

# VHDL

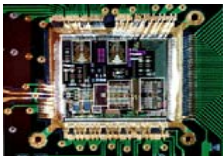
Instructor: Dr. Saraju P. Mohanty

In this Lecture

- Test Bench and Simulation
- Scalar Data Types and Operations
- Sequential Statements
- Composite Data Types and Operations

*Proverb: If we hear, we forget; if we see, we remember; if we do, we understand. So, implement designs !!*

**Note:** The slides are from either text book or reference book authors or publishers.

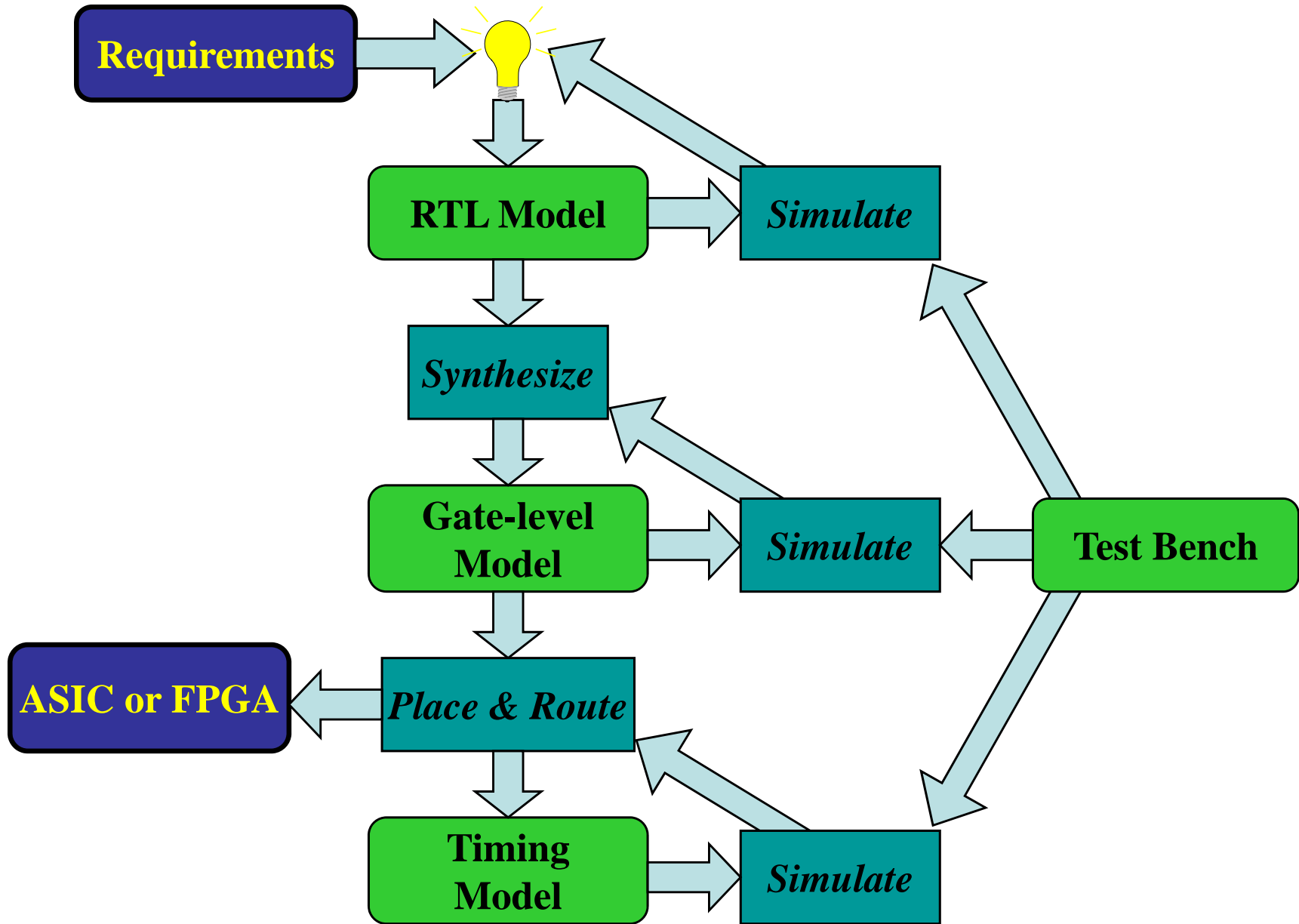


# Synthesis

- Translates register-transfer-level (RTL) design into gate-level netlist
- Restrictions on coding style for RTL model
- Tool dependent: A subset of RTL is synthesizable depending on the tool

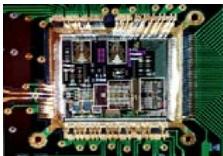


# Basic Design Methodology



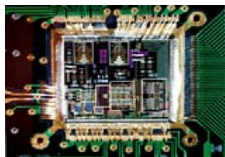
# Signal Assignment and Wait

- Process statements consist of sequential statements very similar to other high-level languages like C, C++ etc.
- Two special kinds of statement are:
  - Signal assignment (modeling carriers)
  - Wait statement (modeling event response)



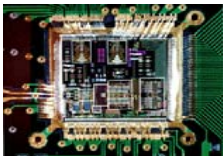
# Simulation

- It refers to the execution of modules in response to the stimuli from the test bench.
- Discrete Event Simulation
  - Based on “events” that occur on signals
  - An event is a change in the value of the signal
  - Simulation is only done of those modules that have the effected signal as input.



# Simulation

- A “typical” process:
  - Waits (suspended) for events on its input signals.
  - Process is said to be sensitive to those signals.
  - Such signals are specified through wait statements.
  - When an event occurs on any one signal the process resumes
    - It executes all sequential statements until the next wait statement and
    - Suspends on reaching the wait statement.



# Simulation

- A process executes sequential statements that include signal assignment statements.
- In contrast to all other sequential assignment statements, a signal assignment is not effected until the next wait statement.
- During execution a process is said to schedule a transaction on a signal.
- The transaction is actually processed at the next wait statement.



# Simulation Algorithm

- Two step algorithm with a
  - Initialization phase and
  - Repeated execution of the simulation cycle





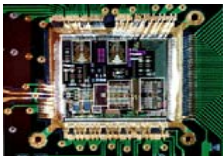
# Simulation Algorithm

- Initialization phase
  - each signal is given its initial value
  - simulation time set to 0
  - for each process
    - activate
    - execute until a wait statement, then suspend
      - execution usually involves scheduling transactions on signals for later times



# Simulation Algorithm

- Simulation cycle
  - advance simulation time to time of next transaction
  - for each transaction at this time
    - update signal value
      - **event if new value is different from old value**
  - for each process sensitive to any of these events, or whose “wait for ...” time-out has expired
    - resume
    - execute until a wait statement, then suspend
- Simulation finishes when there are no further scheduled transactions



# Signal Assignment

- A process executes sequential statements that include signal assignment statements.
- In contrast to all other sequential assignment statements, a signal assignment is not effected until the next wait statement.
- During execution a process is said to schedule a transaction on a signal.
- The transaction is actually processed at the next wait statement.



# Simulation Example

```
architecture behav of top is
signal x,y,z : integer := 0;
begin
  p1 : process is
    variable a, b : integer := 0;
    begin
      a := a + 20;
      b := b + 10;
      x <= a + b after 10 ns;
      y <= a - b after 20 ns;
      wait for 30 ns;
    end process;

  p2: process is
    begin
      z <= (x + y);
      wait on x,y;
    end process;
end behav;
```

	0 ns	10 ns	20 ns	30 ns
a	20	20	20	40
b	10	10	10	20
x	0	30	30	30
y	0	0	10	10
z	0	30	40	40



# Test Bench ??

- Test benches emulate a hardware breadboard in which we place our designed chip for purpose of verification.
- A test bench applies some sequence of inputs to the circuit being tested (the Design Under Test, or DUT) so that its operation can be observed in simulation.
- Waveforms are typically used to represent the values of signals in the design at various time instances.
- A test bench must consist of a component declaration corresponding to the DUT, and a description of the input stimulus being applied to the DUT.

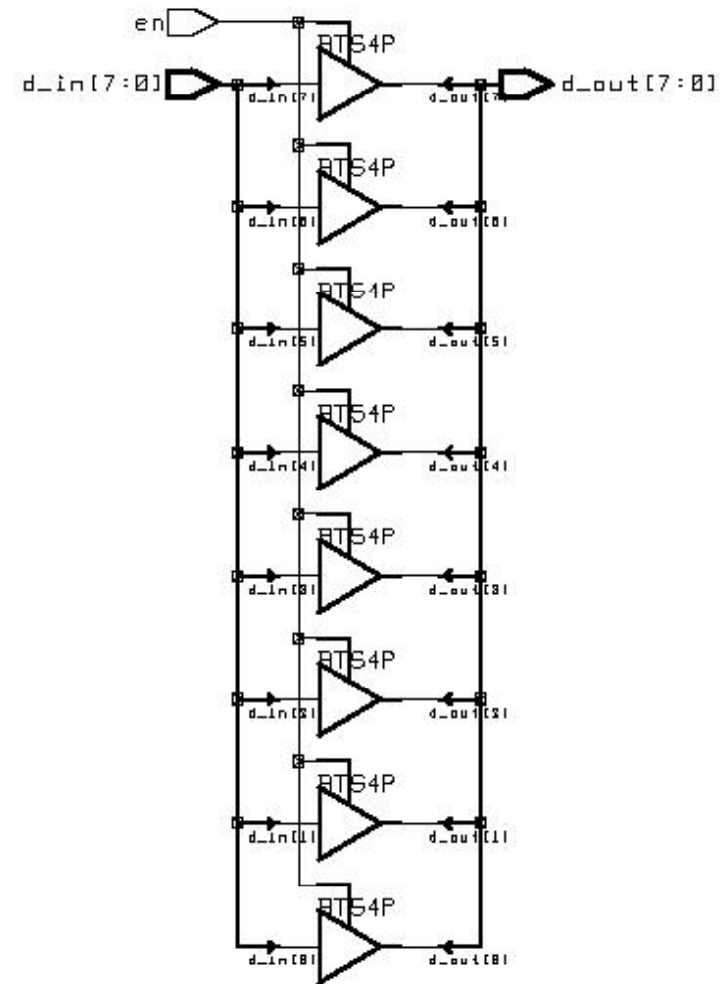


# Design and Test Bench Example: Design

```
entity tristate_dr is
  port( d_in: in std_logic_vector(7 downto
    0); en: in std_logic; d_out: out
    std_logic_vector(7 downto 0) );
end tristate_dr;
```

```
architecture behavior of tristate_dr is
begin
  process(d_in, en)
  begin
    if en='1' then
      d_out <= d_in;
    else
      d_out <= "ZZZZZZZZ";
    end if;
  end process;
end behavior;
```

Source: <http://www.cs.ucr.edu/content/esd/labs/tutorial/>

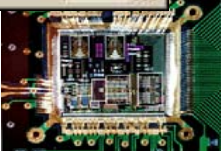
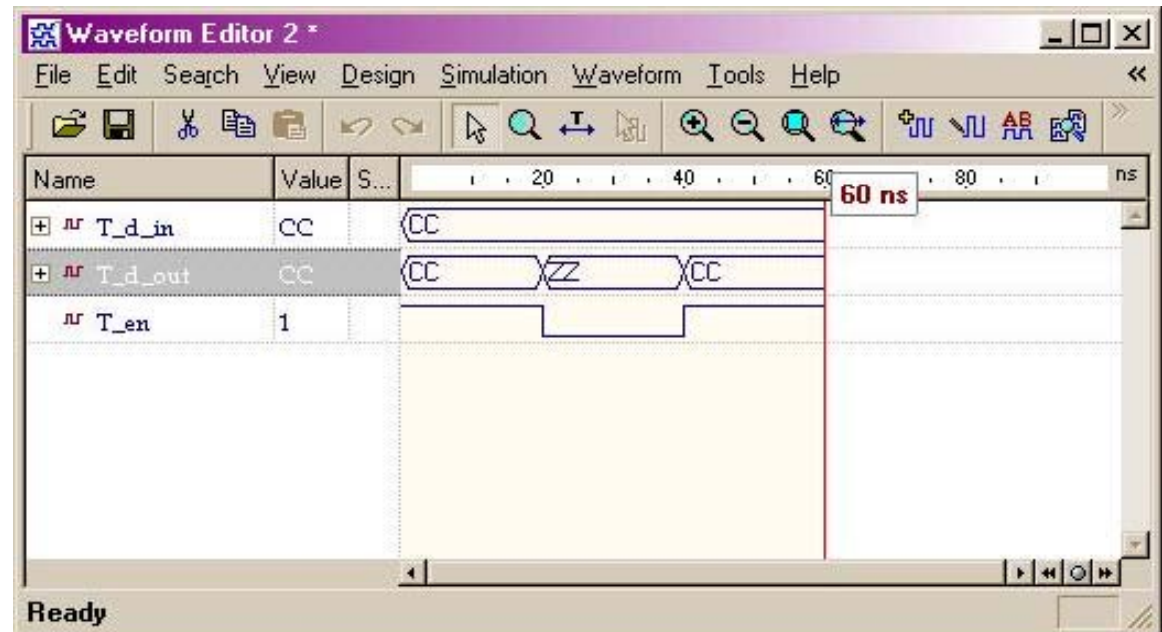


Schematic



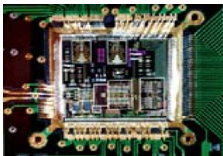
# Design and Test Bench Example: Test Bench

```
entity TB_tridr is
end TB_tridr;
architecture TB of TB_tridr is
    component tristate_dr is
        port( d_in: in std_logic_vector(7 downto 0); en: in std_logic; d_out:
              out std_logic_vector(7 downto 0) );
    end component;
    signal T_d_in, T_d_out: std_logic_vector(7 downto 0); signal T_en: std_logic;
begin
    DUT: tristate_dr port map (T_d_in, T_en, T_d_out);
    process
    begin
        T_d_in <= "11001100";
        T_en <= '1';
        wait for 20 ns;
        T_en <= '0'; wait for 20 ns;
        T_en <= '1'; wait for 10 ns;
        wait;
    end process;
end TB;
```



# Lexical Elements

- Lexical Elements of VHDL: comments, identifiers, special symbols, numbers, characters, strings, and bit strings.
- Comments
  - A comment line in VHDL is represented by two successive dashes “—”.
  - A comment extends from “—” to the end of the line.
- Identifiers
  - Identifiers are names that can be given by the user.
  - rules:
    - must start with an alphabetic letter.
    - can contain alphabetic letters, decimal digits and underline character “\_”.
    - cannot end with “\_”.
    - cannot contain successive “\_”.





# Lexical Elements ....

- Numbers
  - Two kinds: integer and real.
  - Both integer and real literals can be written in exponential notation, eg, 46E5, 1.34E5.
  - Both can be expressed in integer base between 2 and 16:
    - The number is written enclosed in “#” and preceded by the base
    - For example, 8 => 2#100#, 1024 => 2#1#E10
  - Long numbers for easier readability can include “\_” as separator. For example, 123\_456.



# Lexical Elements ....

- Characters

- written by enclosing it in single quotation marks.
- 'A', 'z'.

- Strings

- written by enclosing in double quotation marks.
- "abcdefg", "123456"
- concatenation operator "&".  
"abc" & "def" => "abcdef".

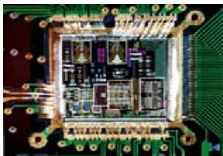
- Bit strings

- string of binary, octal or hexadecimal digits
- enclosed in double quotation marks and preceded by base
- B"10000", O"20", X"10"



# Lexical Elements ....

- Special symbols
  - operators: +, - , \*, /, &
  - real number and record: .
  - string, bit string and character delimiters: “, #, ‘
  - lexical delimiters: , ; :
  - precedence specifiers: ()
  - array indices: []
  - relational: =, >, <
  - two character symbols: :=, =>, /=, >=, <=, \*\*
- Reserve words
  - will be addressed while discussing language constructs

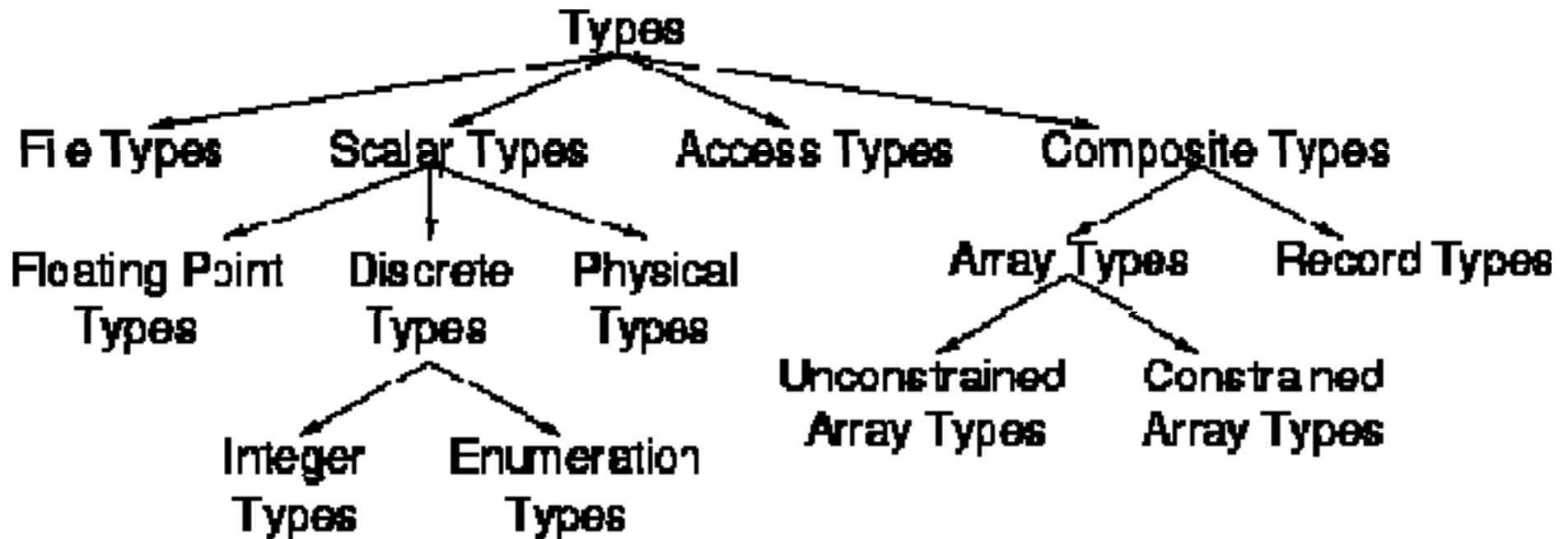


# Data Types

- VHDL is a strongly typed language; each objects assumes values of its declared type.
- Object is a named item that has a specified type.
- Four classes of objects: constants, variables, signals, and files.
- The “type” determines the operations that can be applied to the name.
- VHDL along with its packages provides pre-defined types.
- Additionally the user can define new types.
- A scalar type consists of single, indivisible values.



# VHDL Types: A Classification



Source: <http://www.eecs.uc.edu/~paw/tyvis/doc/node7.html>



# Constant Declaration

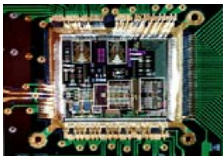
Constants are used for giving a name to a literal.

Syntax:

*constant\_decl* <=  
constant *id* { ,... } : *subtype\_indication* [ := *expr* ] ;

Example:

constant number\_of\_bytes: integer := 4;  
constant size, count: integer := 255;



# Variable Declaration and Assignment

Variables act as placeholders for quantities that change during simulation.

## Declaration

Syntax:

```
variable_decl <=  
variable id { ,... } : subtype_indication [ := expr ] ;
```

Example:

```
variable index, sum : integer := 0;
```

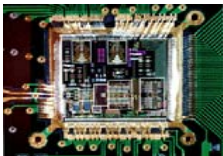
## Assignment

Syntax:

```
variable_assign <= [label : ] id := expr ;
```

Example:

```
pc := 1;  
index := index + 1;
```



# User defined type

Useful when pre-defined types are insufficient.

Syntax:

*type\_decl* <= type *identifier* is *type\_defn* ;

Example:

type apples is range 0 to 100;  
type oranges is range 0 to 100;

Default value is left hand side of range.





# Integer type

“integer” is a pre-defined type used to represent whole numbers.

Example:

```
variable x, y : integer ;
```

VHDL standard requires that the implementation be able to represent numbers from  $-2^{31} + 1$  to  $2^{31} - 1$ .

User can define new “integer” types.

Syntax:

```
type_decl <= type identifier is int_type_defn ;  
int_type_defn <= range expr ( to | downto ) expr
```

Example:

```
type month is range 1 to 12 ;  
type count_down is range 10 downto 0;
```



# Integer type : operations

- Addition: +
- Subtraction or negation: -
- Multiplication: \*
- Division: /
- Modulo: mod

$$a = b * n + (a \text{ mod } b), \text{ sign of } b, n: \text{ integer}$$
$$(-5) \text{ mod } 3 = 1$$

- Remainder: rem

$$a = (a/b) * b + (a \text{ rem } b), \text{ sign of } a$$
$$(-5) \text{ rem } 3 = 1$$

- Absolute value: abs
- Exponentiation: \*\*
- Logical: =, /=, <, >, <=, >=



# Floating-point type

“real” is a pre-defined type used to represent floating-point numbers.

Example:

```
variable x, y : real ;
```

Similar to integers the user can also define new real types with limited range.

```
type temp is range -273.0 to 1000.0 ;
```



# Floating point type: operations

- Addition: +
- Subtraction or negation: -
- Multiplication: \*
- Division: /
- Absolute value: abs
- Exponentiation: \*\*
- Logical: =, /=, <, >, <=, >=



# Physical type

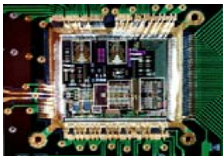
User defined types for mass, length, current etc.

Syntax:

```
phy_type_defn <=  
    type type_id is range expr ( to / downto ) expr  
    units unit_id ;  
    { id = phy_literal ; }  
end units [ type_id ] ;
```

Example:

```
type distance is range 0 to 1E9  
units  
    mm;  
    m = 1000 mm;  
    km = 1000 m;  
end units distance;
```



# Physical type: operations

- Addition: +
- Subtraction or negation: -
- Multiplication by integer or real: \*
- Division by integer or real: /
- Absolute value: abs
- Exponentiation: \*\*
- Logical: =, /=, <, >, <=, >=



# Time type

- Predefined physical type.

type time is range *implementation defined*

units

fs;

ps = 1000 fs;

ns = 1000 ps;

us = 1000 ns;

ms = 1000 us;

sec = 1000 ms;

min = 60 sec;

hr = 60 min;

end units;



# Enumerated types

- Useful for giving names to a values of an object (variable or signal).

type alu\_func is (disable, pass, add, sub, mult, div);





# Predefined Enumerated types

- Character

type character is ( 'a', 'b', 'c', ..... );

Operations: =, /=, <, >, <=, >=

- Boolean

type boolean is ( false,true );

Operations: and, or, nand, nor, xor, xnor, not,  
=, /=, <, >, <=, >=



# Bit type

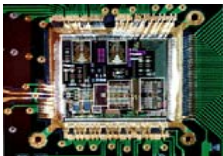
- Bit is also a predefined enumerated type
  - type bit is ('0', '1');

- Operations

Logical: =, /=, <, >, <=, >=

Boolean: and, or, nand, nor, xor, xnor, not

Shift: sll, srl, sla, sra, rol, ror



# Subtypes

- Sub types are useful for limiting the range of base type

```
type month is 1 to 31;  
subtype working_day is 1 to 3;
```

```
variable x,y : month;  
variable z : working_day;
```

```
y = x + z;
```



# Scalar Type Attributes (all)

- T'left : Left most value of T
- T'right : Right most value of T
- T'low : Least value of T
- T'high : Highest value of T
- T'ascending : true if T is ascending, false otherwise
- T'image(x) : A string representing the value of x
- T'value(s) : The value in T that is represented by s.



# Scalar Type Attributes: Example

type set\_index is range 21 downto 11;

```
set_index'left = 21  
set_index'right = 11  
set_index'low = 11  
set_index'high = 21  
set_index'ascending = false  
set_index'image(14) = "14"  
set_index'value("20") = 20
```



## Scalar attributes (discrete)

- Discrete types are integer and all enumerated types.

$T'pos(x)$ : position of  $x$  in  $T$

$T'val(n)$ : value in  $T$  at position  $n$

$T'succ(x)$ : successor of  $x$  in  $T$

$T'pred(x)$ : predecessor of  $x$  in  $T$

$T'leftof(x)$ : value in  $T$  at position one left of  $x$

$T'rightof(x)$ : value in  $T$  at position one right of  $x$



## Scalar attributes (discrete): Example

type logic\_level is (unknown, low, undriven, high);

logic\_level'pos(unknown) = 0

logic\_level'val(3) = high

logic\_level'succ(unknown) = low

logic\_level'pred(undriven) = low



# The VHDL Operators' Precedence

	Operator Class	Operator
Highest precedence	Miscellaneous	<b>**</b> , ABS, NOT
	Multiplying	<b>*</b> , <b>/</b> , MOD, REM
	Sign	<b>+</b> , <b>-</b>
	Adding	<b>+</b> , <b>-</b> , <b>&amp;</b>
	Shift	SLL, SRL, SLA, SRA, ROL, ROR
	Relational	<b>=</b> , <b>/=</b> , <b>&lt;</b> , <b>&lt;=</b> , <b>&gt;</b> , <b>&gt;=</b>
Lowest precedence	Logical	AND, OR, NAND, NOR, XOR, XNOR

