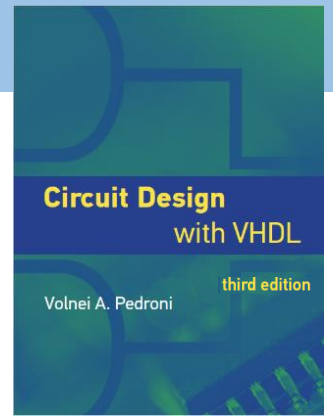
Examples:

```
signal s1, s2, s3: std_logic_vector(5 to 0);
...
s1 <= "11" & "0000";                          --s1="110000"
s2 <= '1' & s1(0) & s1(5 downto 4) & "00";    --s2="101100"
s3 <= s1(5) & s1(5) & ('1', '0', '1', '0');   --s3="111010"
```



8. Aggregation, concatenation, and resizing

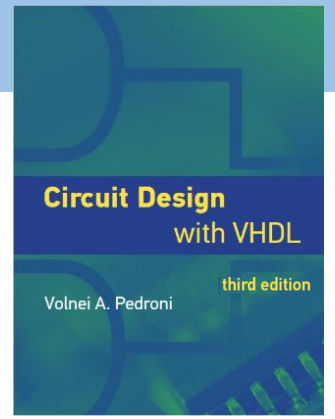
c) The *resize* function



8. Aggregation, concatenation, and resizing

c) The *resize* function

- VHDL-2008: Resize includes BV, SUV/SLV, UNS/SIG, UFIX/SFIX, FLO
- Full details shown in section 7.9.3
- Except for UNS/SIG, compilation support might still be limited

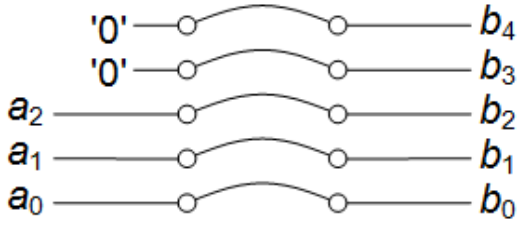


8. Aggregation, concatenation, and resizing

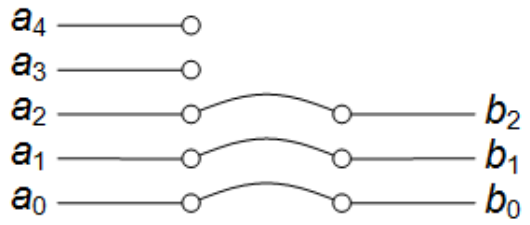
c) The *resize* function

- VHDL-2008: Resize includes BV, SUV/SLV, UNS/SIG, UFIX/SFIX, FLO
- Full details shown in section 7.9.3
- Except for UNS/SIG, compilation support might still be limited

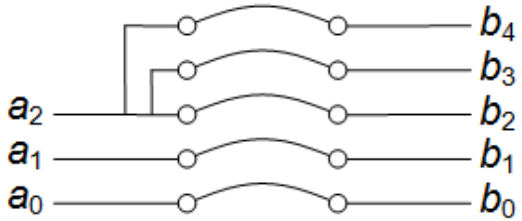
The case of UNS/SIG:



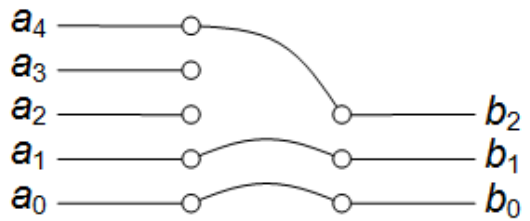
3:5 unsigned extension



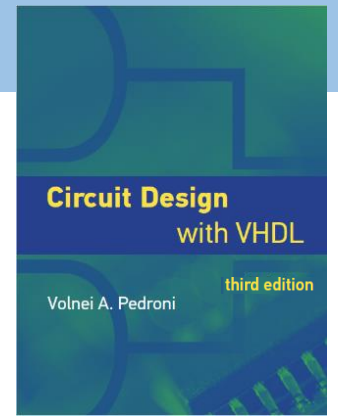
5:3 unsigned reduction



3:5 signed extension



5:3 signed reduction



8. Aggregation, concatenation, and resizing

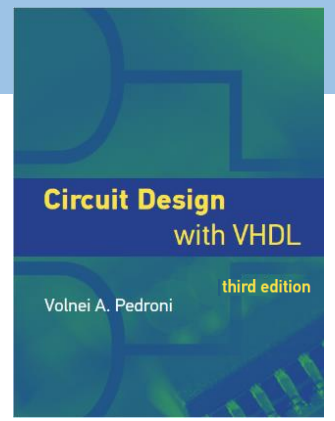
c) The *resize* function

- VHDL-2008: Resize includes BV, SUV/SLV, UNS/SIG, UFIX/SFIX, FLO
- Full details shown in section 7.9.3
- Except for UNS/SIG, compilation support might still be limited

The case of UNS/SIG:

Example (for SIG):

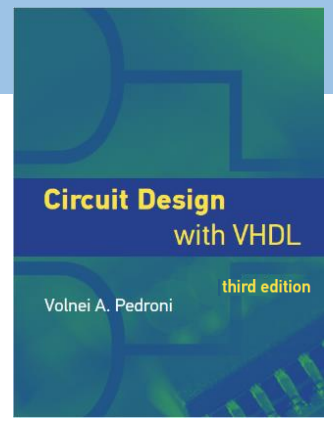
```
signal s1: signed(7 downto 0);  --8 bits
signal s2: signed(5 downto 0);  --6 bits
...
s1 <= resize(s2, 8);     --result: s1 = s2(5) & s2(5) & s2
s2 <= resize(s1, 6);     --result: s2 = s1(7) & s1(4 downto 0)
```



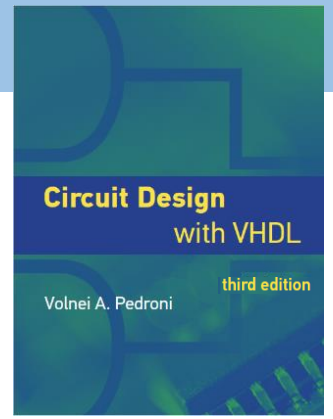
Chapter 7

Predefined Data Types

1. Main *synthesizable* types
2. Type declarations (how types are created)
3. Subtypes
4. Record types
5. Description of predefined types, one-by-one
6. Application examples for *standard-logic* types
7. Type conversion
8. Aggregation, concatenation, and resizing
9. Type-qualification expressions



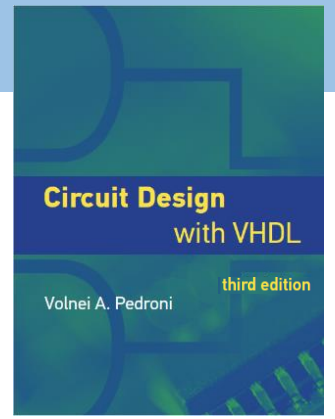
9. Type-qualification expressions



9. Type-qualification expressions

- Help compiler resolve ambiguities about type of an object
- Require the ' (tick) symbol, as shown below

```
type_name'(expression);
```



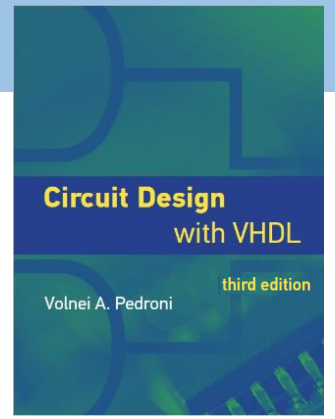
9. Type-qualification expressions

- Help compiler resolve ambiguities about type of an object
- Require the ' (tick) symbol, as shown below

```
type_name'(expression);
```

Example:

```
signal x, y: integer range ...;
...
y <= x when x > "1001" else 0;           --illegal (type of "1001" undetermined)
y <= x when x > signed'("10001") else 0; --legal
```



9. Type-qualification expressions

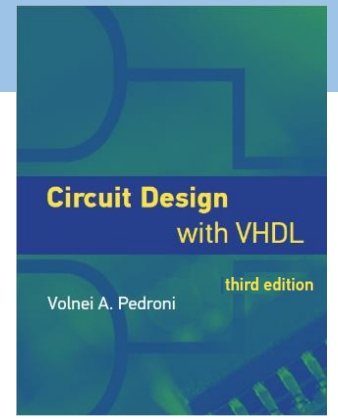
- Help compiler resolve ambiguities about type of an object
- Require the ' (tick) symbol, as shown below

```
type_name'(expression);
```

Example:

```
signal x, y: integer range ...;
...
y <= x when x > "1001" else 0;          --illegal (type of "1001" undetermined)
y <= x when x > signed'("10001") else 0; --legal
```

How do we know the 1st expression above is illegal?



9. Type-qualification expressions

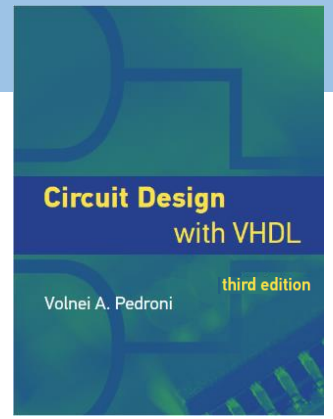
- Help compiler resolve ambiguities about type of an object
- Require the ' (tick) symbol, as shown below

```
type_name'(expression);
```

Example:

```
signal x, y: integer range ...;
...
y <= x when x > "1001" else 0;          --illegal (type of "1001" undetermined)
y <= x when x > signed'("10001") else 0; --legal
```

How do we know the 1st expression above is illegal?
- It involves a comparison operator (>)
- So we check table 9.5, where we see that INT can be compared to INT, SIG, SFIX, FLO, ...
- Therefore, the type of "10001" cannot be determined unambiguously



End of Chapter 7