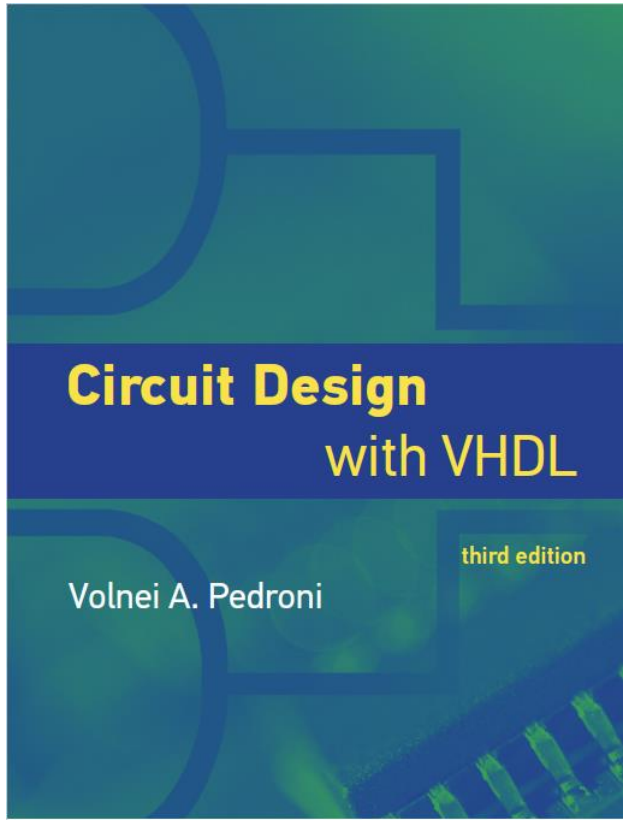


Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 8

User-Defined Data Types

Revision 1

Book Contents

Part I: Digital Circuits Review

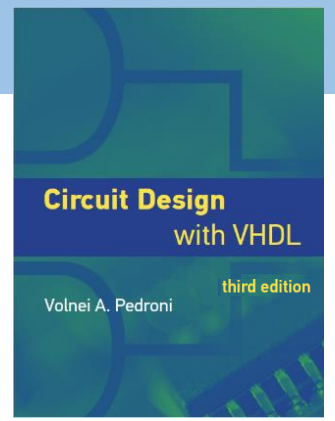
1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

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

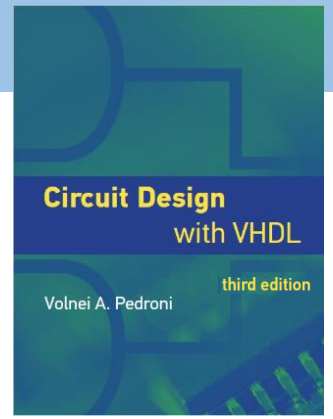
Appendices

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

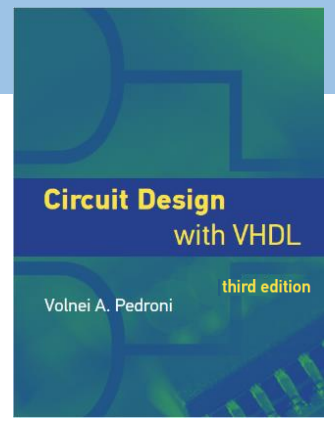
Chapter	Title
5	Introduction to VHDL
6	Code Structure and Composition
7	Predefined Data Types
8	User-Defined Data Types
9	Operators and Attributes
10	Concurrent Code
11	Concurrent Code – Practice
12	Sequential Code
13	Sequential Code – Practice
14	Packages and Subprograms



Chapter 8

User-Defined Data Types

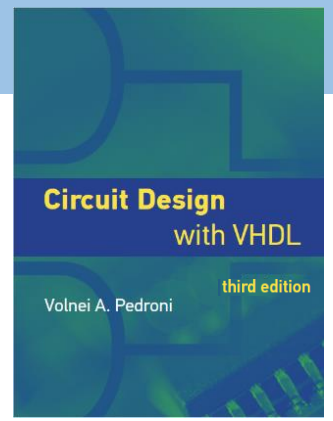
1. User-defined types
2. Array dimensionality
3. Building and addressing complex arrays
4. Checking and resetting data arrays
5. Classical mistakes in assignments



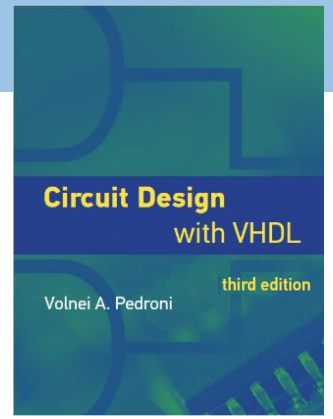
Seen so far…

Table 8.1. Representation of synthesizable predefined types.

Type class	Subclass	Type or subtype	Representation examples
Scalar types	Integer types	integer	-35 (≤32 bits)
		natural	0 (≤31 bits)
		positive	255 (≤31 bits)
	Enumeration types	boolean	true (1 bit)
		bit std_(u)logic	'1' (1 bit)
		character	'A' (8 bits)
Composite types	Array types	integer_vector	63 | -9 | 0 | 127
		boolean_vector	true | false | false | true
		bit_vector std_(u)logic_vector unsigned/signed ufixed/sfixed float	'1' | '0' | '0' | '0'
		string	'V' | 'H' | 'D' | 'L'

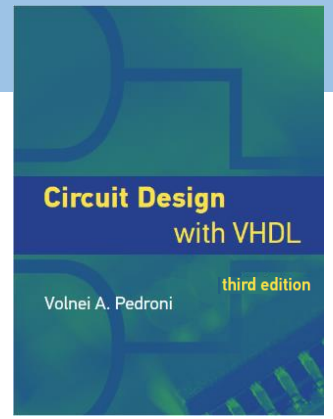


1. User-defined types



1. User-defined types

a) Integer types
b) Enumeration types
c) Array types

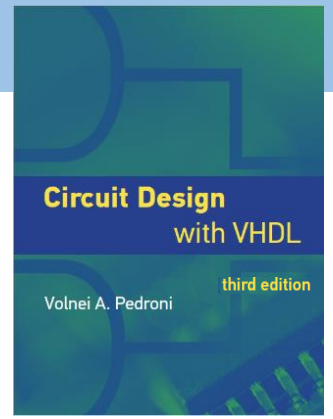


1. User-defined types

a) Integer types
b) Enumeration types
c) Array types

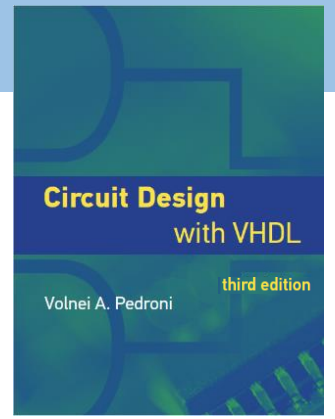
Notes on built-in types:

- Are those from the package *standard*
- For example, a user-defined type based on `integer` or `bit` or `boolean` inherits the base-type properties (i.e., operators and other functions)
- But they are still considered to be different types



1. User-defined types

a) Integer types

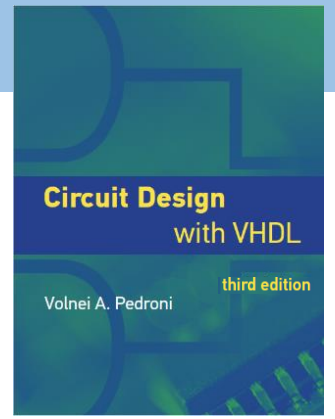


1. User-defined types

a) Integer types

- Recall that `integer` is a built-in type
- Integer types are declared (created) as shown below:

```
type type_name is range range_specification;
```



1. User-defined types

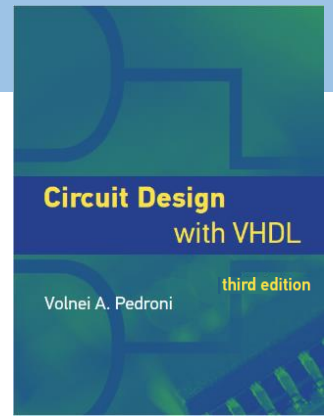
a) Integer types

- Recall that `integer` is a built-in type
- Integer types are declared (created) as shown below:

```
type type_name is range range_specification;
```

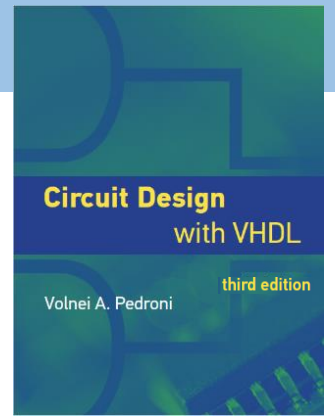
Example:

```
type small_integer is range –32 to 31;
signal s1, s2: integer range –32 to 31;   --with predefined integer
signal s3, s4: small_integer;             --with user-defined integer above
...
s1 <= s2**2 + 5;   --legal (same type and with support for arith. operators)
s3 <= s4**2 + 5;   --legal (same as above)
s1 <= s3;          --illegal (type mismatch)
```



1. User-defined types

b) Enumeration types

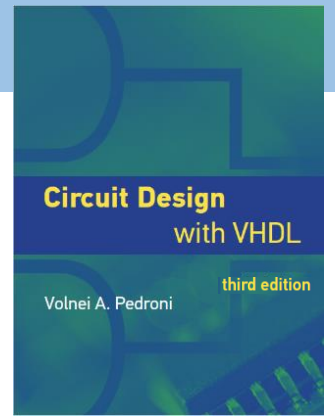


1. User-defined types

b) Enumeration types

- A list of named values (a list of "symbols")
- Very useful for finite state machines (chapters 15-16)
- They are declared (created) as shown below:

```
type type_name (type_values_list);
```



1. User-defined types

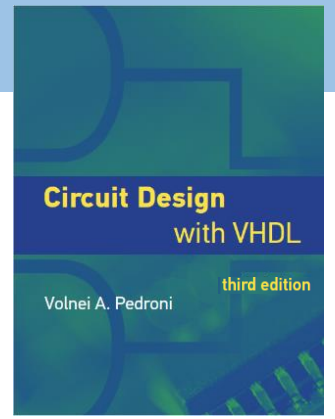
b) Enumeration types

- A list of named values (a list of "symbols")
- Very useful for finite state machines (chapters 15-16)
- They are declared (created) as shown below:

```
type type_name (type_values_list);
```

Examples showing predefined types:

```
type boolean is (false, true); --predefined type boolean
type bit is ('0', '1');        --predefined type bit
```



1. User-defined types

b) Enumeration types

- A list of named values (a list of "symbols")
- Very useful for finite state machines (chapters 15-16)
- They are declared (created) as shown below:

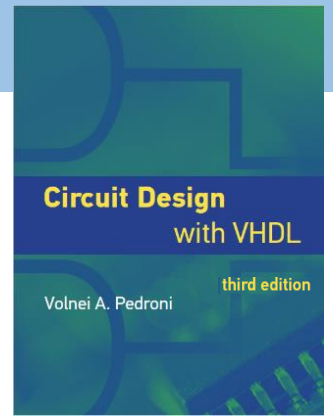
```
type type_name (type_values_list);
```

Examples showing predefined types:

```
type boolean is (false, true); --predefined type boolean
type bit is ('0', '1');        --predefined type bit
```

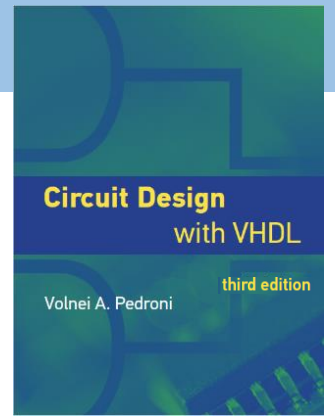
Example of user-defined type:

```
type state_type is (hold, read, add, shift, store);
signal pr_state, nx_state: state_type;
```



1. User-defined types

c) Array types

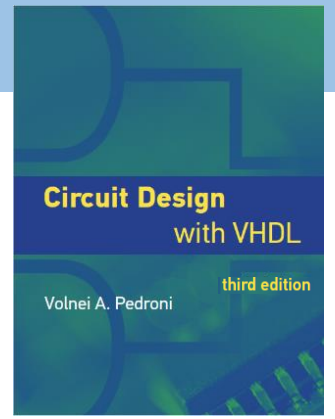


1. User-defined types

c) Array types

- Recall that `bit`, `boolean`, `integer`, … are built-in types
- So vectors of these types inherit the base-type properties (operators, …)
- But they are still different types
- Array types are declared (created) as shown below:

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



1. User-defined types

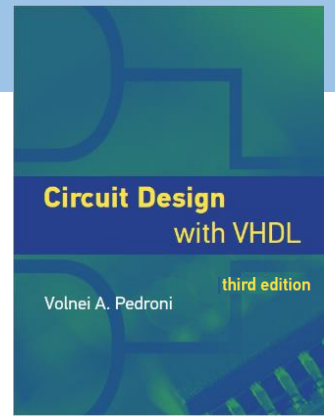
c) Array types

- Recall that `bit`, `boolean`, `integer`, … are built-in types
- So vectors of these types inherit the base-type properties (operators, …)
- But they are still different types
- Array types are declared (created) as shown below:

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

Examples showing predefined array types:

```
type integer_vector is array (natural range <>) of integer;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```



1. User-defined types

c) Array types

- Recall that `bit`, `boolean`, `integer`, … are built-in types
- So vectors of these types inherit the base-type properties (operators, …)
- But they are still different types
- Array types are declared (created) as shown below:

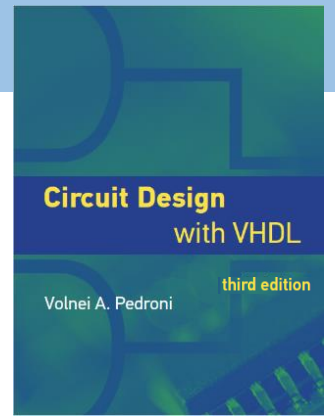
```
type type_name is array (range_spec) of base_type_name [range_spec];
```

Examples showing predefined array types:

```
type integer_vector is array (natural range <>) of integer;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```

Examples of user-defined array types:

```
type int_vec is array (natural range <>) of integer range -32 to 31;  -- ?-bit integers
type bv_array is array (0 to 255) of bit_vector;  -- ? values of size ?
```



1. User-defined types

c) Array types

- Recall that `bit`, `boolean`, `integer`, … are built-in types
- So vectors of these types inherit the base-type properties (operators, …)
- But they are still different types
- Array types are declared (created) as shown below:

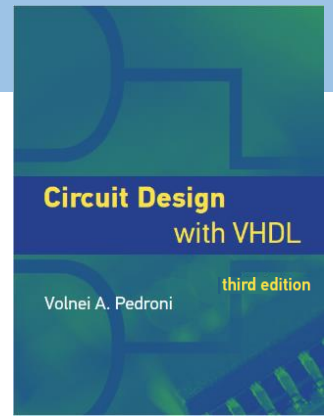
```
type type_name is array (range_spec) of base_type_name [range_spec];
```

Examples showing predefined array types:

```
type integer_vector is array (natural range <>) of integer;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```

Examples of user-defined array types:

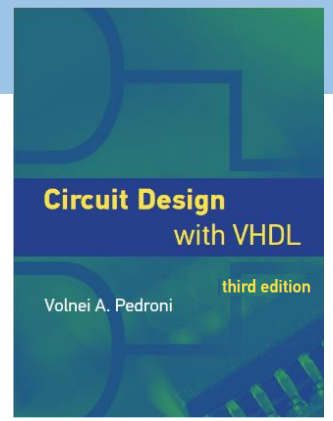
```
type int_vec is array (natural range <>) of integer range -32 to 31;  -- 6-bit integers
type bv_array is array (0 to 255) of bit_vector;  -- 256 values of unspecified size
```



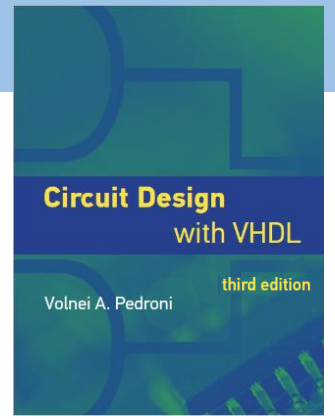
Chapter 8

User-Defined Data Types

1. User-defined types
2. Array dimensionality
3. Building and addressing complex arrays
4. Checking and resetting data arrays
5. Classical mistakes in assignments

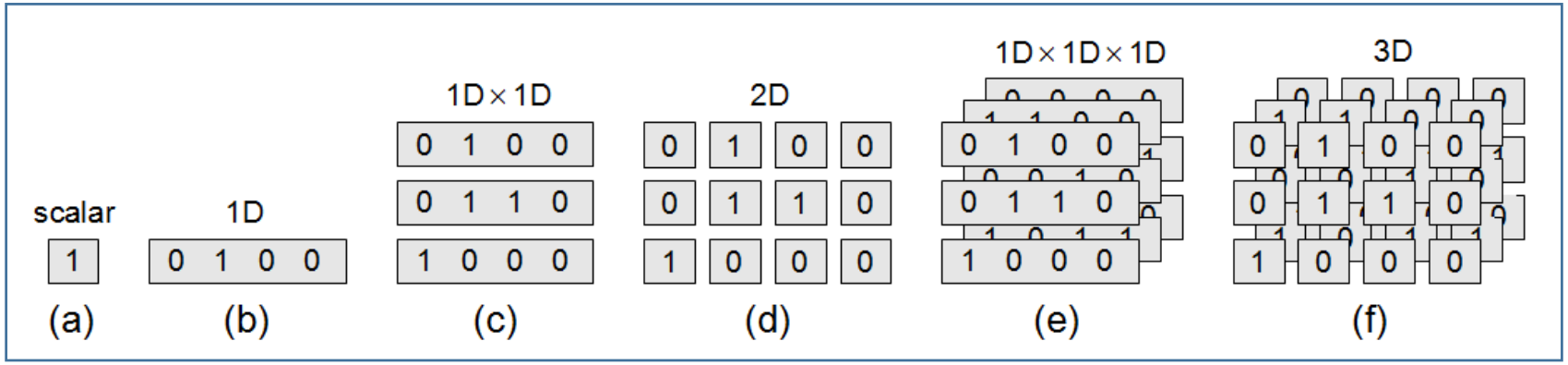


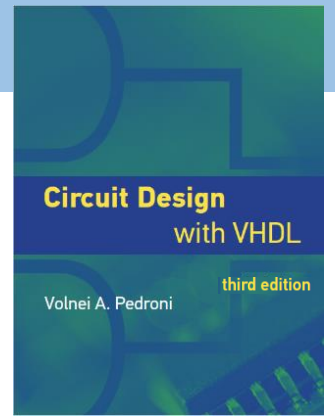
2. Array dimensionality



2. Array dimensionality

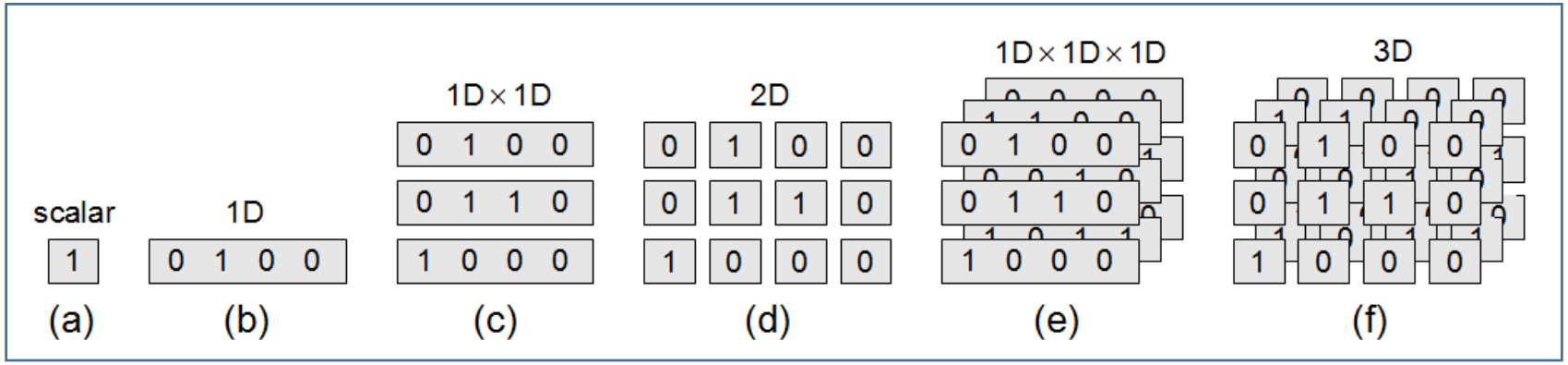
Dimension when "bit" is the measurement unit:



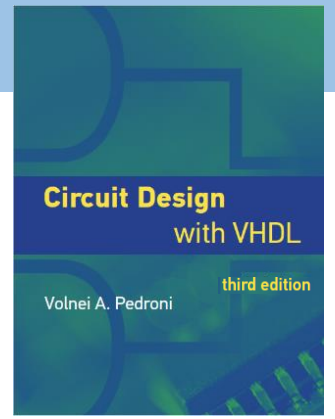


2. Array dimensionality

Dimension when "bit" is the measurement unit:

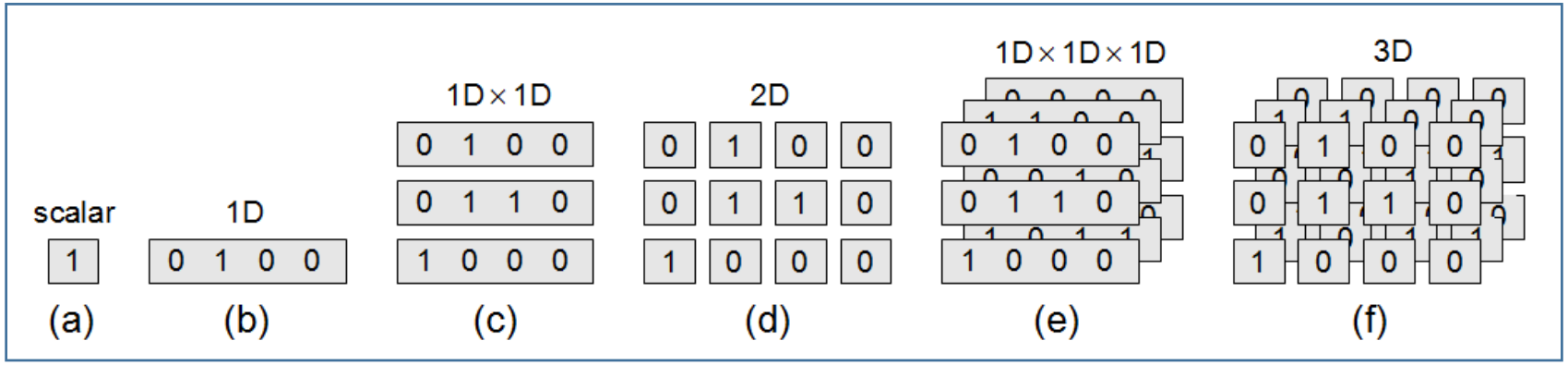


- Very important because 1 bit = 1 wire (routing density)
- An addressing scheme, not "physical" dimension
- Values above 3D or 1Dx1Dx1D generally not expected in hardware



2. Array dimensionality

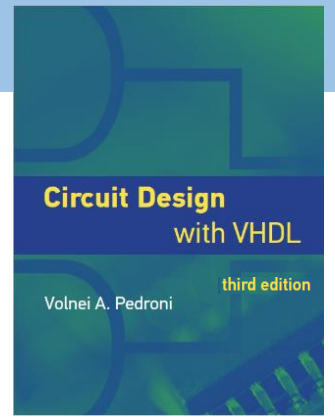
Dimension when "bit" is the measurement unit:



- Very important because 1 bit = 1 wire (routing density)
- An addressing scheme, not "physical" dimension
- Values above 3D or 1Dx1Dx1D generally not expected in hardware

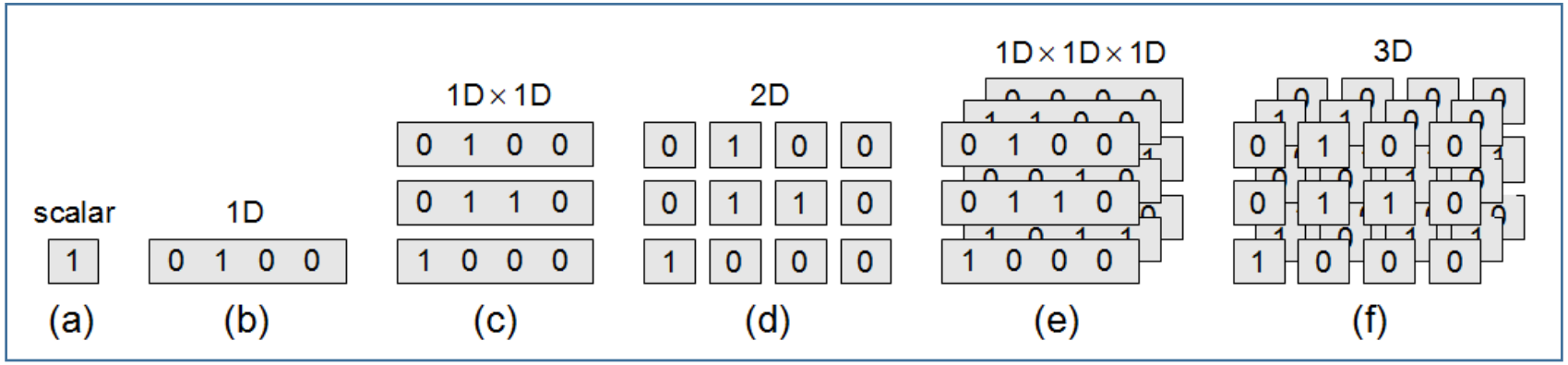
Example:
What is the dimension of these types?

```
std_logic → ?
  integer → ?
   signed → ?
   string → ?
```



2. Array dimensionality

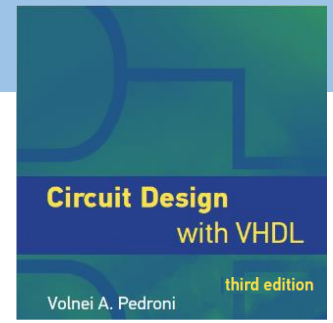
Dimension when "bit" is the measurement unit:



- Very important because 1 bit = 1 wire (routing density)
- An addressing scheme, not "physical" dimension
- Values above 3D or 1Dx1Dx1D generally not expected in hardware

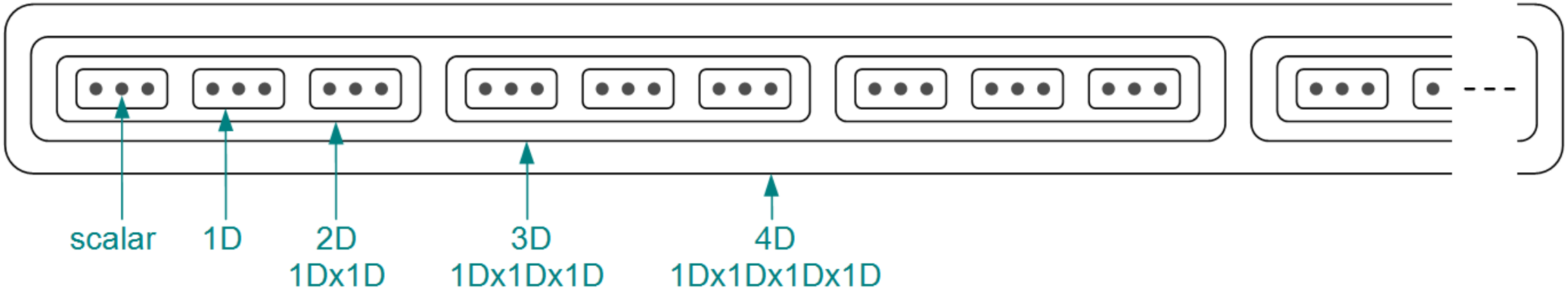
Example:
What is the dimension of these types?

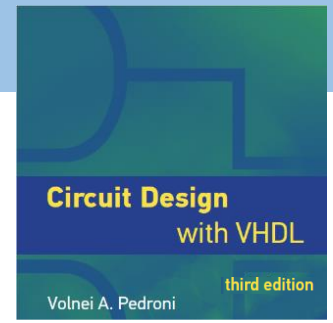
```
std_logic → scalar
  integer → 1D
   signed → 1D
   string → 1D x 1D
```



2. Array dimensionality

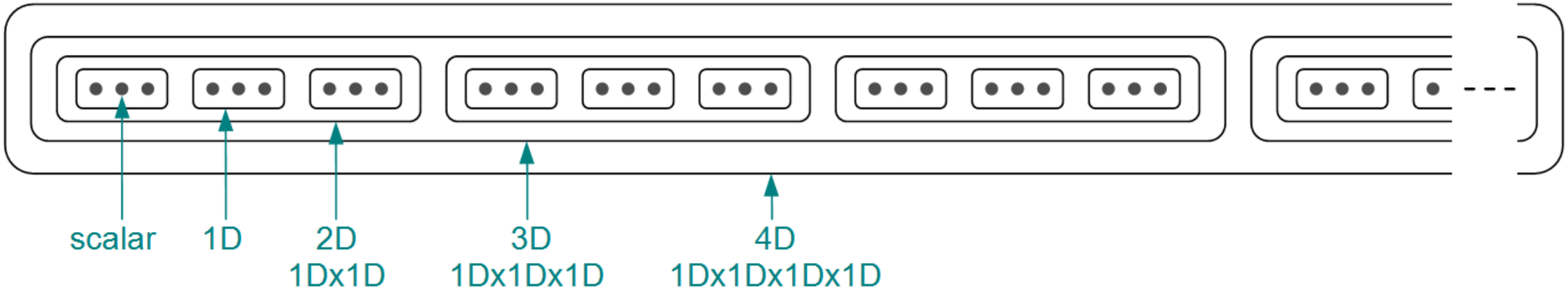
Another way of representing it:





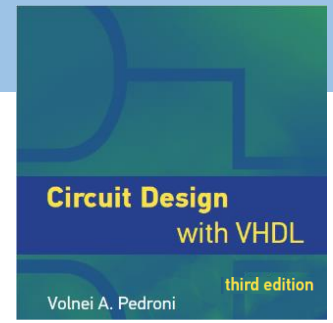
2. Array dimensionality

Another way of representing it:



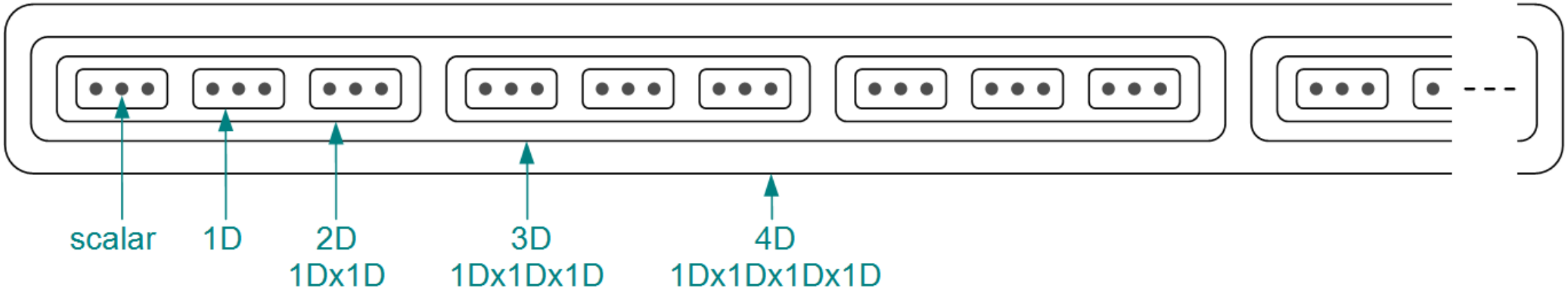
The type above could be declared as follows:

```
type type1 is array (? downto ?) of std_logic;
type type2 is array (? downto ?) of  ?;
type type3 is array (? downto ?) of  ?;
type type4 is array (? downto ?) of  ?;
```



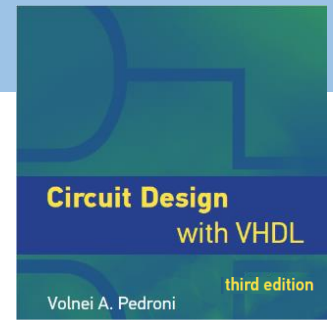
2. Array dimensionality

Another way of representing it:



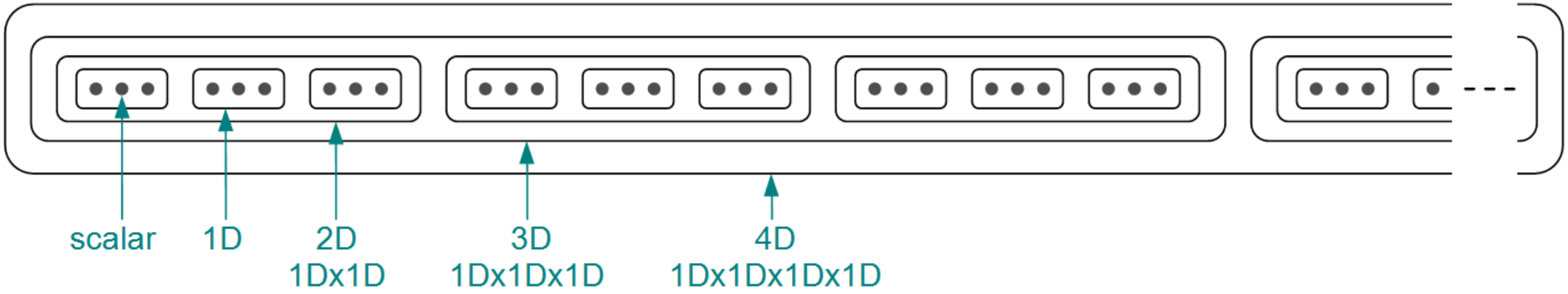
The type above could be declared as follows:

```
type type1 is array (2 downto 0) of std_logic;
type type2 is array (? downto ?) of  ?;
type type3 is array (? downto ?) of  ?;
type type4 is array (? downto ?) of  ?;
```



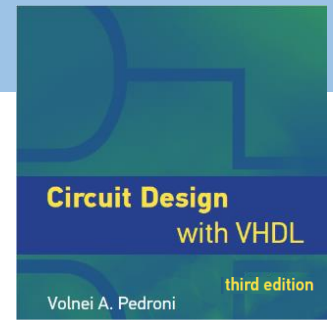
2. Array dimensionality

Another way of representing it:



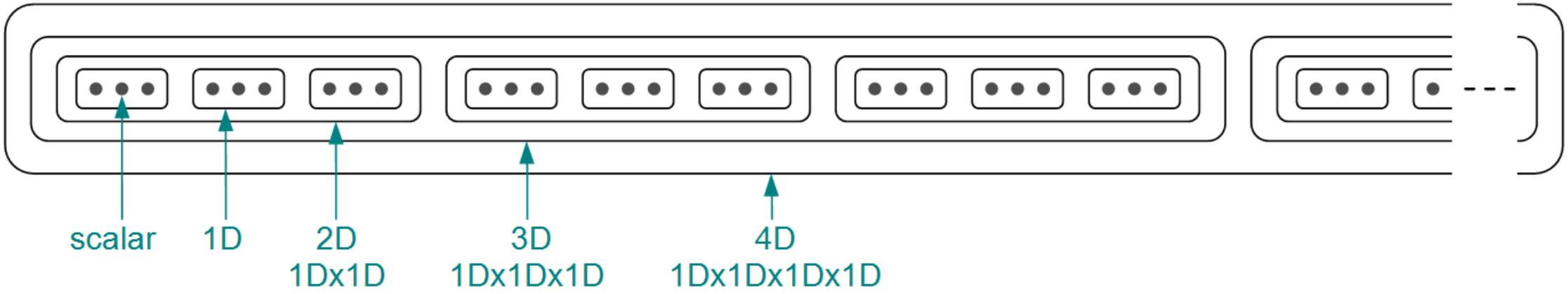
The type above could be declared as follows:

```
type type1 is array (2 downto 0) of std_logic;
type type2 is array (2 downto 0) of type1;
type type3 is array (? downto ?) of  ?;
type type4 is array (? downto ?) of  ?;
```



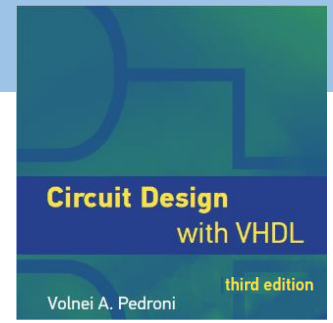
2. Array dimensionality

Another way of representing it:



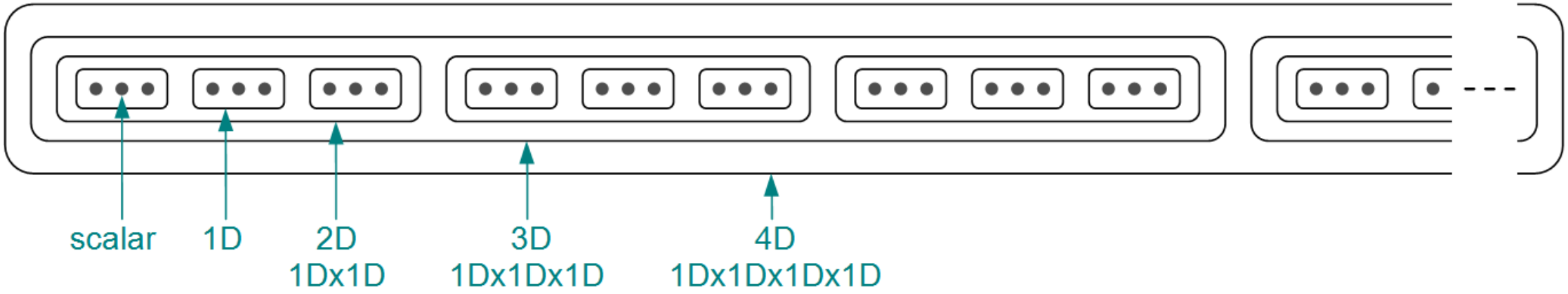
The type above could be declared as follows:

```
type type1 is array (2 downto 0) of std_logic;
type type2 is array (2 downto 0) of type1;
type type3 is array (2 downto 0) of type2;
type type4 is array (? downto ?) of  ?;
```



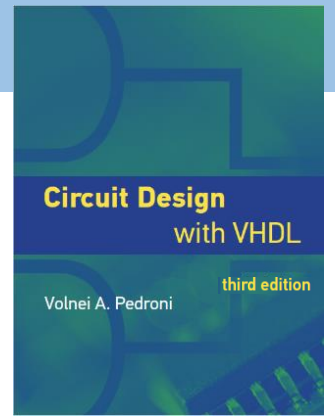
2. Array dimensionality

Another way of representing it:



The type above could be declared as follows:

```
type type1 is array (2 downto 0) of std_logic;
type type2 is array (2 downto 0) of type1;
type type3 is array (2 downto 0) of type2;
type type4 is array (1 downto 0) of type3;
```

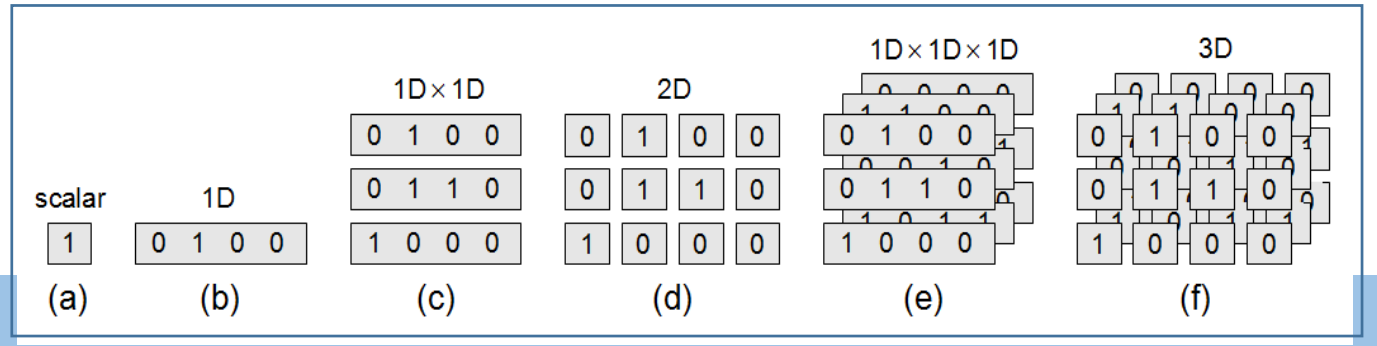


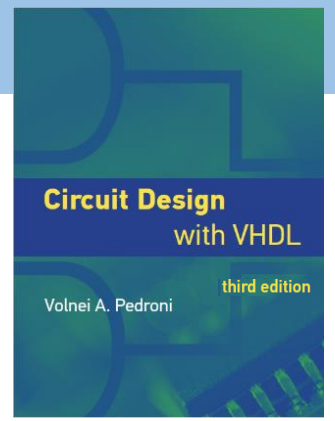
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1';
x2 <= 'Z';
x3 <= "0000";
x4 <= 50_000_000;
x5 <= ("0001", "0000", "1111 ");
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



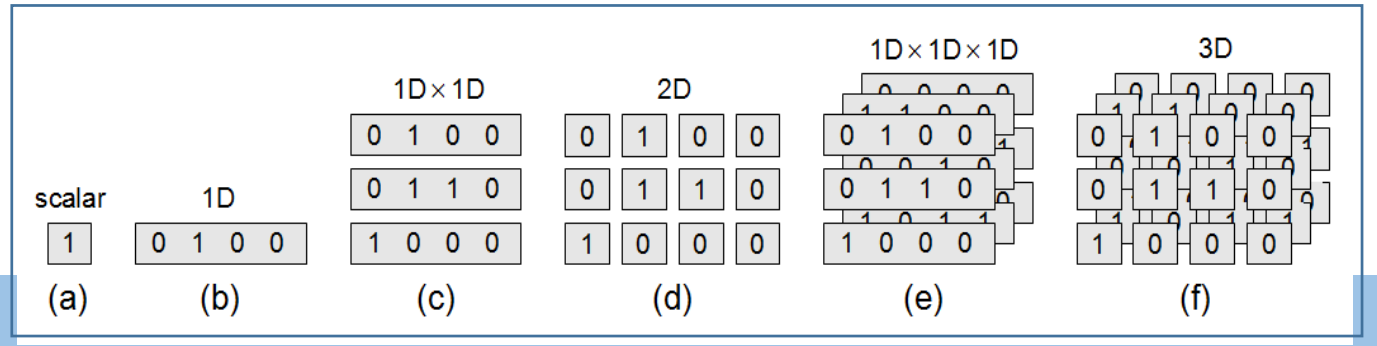


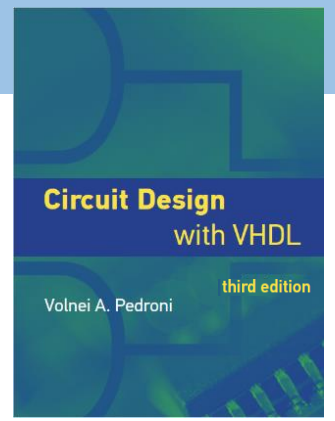
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z';
x3 <= "0000";
x4 <= 50_000_000;
x5 <= ("0001", "0000", "1111 ");
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



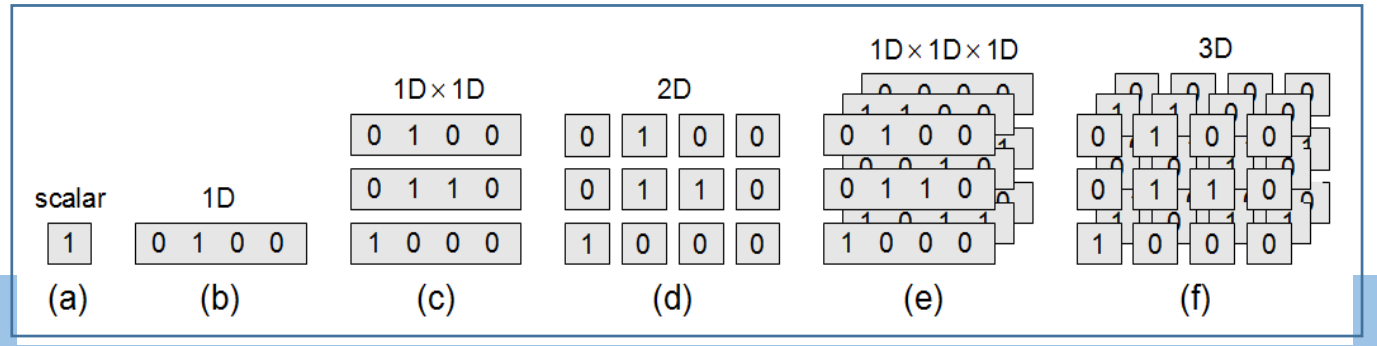


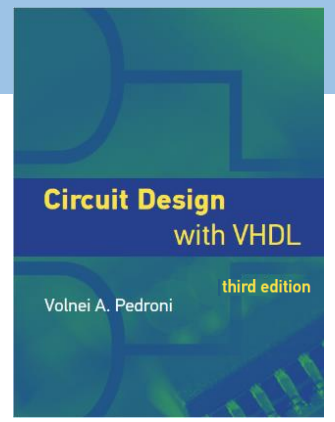
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000";
x4 <= 50_000_000;
x5 <= ("0001", "0000", "1111 ");
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



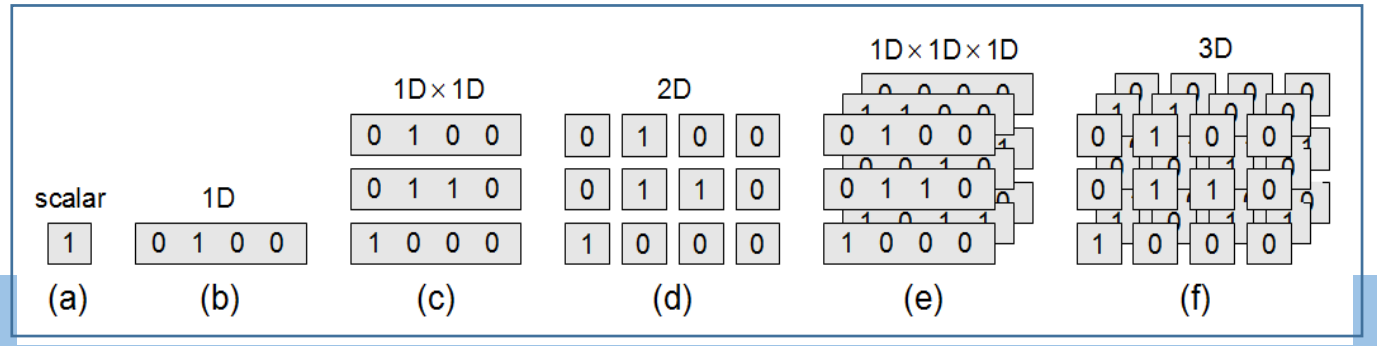


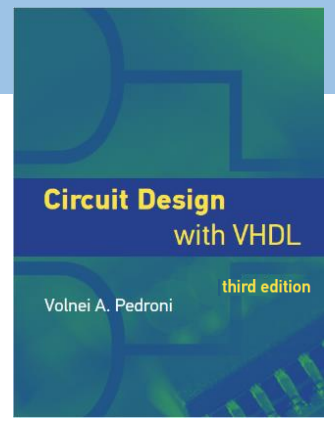
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000;
x5 <= ("0001", "0000", "1111 ");
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



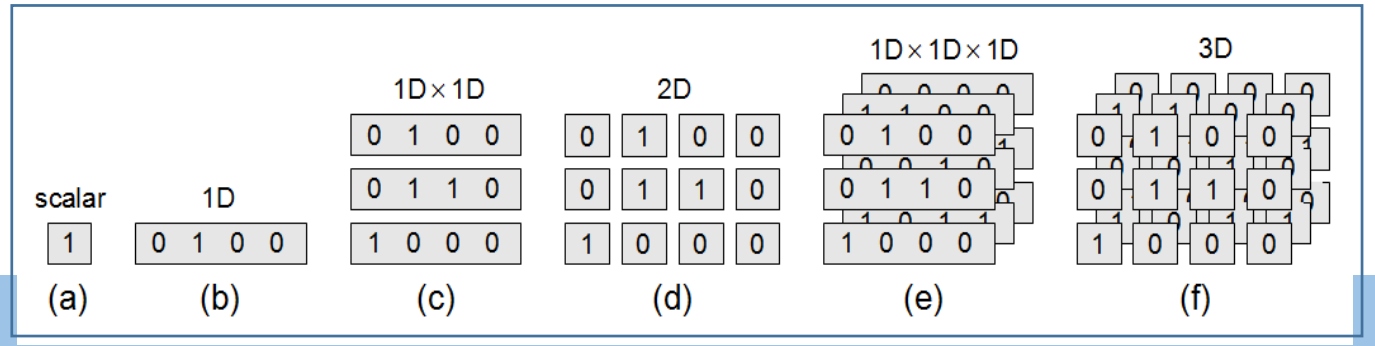


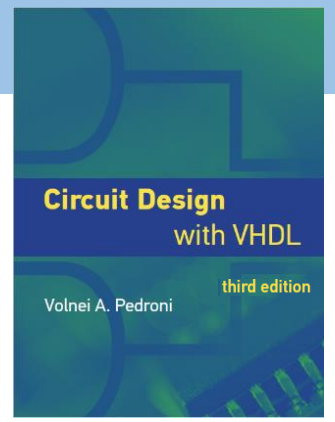
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 ");
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



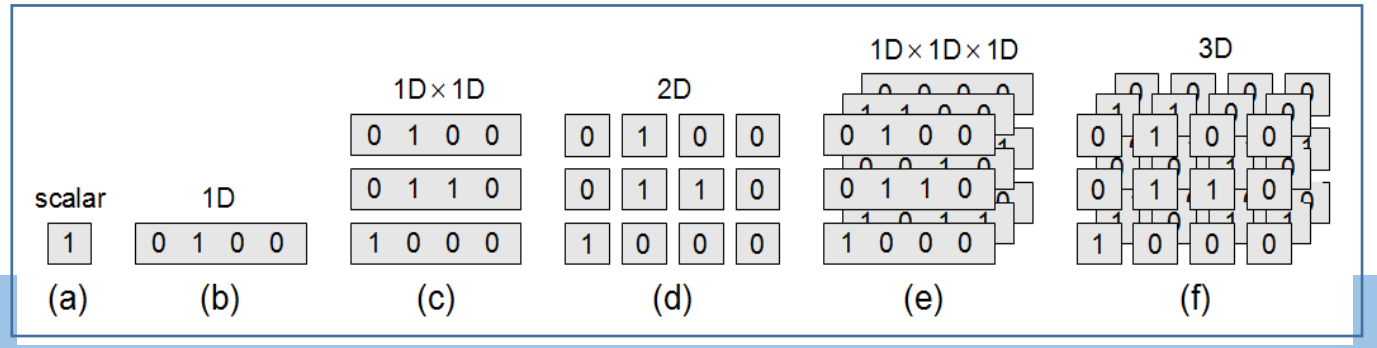


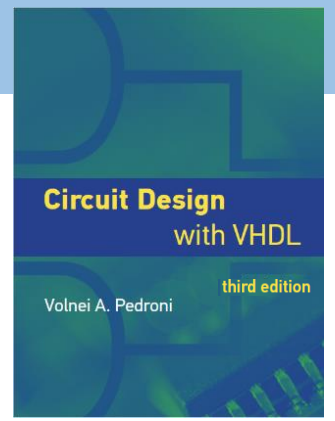
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 "); --1Dx1D or 2D
x6 <= (others => 'Z');
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



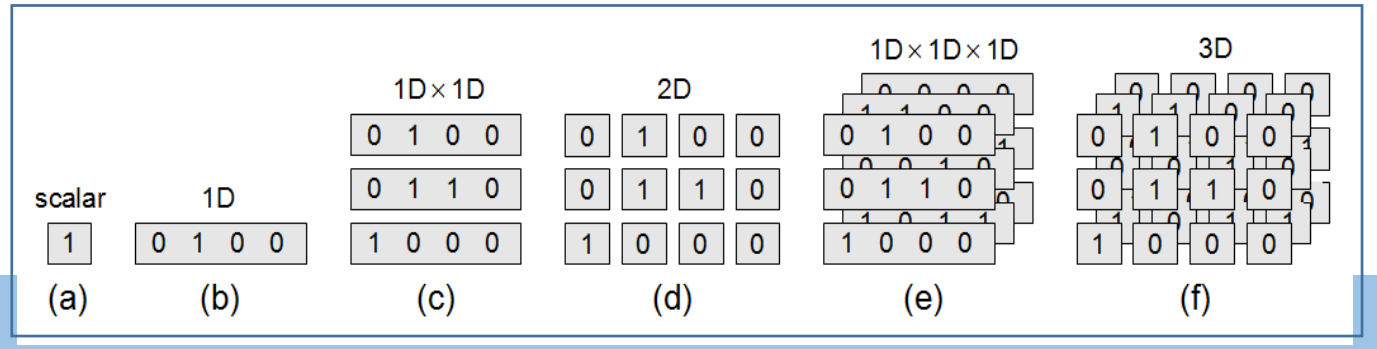


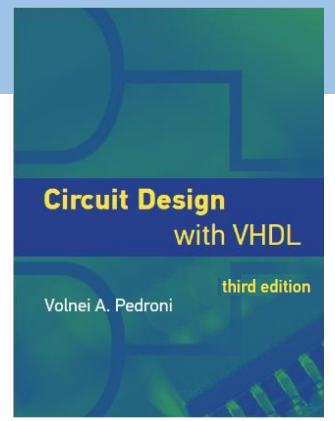
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 "); --1Dx1D or 2D
x6 <= (others => 'Z'); --1D
x7 <= ('1', '1', '0', '1', '0');
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



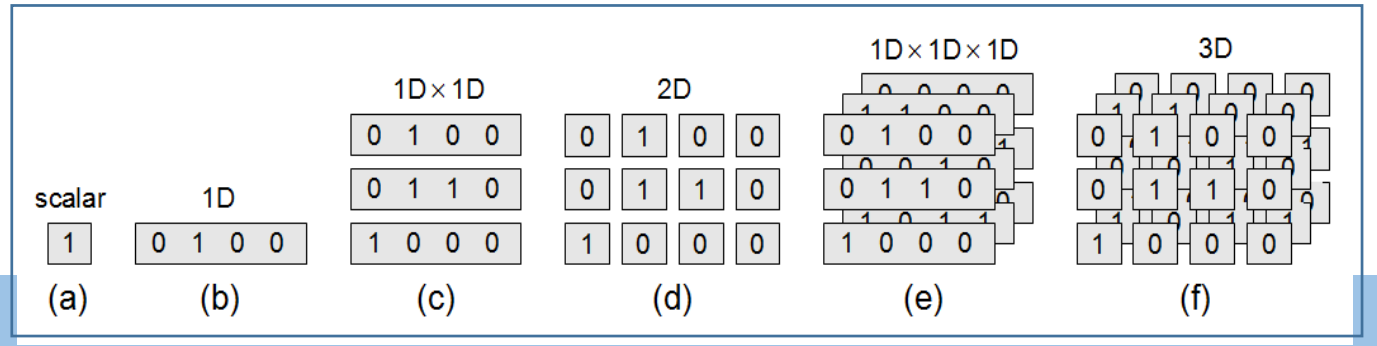


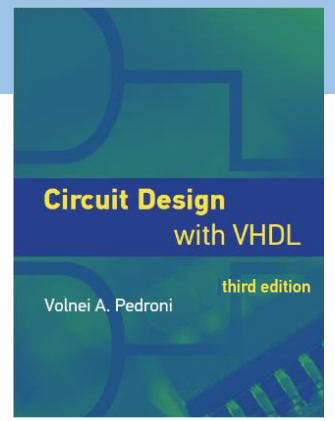
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 "); --1Dx1D or 2D
x6 <= (others => 'Z'); --1D
x7 <= ('1', '1', '0', '1', '0'); --1D
x8 <= false;
x9 <= (("000","000"),("000","000"));
```



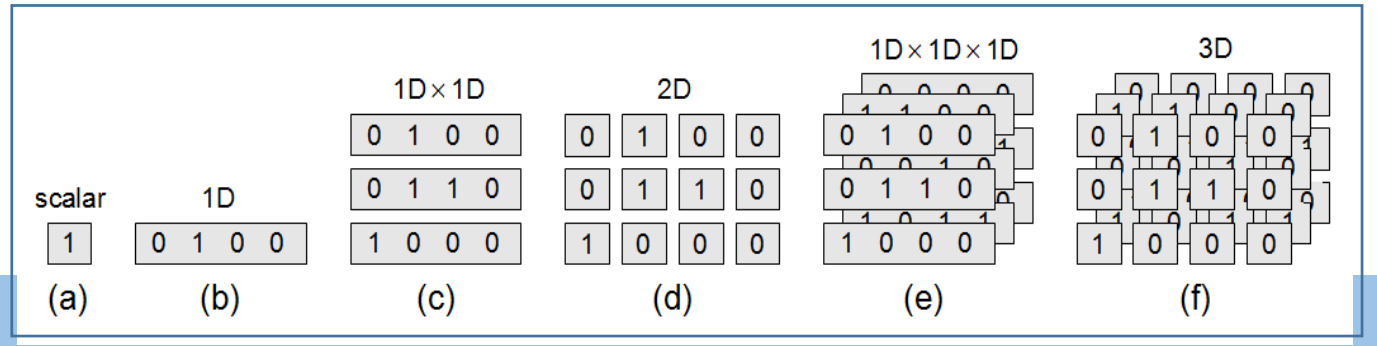


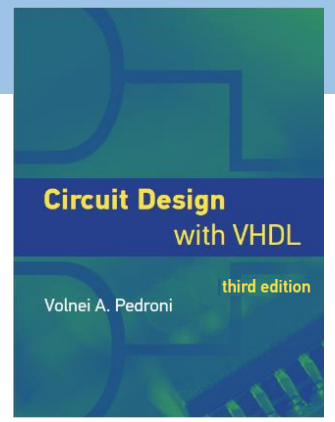
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 "); --1Dx1D or 2D
x6 <= (others => 'Z'); --1D
x7 <= ('1', '1', '0', '1', '0'); --1D
x8 <= false; --scalar
x9 <= (("000","000"),("000","000"));
```



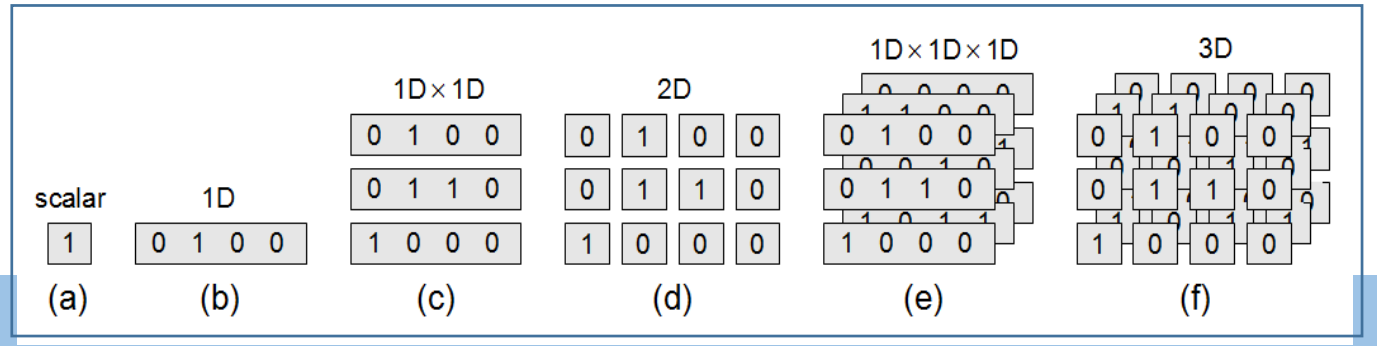


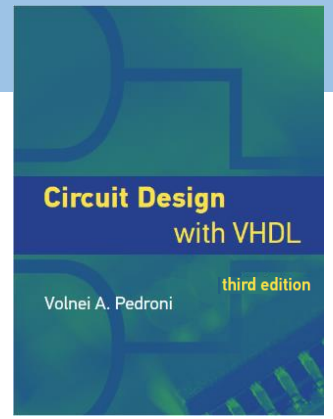
2. Array dimensionality

Another example:

What is the dimension of each signal below? Leave `char` and `string` out.

```
x1 <= '1'; --scalar
x2 <= 'Z'; --scalar
x3 <= "0000"; --1D
x4 <= 50_000_000; --1D (INT, NAT, or POS)
x5 <= ("0001", "0000", "1111 "); --1Dx1D or 2D
x6 <= (others => 'Z'); --1D
x7 <= ('1', '1', '0', '1', '0'); --1D
x8 <= false; --scalar
x9 <= (("000","000"),("000","000")); --1Dx1Dx1D or 3D
```

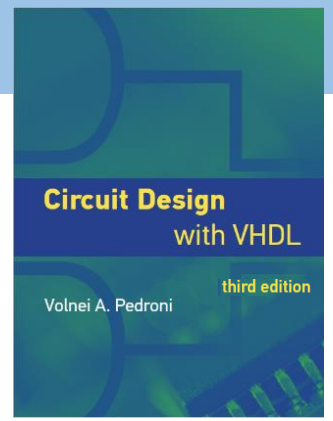




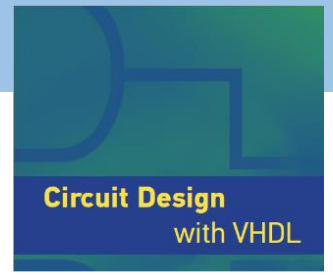
Chapter 8

User-Defined Data Types

1. User-defined types
2. Array dimensionality
3. Building and addressing complex arrays
4. Checking and resetting data arrays
5. Classical mistakes in assignments



3. Building and addressing complex arrays



3. Building and addressing complex arrays

1D×1D

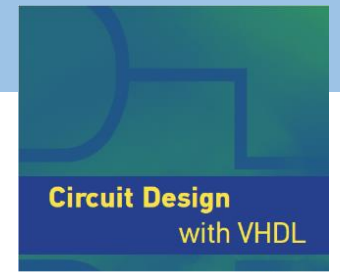
0	1	0	0
0	1	1	0
1	0	0	0

Example 1: 1Dx1D array

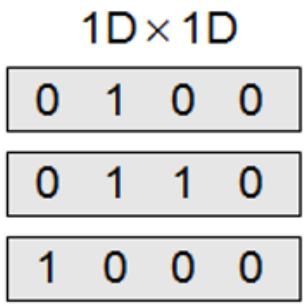
2D

0	1	0	0
0	1	1	0
1	0	0	0

Example 2: 2D array

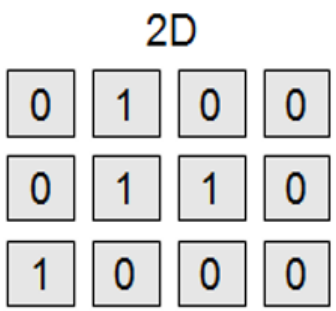


3. Building and addressing complex arrays

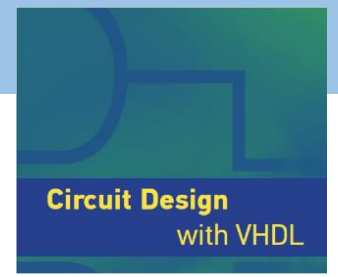


Example 1: 1Dx1D array

```
type type_1Dx1D is array (1 to 3) of std_logic_vector(3 downto 0);
signal s: type_1Dx1D;
...
s <= ("0100", "0110", "1000");
s(3) <= "1000";
s(3)(3) <= '1'; --notice two pairs of parentheses
```



Example 2: 2D array



3. Building and addressing complex arrays

1D×1D

0	1	0	0
0	1	1	0
1	0	0	0

Example 1: 1Dx1D array

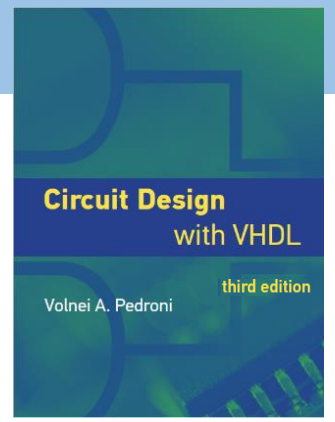
```
type type_1Dx1D is array (1 to 3) of std_logic_vector(3 downto 0);
signal s: type_1Dx1D;
...
s <= ("0100", "0110", "1000");
s(3) <= "1000";
s(3)(3) <= '1'; --notice two pairs of parentheses
```

2D

0	1	0	0
0	1	1	0
1	0	0	0

Example 2: 2D array

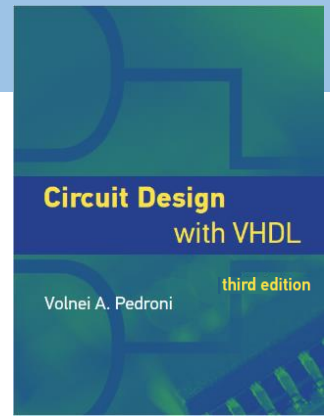
```
type type_2D is array (natural range <>, natural range <>) of std_logic;
signal s: type_2D(1 to 3, 3 downto 0);
...
s <= ("0100", "0110", "1000");
s(3) <= "1000";
s(3, 3) <= '1'; --notice a single pair of parentheses
```



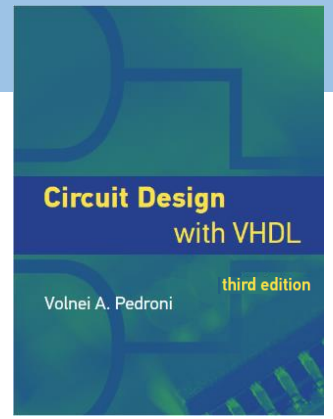
Chapter 8

User-Defined Data Types

1. User-defined types
2. Array dimensionality
3. Building and addressing complex arrays
4. Checking and resetting data arrays
5. Classical mistakes in assignments



4. Checking and resetting data arrays

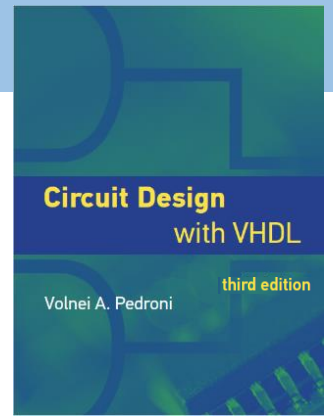


4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= ____________???____________ ;  --all zero
```

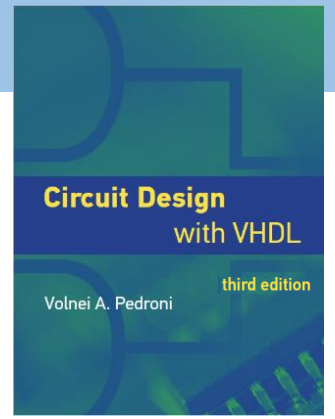


4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```



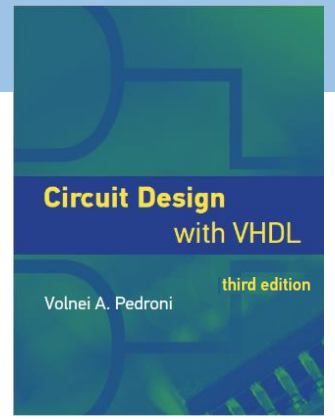
4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```

b) Checking whether entire array is zero



4. Checking and resetting data arrays

a) Zeroing entire array

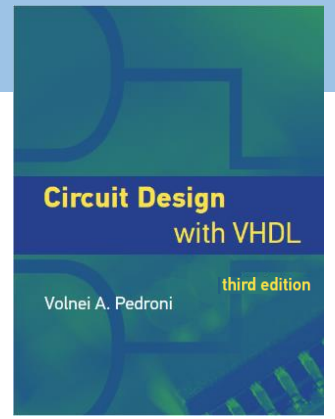
Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```

b) Checking whether entire array is zero

Example: For integer-related types (INT, UNS, SIG, …)

```
if s=0 then ...  --direct comparison to zero is fine
```



4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```

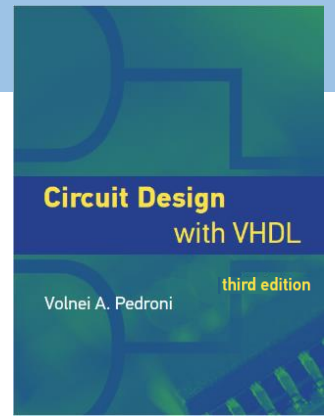
b) Checking whether entire array is zero

Example: For integer-related types (INT, UNS, SIG, …)

```
if s=0 then ...  --direct comparison to zero is fine
```

Example: For any vector type:

```
if _____???_____ then ...                --1D
```



4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```

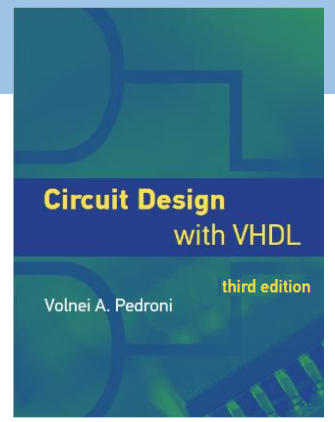
b) Checking whether entire array is zero

Example: For integer-related types (INT, UNS, SIG, …)

```
if s=0 then ...  --direct comparison to zero is fine
```

Example: For any vector type:

```
if (others => '0') then ...                --1D, illegal
```



4. Checking and resetting data arrays

a) Zeroing entire array

Example: 1D×1D or 2D array

```
s <= (others => (others => '0'));  --all zero
```

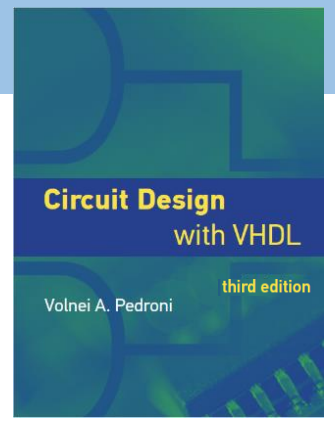
b) Checking whether entire array is zero

Example: For integer-related types (INT, UNS, SIG, …)

```
if s=0 then ...  --direct comparison to zero is fine
```

Example: For any vector type:

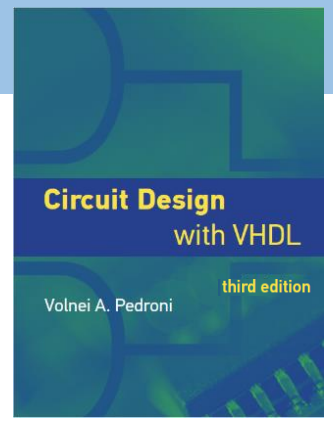
```
if (others => '0') then ...                --1D, illegal
if (s'range => '0') then ...               --1D, legal
if (s'range => (others => '0')) then ... --1D×1D, legal
```



Chapter 8

User-Defined Data Types

1. User-defined types
2. Array dimensionality
3. Building and addressing complex arrays
4. Checking and resetting data arrays
5. Classical mistakes in assignments



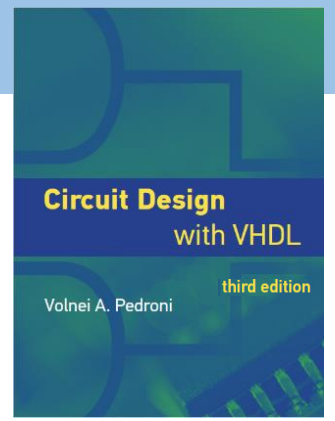
5. Classical mistakes in assignments

5. Classical mistakes in assignments

Table 8.2

Common errors in assignments

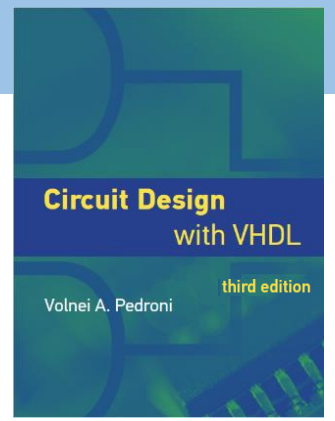
Error		Comments/examples
1	Incorrect assignment symbol	Use <= for signals Use := for variables, constants, and initial/default values
2	Type mismatch	Type must be the same on both sides of an expression
3	Invalid value or invalid representation	Incorrect use of quotation marks Type `bit` cannot receive value `'Z'`
4	Incorrect indexing	Incorrect use of parentheses for range specifications, incorrect range direction, index out of range
5	Size mismatch	Both sides of an expression must have same dimensionality and same number of bits in each dimension
6	Illegal aggregation or illegal concatenation	Can be tricky; see examples in sections 7.9.1 and 7.9.2



5. Classical mistakes in assignments

Error 1: Incorrect assignment symbol (s=signal, v=variable)

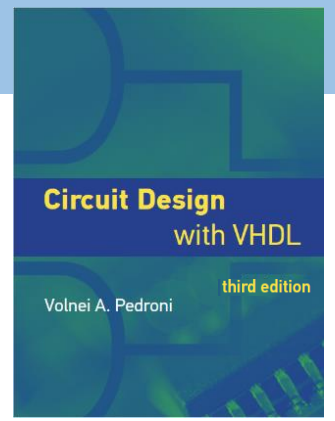
```
s := 25;     --legal?
s := v;      --legal?
s <= v;      --legal?
v <= 25;     --legal?
v := s;      --legal?
v := 25;     --legal?
```



5. Classical mistakes in assignments

Error 1: Incorrect assignment symbol (s=signal, v=variable)

```
s := 25;     --legal? No
s := v;      --legal? No
s <= v;      --legal? Yes
v <= 25;     --legal? No
v := s;      --legal? Yes
v := 25;     --legal? Yes
```



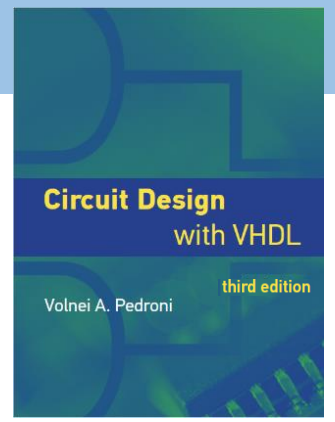
5. Classical mistakes in assignments

Error 1: Incorrect assignment symbol (s=signal, v=variable)

```
s := 25;     --legal? No
s := v;      --legal? No
s <= v;      --legal? Yes
v <= 25;     --legal? No
v := s;      --legal? Yes
v := 25;     --legal? Yes
```

Error 2: Type mismatch

```
std_logic ← bit                    --legal?
bit_vector ← std_logic_vector      --legal?
signed(0) ← unsigned(1)            --legal?
signed(0) ← std_logic_vector(1)    --legal?
```



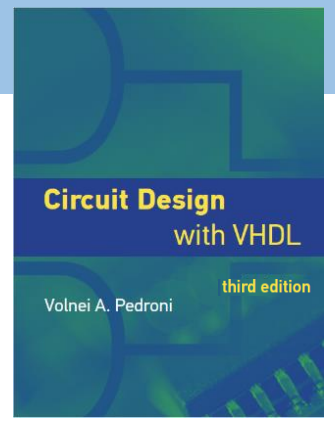
5. Classical mistakes in assignments

Error 1: Incorrect assignment symbol (s=signal, v=variable)

```
s := 25;     --legal? No
s := v;      --legal? No
s <= v;      --legal? Yes
v <= 25;     --legal? No
v := s;      --legal? Yes
v := 25;     --legal? Yes
```

Error 2: Type mismatch

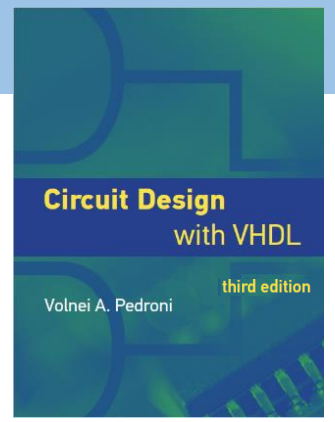
```
std_logic ← bit                    --legal? No (different types)
bit_vector ← std_logic_vector      --legal? No (different types)
signed(0) ← unsigned(1)            --legal? Yes (SU element on both sides)
signed(0) ← std_logic_vector(1)    --legal? Yes (SU element on both sides)
```



5. Classical mistakes in assignments

Error 3: Invalid value or invalid representation

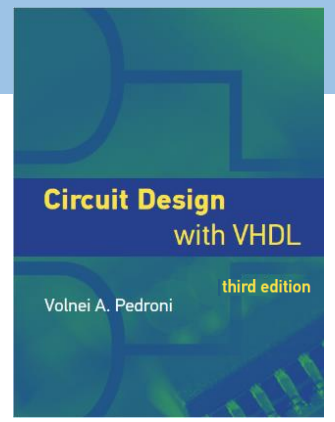
```
bit ← 'Z'                --legal?
integer ← "1111"         --legal?
std_logic ← 'Z'          --legal?
integer ← 1111           --legal?
bit_vector ← 0000        --legal?
```



5. Classical mistakes in assignments

Error 3: Invalid value or invalid representation

```
bit ← 'Z'                --legal? No
integer ← "1111"         --legal? No
std_logic ← 'Z'          --legal? Yes
integer ← 1111           --legal? Yes
bit_vector ← 0000        --legal? No
```



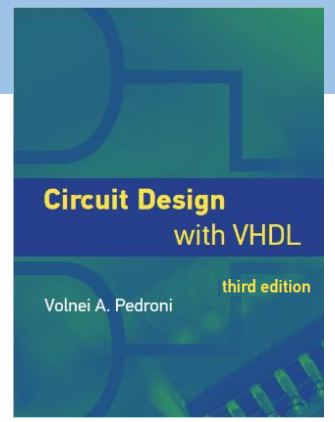
5. Classical mistakes in assignments

Error 3: Invalid value or invalid representation

```
bit ← 'Z'                --legal? No
integer ← "1111"         --legal? No
std_logic ← 'Z'          --legal? Yes
integer ← 1111           --legal? Yes
bit_vector ← 0000        --legal? No
```

Error 4: Incorrect indexing

```
signal s1: std_logic_vector(7 downto 0);
signal s2: std_logic_vector(1 to 4);
...
s1(7 downto 5) <= s2(4 downto 2);    --legal?
s1(7 downto 5) <= s2(0 to 2);        --legal?
s1(7 downto 5) <= s2(2 to 4);        --legal?
s1(7, 5) <= "110";                   --legal?
```



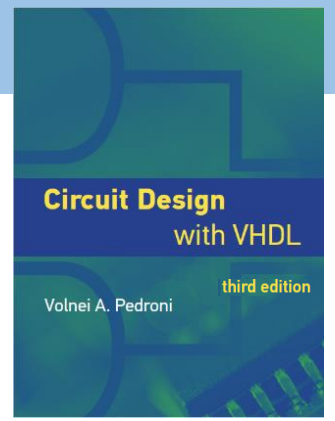
5. Classical mistakes in assignments

Error 3: Invalid value or invalid representation

```
bit ← 'Z'                --legal? No
integer ← "1111"         --legal? No
std_logic ← 'Z'          --legal? Yes
integer ← 1111           --legal? Yes
bit_vector ← 0000        --legal? No
```

Error 4: Incorrect indexing

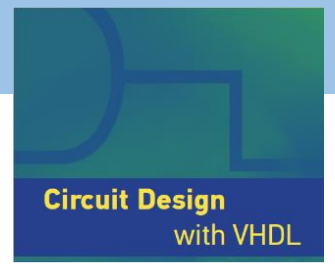
```
signal s1: std_logic_vector(7 downto 0);
signal s2: std_logic_vector(1 to 4);
...
s1(7 downto 5) <= s2(4 downto 2);    --legal? No (wrong direction for s2)
s1(7 downto 5) <= s2(0 to 2);        --legal? No (s2 is out of range)
s1(7 downto 5) <= s2(2 to 4);        --legal? Yes
s1(7, 5) <= "110";                   --legal? No (should be (7 downto 5))
```



5. Classical mistakes in assignments

Error 5: Size mismatch

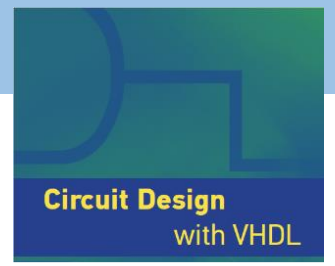
```
bit_vector(7 downto 0) ← bit_vector(0 to 3)   --legal?
std_logic_vector(0) ← std_logic               --legal?
bit_vector(7 downto 4) ← bit_vector(0 to 3)   --legal?
std_logic_vector(1 to 2) ← std_logic          --legal?
```



5. Classical mistakes in assignments

Error 5: Size mismatch

```
bit_vector(7 downto 0) ← bit_vector(0 to 3)   --legal? No (8b x 4b)
std_logic_vector(0) ← std_logic               --legal? Yes (1b x 1b, same type)
bit_vector(7 downto 4) ← bit_vector(0 to 3)   --legal? Yes (4b x 4b, same type)
std_logic_vector(1 to 2) ← std_logic          --legal? No (2b x 1b, type mismatch)
```



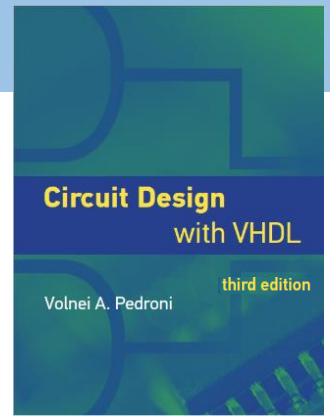
5. Classical mistakes in assignments

Error 5: Size mismatch

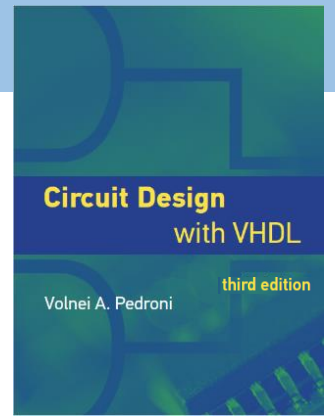
```
bit_vector(7 downto 0) ← bit_vector(0 to 3)   --legal? No (8b x 4b)
std_logic_vector(0) ← std_logic               --legal? Yes (1b x 1b, same type)
bit_vector(7 downto 4) ← bit_vector(0 to 3)   --legal? Yes (4b x 4b, same type)
std_logic_vector(1 to 2) ← std_logic          --legal? No (2b x 1b, type mismatch)
```

Error 6: Incorrect aggregation/concatenation

Seen in sections 7.9.1 and 7.9.2 (can be tricky!)



Final example:



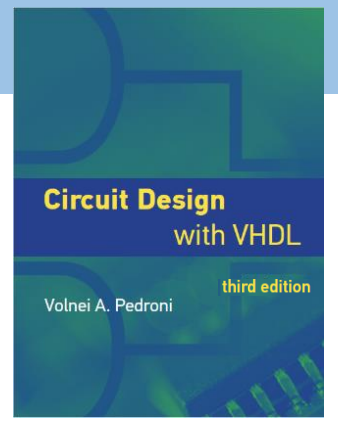
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --
a(0) <= '0';           --
a <= b;                --
c <= d;                --
e(0) <= "1111";        --
e(0,7) <= 0;           --
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



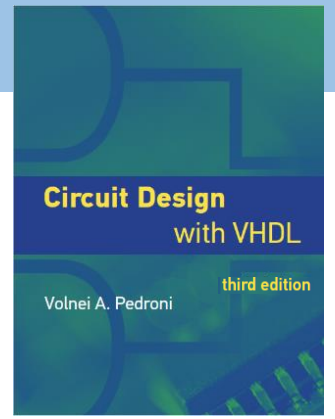
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --
a <= b;                --
c <= d;                --
e(0) <= "1111";        --
e(0,7) <= 0;           --
f(0)(8) := '-';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



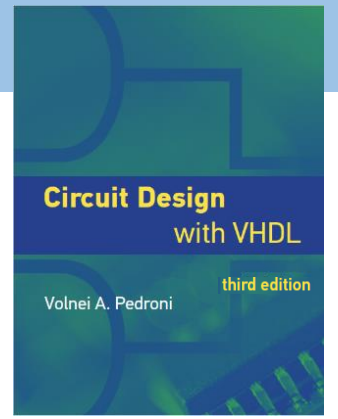
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --
c <= d;                --
e(0) <= "1111";        --
e(0,7) <= 0;           --
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



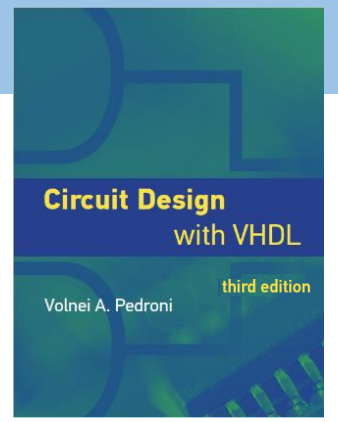
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --error 2
c <= d;                --
e(0) <= "1111";        --
e(0,7) <= 0;           --
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



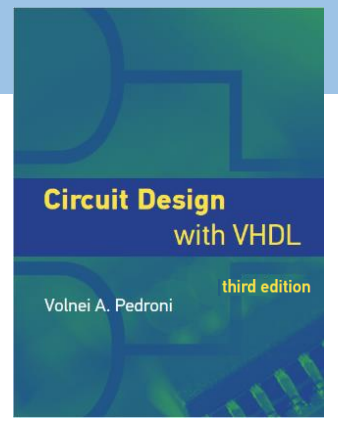
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --error 2
c <= d;                --error 2 + error 5
e(0) <= "1111";        --
e(0,7) <= 0;           --
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



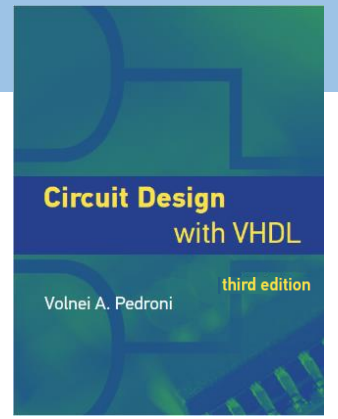
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --error 2
c <= d;                --error 2 + error 5
e(0) <= "1111";        --error 5
e(0,7) <= 0;           --
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



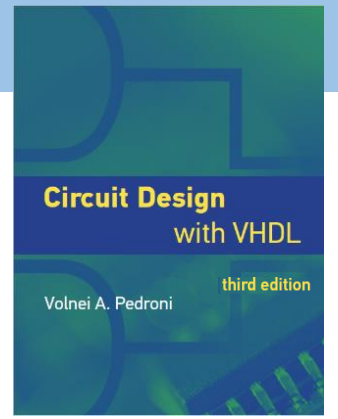
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --error 2
c <= d;                --error 2 + error 5
e(0) <= "1111";        --error 5
e(0,7) <= 0;           --error 3 + error 4
f(0)(8) := '–';        --
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



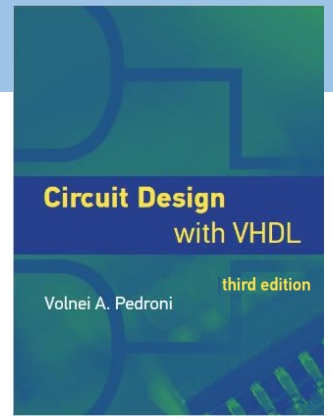
Final example:

```
type row is array (7 downto 0) of std_logic;
type matrix1 is array (0 to 3) of row;
type matrix2 is array (0 to 3, 7 downto 0) of std_logic;
...
signal a: bit;
signal b: std_logic;
signal c: std_logic_vector(3 downto 0);
signal d: row;
signal e: matrix1;
signal f: matrix2;
```

Why are the assignments below illegal?

```
a <= 'Z';              --error 3
a(0) <= '0';           --error 4
a <= b;                --error 2
c <= d;                --error 2 + error 5
e(0) <= "1111";        --error 5
e(0,7) <= 0;           --error 3 + error 4
f(0)(8) := '–';        --error 1 + error 4 (wrong parent. and out of range)
```

Error	
1	Incorrect assignment symbol
2	Type mismatch
3	Invalid value or invalid representation
4	Incorrect indexing
5	Size mismatch
6	Illegal aggregation or illegal concatenation



End of Chapter 8