



Chapter 2.16-2.19

Introduction to VHDL

Version: 2023/11/29

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.16 Variables, Signals and Constants

- VHDL provides two objects for dealing with non-static data values: **signal** and **variable**
 - VHDL also provides means for establishing default (static) values: **constant** and **generic**
-
- **constant** and **signal** can be **global** (that is, seen by the whole code), and can be used in either type of code, concurrent or sequential
 - **variable** is **local**, for it can only be used inside **a piece of sequential code** (that is in a process, function, or procedure) and its value can never be passed out directly

2.16 Variables, Signals and Constants

Declaration

```
variable list_of_variable_names: type_name [:= initial_value];
```

```
signal list_of_signal_names : type_name [:= initial_value];
```

```
constant constant_name : type_name := constant_value;
```

	declared ...	where it can be used
Variables	<u>within process</u>	local to process
Signals	at the start of an architecture	anywhere within architecture
Constants	at the start of an architecture	anywhere within architecture
	within a process	local to process

2.16 Variables, Signals and Constants

Assignment statements

```
variable_name := expression;
```

```
signal_name <= expression [after delay];
```

	scheduled to change...
Signals	after delay or Δ delay
Variables	with no delay

2.16 Variables, Signals and Constants

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
    begin
        wait on trigger;
        var1 := var2 + var3;
        var2 := var1;
        var3 := var2;
        sum <= var1 + var2 + var3;
    end process;
end var;
```

Variable: must be declared and initialized inside the process

```
entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
        wait on trigger;
        sig1 <= sig2 + sig3;
        sig2 <= sig1;
        sig3 <= sig2;
        sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

Signal: must be declared and initialized outside the process

2.16 Variables, Signals and Constants

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
            wait on trigger;
            var1 := var2 + var3;
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
        end process;
    end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0	+0	0	1	2	3	0

During **simulation**, initialization makes the process execute once, and it stops when **wait** statement are encountered

assume trigger changes at time = 10 ns

2.16 Variables, Signals and Constants

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
            wait on trigger;
            var1 := var2 + var3;
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
        end process;
    end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0

2.16 Variables, Signals and Constants

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
    begin
        wait on trigger;
        var1 := var2 + var3;
        var2 := var1;
        var3 := var2;
        sum <= var1 + var2 + var3;
    end process;
end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	5	5	5	0

var1~3 are computed sequentially and updated instantly (using **new values**)

Sum is computed using **new var1~3**

2.16 Variables, Signals and Constants

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
            wait on trigger;
            var1 := var2 + var3;
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
        end process;
    end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	5	5	5	0
10	+1	1	5	5	5	15

signal is updated with Δ delay

Variables work just as variables used in another language,
whereas signals get updated with time delays

2.16 Variables, Signals and Constants

ns	delta	Trigger	sig1	sig2	sig3	sum
0	+0	0	1	2	3	0

```
entity dummy is  
end dummy;
```

```
architecture sig of dummy is  
    signal trigger, sum: integer:=0;  
    signal sig1: integer:=1;  
    signal sig2: integer:=2;  
    signal sig3: integer:=3;  
begin  
    process  
    begin  
        wait on trigger;  
        sig1 <= sig2 + sig3;  
        sig2 <= sig1;  
        sig3 <= sig2;  
        sum <= sig1 + sig2 + sig3;  
    end process;  
end sig;
```

2.16 Variables, Signals and Constants

ns	delta	Trigger	sig1	sig2	sig3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0

```
entity dummy is  
end dummy;
```

```
architecture sig of dummy is  
    signal trigger, sum: integer:=0;  
    signal sig1: integer:=1;  
    signal sig2: integer:=2;  
    signal sig3: integer:=3;  
begin  
    process  
    begin  
        wait on trigger;  
        sig1 <= sig2 + sig3;  
        sig2 <= sig1;  
        sig3 <= sig2;  
        sum <= sig1 + sig2 + sig3;  
    end process;  
end sig;
```

2.16 Variables, Signals and Constants

ns	delta	Trigger	sig1	sig2	sig3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	1	2	3	0

```
entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
        wait on trigger;
        sig1 <= sig2 + sig3;
        sig2 <= sig1;
        sig3 <= sig2;
        sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

2.16 Variables, Signals and Constants

ns	delta	Trigger	sig1	sig2	sig3	sum
0	+0	0	1	2	3	0
0	+1	0	1	2	3	0
10	+0	1	1	2	3	0
10	+1	1	5	1	2	6

- **Old values** of sig1 and sig2 are used to compute sig2 and sig3
- **Old values** of sig1,2,3 are used to compute sum

```
entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
        wait on trigger;
        sig1 <= sig2 + sig3;
        sig2 <= sig1;
        sig3 <= sig2;
        sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

2.16 Variables, Signals and Constants

```
entity dummy3 is  
end dummy3;  
architecture var of dummy3 is  
signal trigger, sum: integer:=0;  
begin  
  process(trigger)  
    variable var1: integer:=1;  
    variable var2: integer:=2;  
    variable var3: integer:=3;  
    begin  
      var1 := var2 + var3;  
      var2 := var1;  
      var3 := var2;  
      sum <= var1 + var2 + var3;  
    end process;  
end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0-	+0	0	1	2	3	0
0	+0	0	5	5	5	0

2.16 Variables, Signals and Constants

```
entity dummy3 is  
end dummy3;  
architecture var of dummy3 is  
signal trigger, sum: integer:=0;  
begin  
  process(trigger)  
    variable var1: integer:=1;  
    variable var2: integer:=2;  
    variable var3: integer:=3;  
    begin  
      var1 := var2 + var3;  
      var2 := var1;  
      var3 := var2;  
      sum <= var1 + var2 + var3;  
    end process;  
end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0-	+0	0	1	2	3	0
0	+0	0	5	5	5	0
0	+1	0	5	5	5	15

2.16 Variables, Signals and Constants

```
entity dummy3 is  
end dummy3;  
architecture var of dummy3 is  
signal trigger, sum: integer:=0;  
begin  
  process(trigger)  
    variable var1: integer:=1;  
    variable var2: integer:=2;  
    variable var3: integer:=3;  
    begin  
      var1 := var2 + var3;  
      var2 := var1;  
      var3 := var2;  
      sum <= var1 + var2 + var3;  
    end process;  
end var;
```

ns	delta	Trigger	var1	var2	var3	sum
0-	+0	0	1	2	3	0
0	+0	0	5	5	5	0
0	+1	0	5	5	5	15
10	+0	1	10	10	10	15
10	+1	1	10	10	10	30

2.16 Variables, Signals and Constants

```

entity dummy3 is
end dummy3;
architecture var of dummy3 is
signal trigger, sum: integer:=0;
begin
    process(trigger)
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
            var1 := var2 + var3;
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
        end process;
    end var;
  
```

ns	delta	Trigger	var1	var2	var3	sum
0-	+0	0	1	2	3	0
0	+0	0	5	5	5	0
0	+1	0	5	5	5	15
10	+0	1	10	10	10	15
10	+1	1	10	10	10	30

```

entity dummy4 is
end dummy4;
architecture sig of dummy4 is
signal trigger, sum: integer:=0;
signal sig1: integer:=1;
signal sig2: integer:=2;
signal sig3: integer:=3;
begin
    process(trigger)
        begin
            sig1 <= sig2 + sig3;
            sig2 <= sig1;
            sig3 <= sig2;
            sum <= sig1 + sig2 + sig3;
        end process;
    end sig;
  
```

ns	delta	Trigger	sig1	sig2	sig3	sum
0	+0	0	1	2	3	0
0	+1	0	5	1	2	6
10	+0	1	5	1	2	6
10	+1	1	3	5	1	8

2.16 Variables, Signals and Constants

dummy3

```
entity dummy is  
end dummy;
```

```
architecture var of dummy is  
  signal trigger, sum : integer := 0;  
begin  
  process(trigger)  
    variable var1 : integer := 1;  
    variable var2 : integer := 2;  
    variable var3 : integer := 3;  
    begin  
      var1 := var2 + var3;  
      var2 := var1;  
      var3 := var2;  
      sum <= var1 + var2 + var3;  
    end process;  
end var;
```

After add trigger, var1~3, and sum

ns	/dummy/trigger	/dummy/line__7/var2	/dummy/sum		
delta	/dummy/line__7/var1	/dummy/line__7/var3			
0 +0	0	1	2	3	0

After simulation running

```
VSIM 21> force dummy/trigger 1 10 ns  
VSIM 22> run 20 ns
```

ns	/dummy/trigger	/dummy/line__7/var2	/dummy/sum		
delta	/dummy/line__7/var1	/dummy/line__7/var3			
0 +0	0	5	5	5	0
0 +1	0	5	5	5	15
10 +0	1	10	10	10	15
10 +1	1	10	10	10	30

- These differences are not important when VHDL is used for **synthesis** of hardware
- These are subtle differences that only affect **simulation**

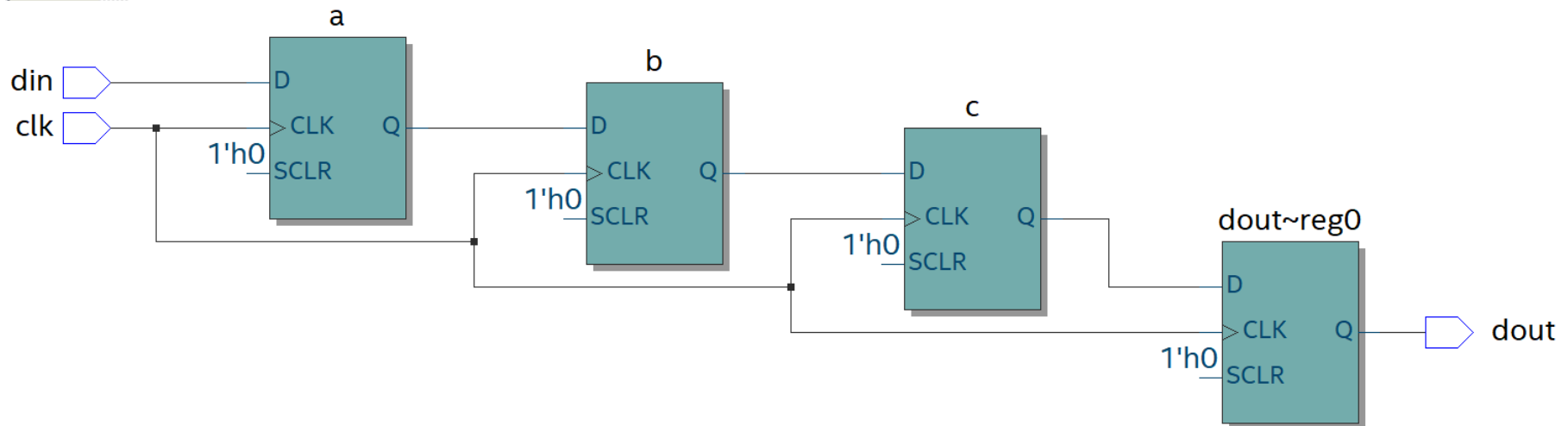
Comparison between **signal** and **variable**

	Signal	Variable
Assignment	<code><=</code>	<code>:=</code>
Utility	Represents circuit interconnections (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding Process, Function, or Procedure)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the Process, Function, or Procedure)	Updated immediately (new value can be used in the next line of code)
Usage	In a Package, Entity, or Architecture. In an Entity, all Ports are Signals by default	Only in sequential code, that is, in a Process, Function, or Procedure

```

1  entity shift is
2  port (din, clk: in bit;
3        dout: out bit);
4  end shift;
5
6  architecture shift of shift is
7  signal a, b, c: bit;
8  begin
9  process(clk)
10   begin
11     if clk'event and clk='1' then
12       a <= din;
13       b <= a;
14       c <= b;
15       dout <= c;
16     end if;
17   end process;
18 end shift;

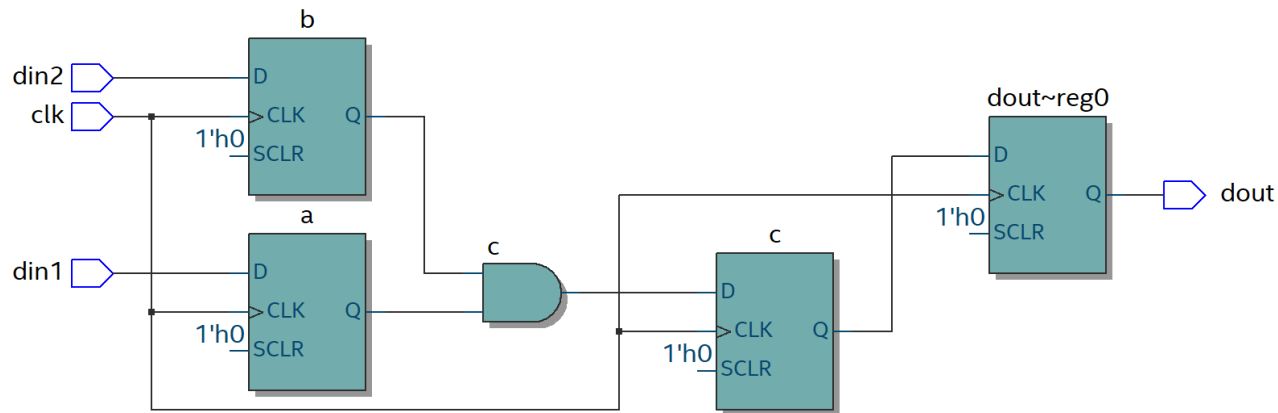
```



```

1  entity shift is
2  port (din1, din2, clk: in bit;
3        dout: out bit);
4  end shift;
5
6  architecture shift of shift is
7  signal a, b, c: bit;
8  begin
9  process(clk)
10 begin
11     if clk'event and clk='1' then
12         a <= din1;
13         b <= din2;
14         c <= a and b;
15         dout <= c;
16     end if;
17 end process;
18 end shift;
19

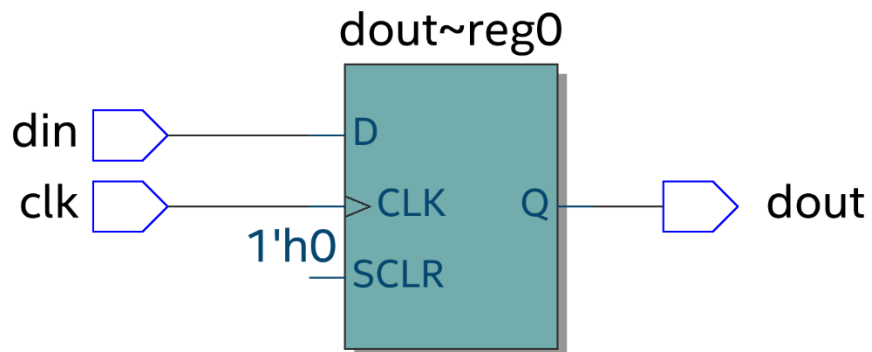
```



```

1  entity shift is
2  port (din, clk: in bit;
3        dout: out bit);
4  end shift;
5
6  architecture shift of shift is
7  begin
8      process(clk)
9          variable a, b, c: bit;
10         begin
11             if clk'event and clk='1' then
12                 a := din;
13                 b := a;
14                 c := b;
15                 dout <= c;
16             end if;
17         end process;
18     end shift;

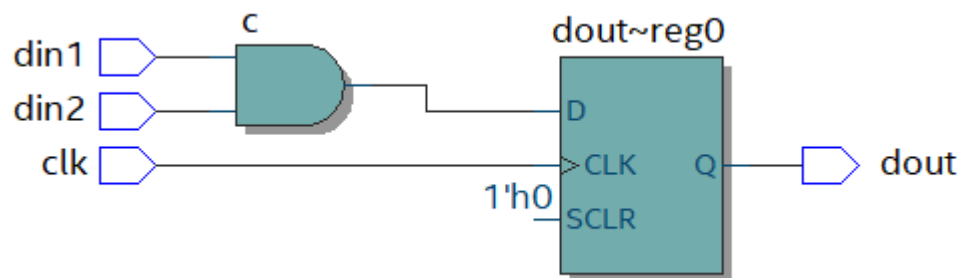
```



```

1  entity shift is
2  port (din1, din2, clk: in bit;
3        dout: out bit);
4  end shift;
5
6  architecture shift of shift is
7  begin
8      process(clk)
9          variable a, b, c: bit;
10         begin
11             if clk'event and clk='1' then
12                 a := din1;
13                 b := din2;
14                 c := a and b;
15                 dout <= c;
16             end if;
17         end process;
18     end shift;
19

```




```

1  entity shift is
2  port (din, clk: in bit;
3        dout: out bit);
4  end shift;
5
6  architecture shift of shift is
7  begin
8  process(clk)
9      variable a, b, c: bit;
10  begin
11      if clk'event and clk='1' then
12          a := din;
13          b := a;
14          c := b;
15          dout := c;
16      end if;
17  end process;
18  end shift;

```

- ✘ 10526 VHDL Signal Assignment Statement error at shift.vhd(15): Signal Assignment Statement must use <= to assign value to signal "dout"
- ✘ Quartus Prime Analysis & Synthesis was unsuccessful. 1 error, 1 warning
- ✘ 293001 Quartus Prime Full Compilation was unsuccessful. 3 errors, 1 warning

Signal assignment statement must use
<= to assign value to signal

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.17 Arrays

Array in VHDL can be used while modeling the repetition

To use array in VHDL,

1. Declare an array type
2. Declare an array object

```
type SHORT_WORD is array (15 downto 0) of bit;
```

- define a one-dimensional array type **SHORT_WORD**
- **SHORT_WORD** has an integer index with range **(15 downto 0)**
- **SHORT_WORD** is actually a bit_vector of size 16

2.17 Arrays

DATA_WORD is initialized (by default) to all '0' bits

signal	DATA_WORD:	SHORT_WORD;
variable	ALT_WORD:	SHORT_WORD := "0101010101010101";
constant	ONE_WORD:	SHORT_WORD := (others => '1');

ALT_WORD(0)

accesses the **rightmost** bit of ALT_WORD

ALT_WORD(5 **downto** 0)

accesses the low-order 6 bits of ALT_WORD

2.17 Arrays

General forms of array type and array object declaration

```
type array_type_name is array index_range of element_type;  
signal array_name: array_type_name [ := initial_values];
```

signal may be replaced with variable or constant

2.17.1 Matrices

Multidimensional array

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;  
variable matrixA: matrix4x3 := ((1,2,3), (4,5,6), (7,8,9),  
                                (10,11,12));
```

matrixA(3,2)

element in 3rd row and 2nd column,
which has a value 8

2.17.1 Matrices

Unconstrained array type

```
type intvec is array (natural range<>) of integer;
```



range **must be specified**
when array object is declared

```
signal intvec5: intvec (1 to 5) := (3, 2, 6, 8, 1);
```

2.17.1 Matrices

Two-dimensional array type with unconstrained row and column index ranges

```
type matrix is array (natural range<>, natural range<>) of  
    integer;
```

Example

- ❑ **Parity** bits are often used in digital communication for error detection and correction
- ❑ The simplest of these involve transmitting one additional bit with the data, a parity bit
- ❑ Use VHDL arrays to represent a parity generator that generates a **5-bit-odd-parity** generation for a 4-bit input number using the look-up table (LUT) method

Predefined unconstrained array types in VHDL

```
type bit_vector is array (natural range<>) of bit;  
type string is array (positive range<>) of character;
```

```
constant string1: string(1 to 29) :=  
    "This string is 29 characters."
```

The characters in a string literal must be enclosed in double quotes

```
constant A: bit_vector(0 to 5) := "101011"
```

```
constant A: bit_vector(0 to 5) := ('1','0','1','0','1','1')
```

Subtype

```
subtype SHORT_WORD is bit_vector (15 downto 0);
```

After a type has been declared, a related **subtype** can be declared to include a subset of the values specified by the type

Predefined subtypes of type integer	
positive	All positive integers
natural	All positive integers and 0

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.18 Loops in VHDL (循环语句)

1. infinite loop (无限循环语句)

```
[loop-label:] loop  
    sequential statements  
end loop [loop-label];
```

A loop statement is a **sequential** statement (only in process, function, or procedure)

```
exit;  
exit when condition;
```

loop will terminate when the **exit** statement is executed, provided that the condition is **TRUE**

2. for loop

```
[loop-label:] for loop-index in range loop  
    sequential statements  
end loop [loop-label];
```

- ❑ **loop-index** is automatically defined when the loop is entered, and it should not explicitly be declared
- ❑ It is initialized to the first value in the range and then the sequential statements are executed

2. for loop

```
[loop-label:] for loop-index in range loop  
    sequential statements  
end loop [loop-label];
```

loop-index can be used inside the loop

loop-index **CANNOT** be changed

2. for loop

```
[loop-label:] for loop-index in range loop  
    sequential statements  
end loop [loop-label];
```


- ❑ When the end of the loop is reached, the loop-index is set to the next value in the range and the sequential statement are executed again
- ❑ This process continues until the loop has been executed for every value in the range, and the loop terminates
- ❑ After the loop terminates, the loop-index is **no longer** available

For loop Example: 4-bit adder

i will be initialized to 0 when the for loop is entered

i is not a variable and is therefore not declared in the process declarative part

```
loop1: for i in 0 to 3 loop  
    cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);  
    sum(i) := A(i) xor B(i) xor cin;  
    cin := cout;  
end loop loop1;
```



The carry out from one iteration (cout) is copied to the carry in (cin) before the end of the loop

3. while loop

```
[loop-label:] while condition loop  
    sequential statements  
end loop [loop-label];
```

The “**loop index**” can be manipulated by the programmer

- ❑ As in **while loops** in most languages, a **condition** is tested before each iteration
- ❑ The loop is terminated if the condition is false

3. while loop

-- Down counter

```
while stop = '0' and count /=0 loop
    wait until clk'event and clk = '1';
    count <= count - 1;
    wait for 0 ns;
end loop [loop-label];
```

The counter is decremented on $\text{clk}\uparrow$ until either $\text{count}='0'$ or $\text{stop}='1'$

IMPORTANT:

- **IF, WAIT, CASE, LOOP** are intended exclusively for sequential code
- They can only be used inside a **PROCESS** (进程) , **FUNCTION** (函数) or **PROCEDURE** (过程)

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.19 Assert and Report Statements

assert statement (断言)

If **boolean-expression** is **false**, an assertion violation has occurred

```
assert boolean-expression  
      report string-expression  
      [severity severity-level;]
```

If an assertion violation occurs during simulation, the simulator reports it with the **string-expression** provided in the report clause

2.19 Assert and Report Statements

assert statement

```
assert boolean-expression  
      report string-expression  
      [severity severity-level;]
```

Four possible severity-level:

- note 注意
- warning 警告
- error 错误
- failure 失败

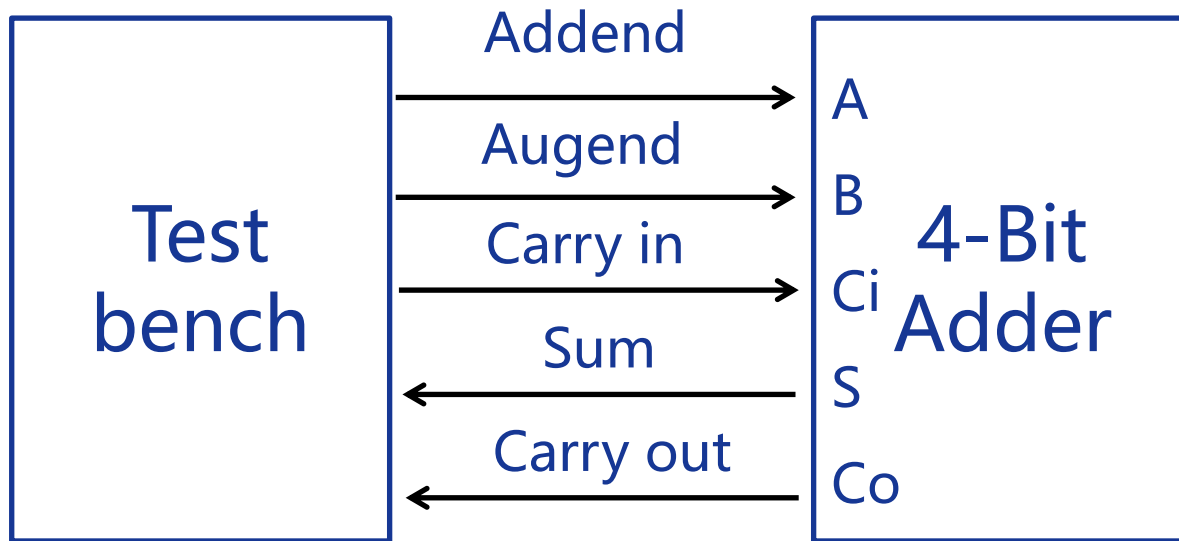
2.19 Assert and Report Statements

```
report "ALL IS WELL";
```

If assert clause is omitted, then the report is always made

2.19 Assert and Report Statements

Example



The adder we are testing will be treated as a component and embedded in **test bench**

```
entity TestAdder is  
end TestAdder;
```

```
architecture test1 of TestAdder is  
component Adder4
```

```
    port(A, B: in bit_vector ( 3 downto 0 ); Ci: in bit;  
          S: out bit_vector (3 downto 0); Co: out bit);
```

```
end component;
```

Adder4 is a
component

```
constant N: integer := 11;
```

```
type bv_arr is array (1 to N) of bit_vector( 3 downto 0 );
```

```
type bit_arr is array (1 to N) of bit;
```

```
constant addend_array: bv_arr := ("0111", "1101", "0101", "1101",  
                                   "0111", "1000", "0111", "1000", "0000", "1111", "0000");
```

```
constant augend_array: bv_arr := ("0101", "0101", "1101", "1101",  
                                   "0111", "0111", "1000", "1000", "1101", "1111", "0000");
```

```
constant cin_array: bit_arr := ('0', '0', '0', '0', '0', '0', '1',  
                                '0', '1', '1', '0',);
```

```
constant sum_array: bv_arr := ("1100", "0010", "0010", "1010",  
                               "1111", "1111", "1000", "0000", "1110", "1111", "0000");
```

```
constant cout_array: bit_arr := ('0', '1', '1', '1', '0', '0', '0',  
                                 '1', '0', '1', '0',);
```

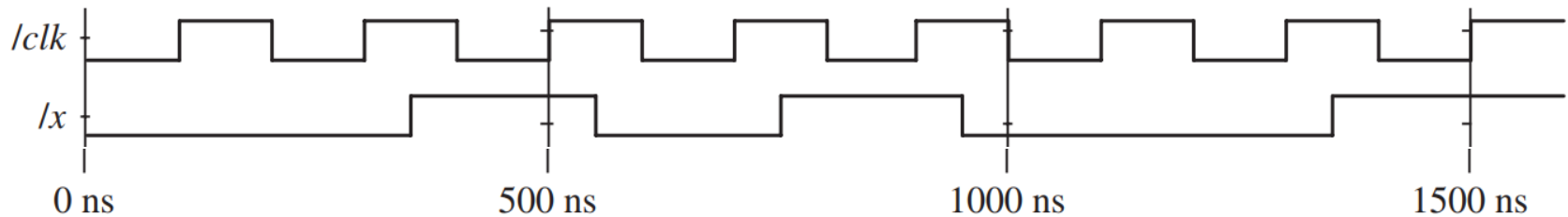
```
signal addend, augend, sum: bit_vector ( 3 downto 0 );
```

```
signal cin, cout: bit;
```

The test bench code uses constant arrays to define the test inputs for the adder and the expected outputs

2.19 Assert and Report Statements

```
begin
  process
    begin
      for i in 1 to N loop
        addend <= addend_array(i);
        augend <= augend_array(i);
        cin <= cin_array(i);
        wait for 40 ns;
        assert (sum = sum_array(i) and cout = cout_array(i))
          report "Wrong Answer"
          severity error;
      end loop;
      report "Test Finished";
    end process;
  add1: adder4 port map (addend, augend, cin, sum, cout);
  and test1;
```



```
add wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

We used simulator commands to generate waveform inputs (Chp2.15)

How a waveform input can be provided in a test bench

Generating a test sequence to test Code Converter

```
entity test_code_conv is  
end test_code_conv;
```

```
architecture tester of test_code_conv is
```

```
signal X, CLK Z: bit;
```

```
component Code_Converter is
```

```
port(X, CLK: in bit;
```

```
      Z: out bit);
```

```
end component;
```

```
begin
```

```
  clk <= not clk after 100 ns;
```

```
  X <= '0', '1' after 350 ns, '0' after 550 ns, '1' after  
      750 ns, '0' after 950 ns, '1' after 1350 ns;
```

```
  CC: Code_Converter port map (X, clk, Z);
```

```
end tester;
```

A time-varying signal is provided to input X