Attributes

- **An attribute is a value, function, type, range, signal, or a constant that may be associated with one or more names within a VHDL description.**

- **Predefined attributes are divided into 5 classes:**
  - **Value attributes: return a constant value**
  - **Function attributes: call a function that return value**
  - **Signal attributes: create a new implicit signal**
  - **Type attributes: return a type**
  - **Range attributes: return a range**

- ***Rationale*: Attributes creates code that is easier to maintain**

Attributes

- **NB! Synthesis packages accept only the following attributes:**
  'base, 'left, 'right, 'high, 'low,
  'range, 'reverse_range, 'length, 'stable, and 'event.

```
-- type myArray is array (9 downto 0) of any_type;
-- variable an_array : myArray;
-- type fourval is ('0', '1', 'Z', 'X');
-- signal sig: sigtype;
-- constant T: time := 10 ns;
```

Attributes

Attribute	Result type	Result
myArray'high|left|low|right	integer	9|9|0|0
myArray'ascending	boolean	false
an_array'length|range|reverse_range	integer	10 | 9 downto 0 | 0 to 9
fourval'leftof('0')|rightof('1')	fourval	error|'Z'
fourval'pos('Z')	integer	2
fourval'pred('1')|succ('Z')|val(3)	fourval	'0'|'X'|'X'
sig'active	boolean	True if activity on sig
sig'delayed(T)	sigtype	Copy of sig delayed by T
sig'driving_value	sigtype	Value of driver on sig
sig'event	boolean	True if event on sig
sig'last_active|last_event	time	Time since last (activity | event)
sig'last_value	sigtype	Value before last event
sig'quiet(T)|stable(T)	boolean	(Activity|event) (now -T) to now
sig'transaction	bit	Toggles on activity on sig

VHDL'93

- **New keywords: allow, element, group, impure, inertial, literal, postponed, private, pure, reject, rol, ror, shared, sla, sll, sra, srl, unaffected, xnor**
- **Arithmetic operations: rol, ror, sla, sll, sra, srl, xnor**
- **Function types: pure, impure**
- **Processes: postponed**
- **Attributes: literal**
- **Variables: shared**
- **Objects: private, allow**
- **Items grouping: group**
- **Hierarchical path names: '/' and '.'**
- **New attributes: foreign, ascending[(N)], image(x), value(x), driving, driving_value, path_name, simple_name**
- **Generalized aliases**
- **Files as separate class of objects: file_open(), file_close()**
- **Function 'now'**
- **Overloading**

VHDL timing

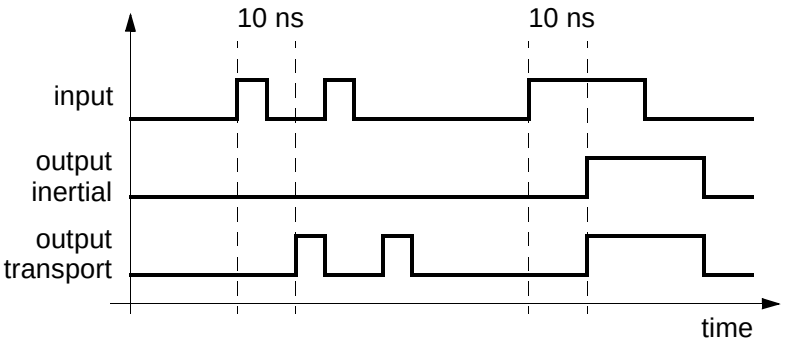
- **Signal attributes.**
- **The “wait” statement:**
  - **delta time**
  - ***wait on* sensitivity list**
  - ***wait until* condition**
  - ***wait for* time_expression**
- **Simulation Engine.**
- **Modeling with Delta Time Delays.**
- **Inertial/transport delay.**

Timing control

- **Postponing a signal assignment -- “... after T;”**
- **Sensitivity list**
- **Wait commands**
  - **wait for a signal event -- wait on x;**
  - **wait for a condition -- wait until x=’1’;**
  - **wait certain time -- wait for 20 us;**
  - **wait (forever) -- wait;**
  - **combined use -- wait on clk until clk=’1’ and ready=’1’ for 1 us;**
  - **wait until sensitivity**
    - **wait on a until a=’1’ and b=’0’; -- sensitive to signals *a* only**
    - **wait until a=’1’ and b=’0’; -- sensitive to signals *a* and *b***

Inertial and transport delays

```
output <= input after 10ns;                    -- VHDL'87
output <= [inertial] input after 10ns;         -- VHDL'93
output <= transport input after 10ns;
```



Structuring a Design

- **Motivation**
  - **models are easier to read**
  - **submodels can be reused**
  - **design and verification are more manageable**

structural granularity	structural modelling unit	VHDL construct
coarse	entity / architecture pairing	configuration
coarse	primary design unit	entity / architecture
coarse/medium	replication of concurrent statements	for / if - generate
coarse/medium	grouping of concurrent statements	block
medium	grouping of sequential statements	process
fine	subprogram	procedure / function

Partitioning features

- **Modularity features:**
  - **Procedures**
  - **Functions**
- **Partitioning features:**
  - **Blocks**
  - **Packages**
  - **Libraries**
  - **Components**
  - **Configurations**

Elements of Entity / Architecture

- **VHDL Entity (declarations, generic and port clauses, etc.)**
- **VHDL Architecture (declarations, statements)**
  - **Process statement**
  - **Concurrent signal assignment**
  - **Component instantiation statement**
  - **Concurrent procedure call**
  - **Generate statement**
  - **Concurrent Assertion Statement**
  - **Block statement**

Process - behavioral description

- entity / architecture / component
  - structural elements

```
entity_declaration ::=
  entity identifier is
    entity_header
    entity_declarative_part
    [begin
      entity_statement_part]
  end [entity] [identifier] ;

entity_statement ::=
  concurrent_assertion_statement
  | passive_concurrent_procedure_call
  | passive_process_statement
```

- process
  - behavior of the model
  - contains timing control
  - concurrent statement (data-flow statement) == process with sensitivity list

Equivalent processes

- Data-flow statement

```
x <= a and b after 5 ns;
```

- Equivalent processes
  - #1

```
process ( a, b ) begin
  x <= a and b after 5 ns;
end process;
```

  - #2

```
process begin
  wait on a, b;
  x <= a and b after 5 ns;
end process;
```

  - #3

```
process
  variable tmp: bit;
begin
  wait on a, b;
  tmp := a and b;
  wait for 5 ns;
  x <= tmp;
end process;
```

Process

- **Sensitivity list**

```
process ( a, b ) begin
  x <= a and b after 5 ns;
end process;
```

```
process begin
  wait on a, b;
  x <= a and b after 5 ns;
end process;
```

- **Timing control in the beginning or in the end?**

```
process begin
  wait on a, b;
  x <= a and b after 5 ns;
end process;
```

```
process begin
  x <= a and b after 5 ns;
  wait on a, b;
end process;
```


Conditional statements

- **if-then-else**

```
if conditional-expression then statements...
elsif conditional-expression then statements...
else statements...
end if;
```

  - *conditional-expression* - must return boolean value

- **case**

```
case expression is
when constant-value [| constant-value] => statements...
when others => null
end case;
```

Loops

```
[label:] [iteration-method] loop
  statements...
end loop [label];

iteration-method ::=
    while conditional-expression | for counter in range

exit [label] [when conditional-expression];
next [label] [when conditional-expression];

range ::=   expression to expression |
            expression downto expression |
            type'range | ...
```


Loops

- for-loop

```
for I in my_array'range loop
  next when I<lower_limit;
  exit when I>upper_limit;
  sum := sum + my_array(I);
end loop;
```

- while-loop

```
while a<10 loop
  a := a + 1;
end loop;
```

Behavioral hierarchy

- Functions & procedures
- function
  - used as an expression
  - can not have timing control statements
  - input parameters only (as constants)
- procedure
  - used as a statement (both sequential and concurrent)
  - can contain timing control statements
  - input parameters (constants)
  - output parameters (variables/signals)

Functions & procedures

- Declaration (prototype)
  - package
  - declarative part of architecture, process, function, procedure, etc.
- Content (body)
  - package body
  - declarative part of architecture, process, function, procedure, etc. (together with declaration)

Functions

```
-- ...
function conv_boolean (a: signed) return boolean is begin
  if to_bit(a(a'low))='1' then return TRUE; else return FALSE; end if;
end conv_boolean;
-- ...
function "and" (l,r: signed) return signed is begin
  return signed(std_logic_vector(l) and std_logic_vector(r));
end;
-- ...

-- ...
  signal a, b, x: signed (7 downto 0);
  signal y: boolean;
-- ...
  X <= a and b;
  -- ...
  y <= conv_boolean(a);
```


Procedures

```
PACKAGE adder_elements IS
--  full_adder :  1-bit full adder  (declaration)
PROCEDURE full_adder (CONSTANT a0, b0, c0: IN bit; VARIABLE o0, c1: OUT bit);
END adder_elements;

PACKAGE BODY adder_elements IS
PROCEDURE half_adder (CONSTANT a0, b0: IN bit; VARIABLE o0, c1: OUT bit) IS
BEGIN
  o0 := a0 XOR b0;    c1 := a0 AND b0;
END half_adder;

PROCEDURE full_adder (CONSTANT a0, b0, c0: IN bit; VARIABLE o0, c1: OUT bit) IS
  VARIABLE c_1, c_2, o_1: bit;
BEGIN
  half_adder ( a0, b0, o_1, c_1 );
  half_adder ( o_1, c0, o0, c_2 );
  c1 := c_1 or c_2;
END full_adder;
END adder_elements;
```

Blocks

- The blocks are used for enhancing readability
- Blocks can be nested
- Block can define local declarations
  - hiding same names declared outside of block

```
BLOCK1: block
    signal a,b: std_logic;
begin
    ...
end block BLOCK1;
```


Guarded blocks

- Special form of block declarations that include an additional expression known as *guard expression*
- Used for describing latches and output enables in dataflow style

```
architecture mylatch of latch is
begin
    L1: block (LE='1')
    begin
        Q    <= guarded D after 5 ns;
        QBar <= guarded not(D) after 7 ns;
    end block L1;
end mylatch;
```

- Remark: Guarded blocks are not supported by all synthesis tools

Packages

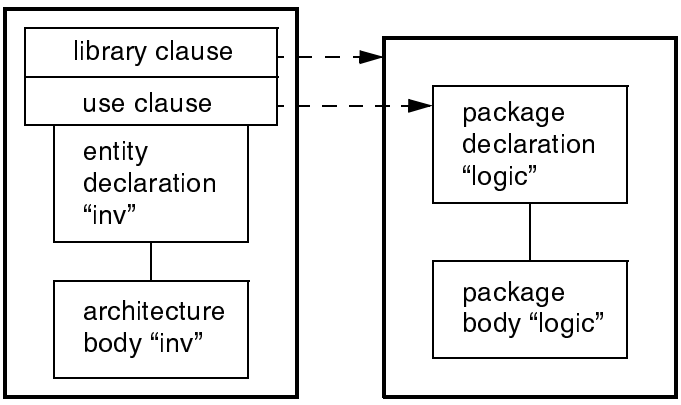
- Package declaration
- Package body (necessary in case of subprograms)
- Deferred constant (declared in package, assigned in package body)
- The “use” clause
- Signals in packages (global signals)
- Resolution function in packages
- Subprograms in packages

- Package TEXTIO

Library and use clauses

- A design library is an implementation-dependent storage facility for previously analyzed design units

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```



Components

- Components are used to connect multiple VHDL design units (entity/ architecture pairs) together to form a lager, hierarchical design
- The subcomponents of current design unit have to be declared in a declarative part of the architecture
- The components are *instantiated* in body part of architecture

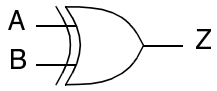
```
architecture toparch of topunit is
    component child1
        port( ... );
    end component;
    component child2 ...
begin
    COMP1: child1 port map(...);
    COMP2: child2 port map(...);
end toparch;
```


Structural replication

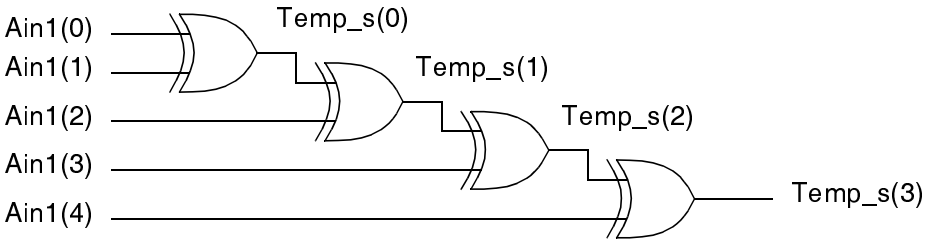
- A *generate* statement provides a mechanism for iterative or conditional elaboration of a portion of a description
- Typical application - instantiation and connecting of multiple identical components (half adders to make up full adder, trees of components etc.)

```
UK: for K_i in 0 to 3 generate
  UK0 : if K_i = 0 generate
    UXOR : XOR_Nty port map(A => Ain1(K_i),
                            B => Ain1(K_i+1),
                            Z => Temp_s(K_i)); end generate UK0;
  UK1_3: if K_i > 0 generate
    UXOR : XOR_Nty port map(A => Temp_s(K_i-1),
                            B => Ain1(K_i+1),
                            Z => Temp_s(K_i)); end generate UK1_3;
end generate UK;
```

Using generate statement




```
entity XOR_Nty is
  port ( A : in Std_Logic;
         B : in Std_Logic;
         Z : out Std_Logic );
end XOR_Nty;
```

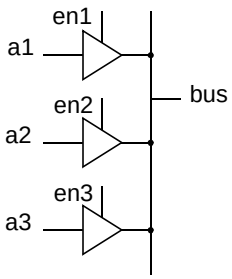


Subprograms

- **Subprogram definition**
- **Subprogram overloading**
- **Functions**
- **Resolution functions**
- **Operator overloading**
- **Concurrent procedure**

Drivers

- **Resolution Function**




```
type tri_value is ('0','1','Z');
type tri_val_vector is array (natural range <>) of tri_value;
function resolve_tri (b:tri_val_vector) return tri_value is ...
    ...
signal bus : resolve_tri tri_value;
```

- **Drivers - a sequence of pairs: (value,time)**
- **Ports**

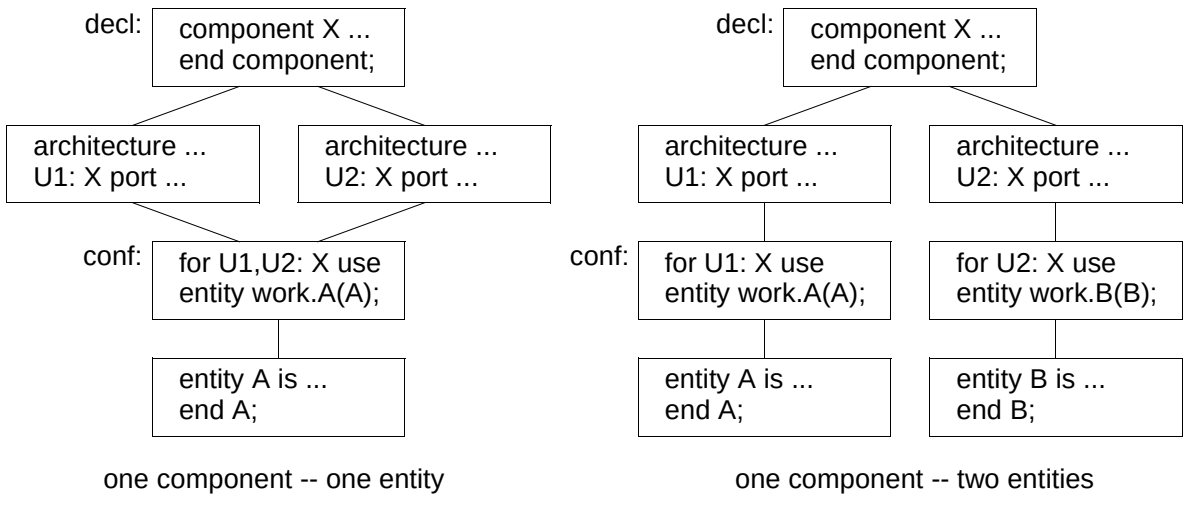
Resolution function -- I^{2}C

```
package I2C_defs is
  type I2C_bit is ( '0', 'Z', 'H' );
  type I2C_bit_vector is array (integer range <>) of I2C_bit;
  function resolved ( v: I2C_bit_vector ) return I2C_bit;
  -- ...
end I2C_defs;
package body I2C_defs is
  function resolved ( v: I2C_bit_vector ) return I2C_bit is
    variable r: I2C_bit := 'Z';
    type I2C_1d is array ( I2C_bit ) of I2C_bit;
    type I2C_2d is array ( I2C_bit ) of I2C_1d;
    constant resolution_table: I2C_2d := (
    ----------------------------
    -- '0'  'Z'  'H'
    ----------------------------
     ( '0', '0', '0' ),   -- '0'
     ( '0', 'Z', 'H' ),   -- 'Z'
     ( '0', 'H', 'H' ) ); -- 'H'
  begin
    for i in v'range loop    r := resolution_table ( r ) ( v(i) );    end loop;
    return r;
  end resolved;
  -- ...
end I2C_defs;
```

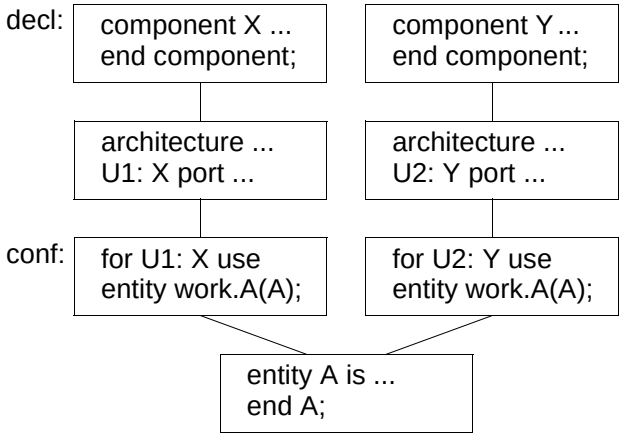
Configuration declaration

- **Configuration is a separate *design unit* which allows different architecture and component bindings to be specified after a model has been analyzed and compiled**
- **Two types**
  - **Configuration declaration**
    - **binds the *entity* to particular *architecture body***
    - **binds components, used in the specified *architecture* to a particular *entity***
    - **binds component statements, used in the specified *architecture*, to particular *configuration* statements**
  - **Configuration specification**
    - **used to specify the binding of component instances to a particular *entity-architecture* pair**

Using configuration - #1



Using configuration - #2



two components -- one entity

Using configuration - #3

```
configuration MEMO_BHV of TEST_TEST is
  for GENER
    for BFF: TEST_BUFF
        use entity work.TEST_BUFF(BEHAVIOUR);
    end for;
  end for;
end MEMO_BHV;
```

```
TEST_TEST(GENER)
  BFF: TEST_BUFF(BEHAVIOUR)
```

```
configuration MEMO_STR of TEST_TEST is
  for GENER
    for BFF: TEST_BUFF
        use entity work.TEST_BUFF(STRUCTURE);
      for STRUCTURE
        for BFF_STR: TEST_BUFF_syn
            use entity work.TEST_BUFF_syn(BEHAVIOUR_syn);
        end for;
      end for;
    end for;
  end for;
end MEMO_STR;
```

```
TEST_TEST(GENER)
  BFF: TEST_BUFF(STRUCTURE)
    BFF_STR: TEST_BUFF_syn(BEHAVIOUR_syn)
```

Sequential and concurrent structures

sequential	concurrent
if, case	
loop, next, exit	
null	
wait	
return	
assertion	concurrent assertion
procedure-call	concurrent procedure-call
signal assignment	concurrent signal assignment
	component instantiation
	generate
	process

Concurrent elements in architecture