



Gallop cross VHDL



Agenda

Design units

`entity`, `architecture`, `configuration`, `package`

Lexical elements

literals, identifiers, objects, expressions

Structural description

`map`, `generate`

Sequential statements

`process`, `wait`, `if`, `case`, `loop`, `next`, `exit`, `assert`, `function`, `procedure`

Concurrent statements

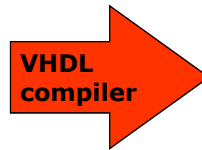
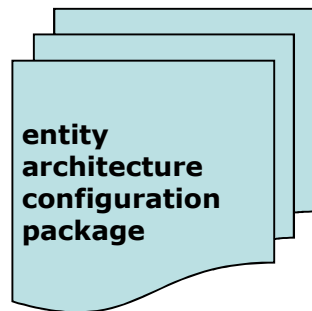
assignments: `unconditional`, `conditional` and `selected`, `subprograms`, `block`

Composite types

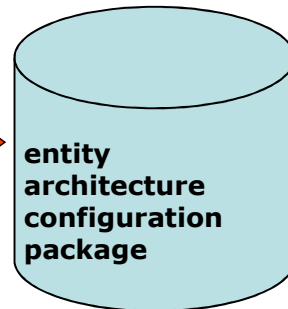
arrays: `one-dimensional` and `multidimensional`



Design units

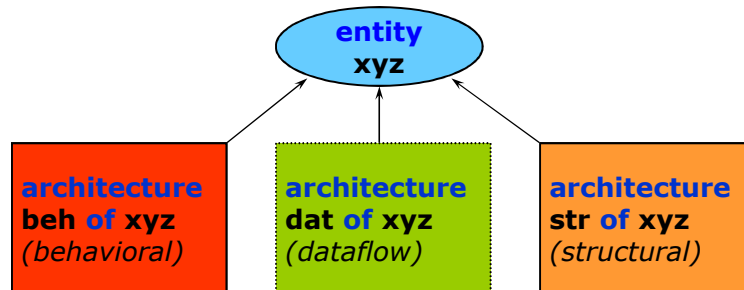


Library



```
-- entity declaration defines the interface between
-- a given design entity and its environment
entity ENTITY_NAME is
  port (
    < PORT_NAME: <mode> <type>; >
  );
end ENTITY_NAME;

-- an architecture body specifies the relationship
-- between the inputs and outputs of a design entity
architecture ARCH_NAME of ENTITY_NAME is
begin
  <statements>
end ARCH_NAME;
```



```
-- comparator example
```

```
entity COMPARE is
  port (A,B: in bit;
        C: out bit);
end COMPARE;
```

```
architecture arch_behavioral of COMPARE is
begin
  process (A,B) -- A,B - signals "active" in process
  begin
    -- sequential assignment operators <=
    if (A=B) then
      C <= '1' after 1 ns;
    else
      C <= '0' after 1 ns;
    end if;
  end process;
end arch_behavioral;
```

Set of sequential operations, describing the behaviour of a model.



Design units

architecture statement – *dataflow style*

```
architecture arch_dataflow of COMPARE is
begin -- concurrent assignment operators <=
  C <= not (A xor B) after 1 ns;
end arch_dataflow
```

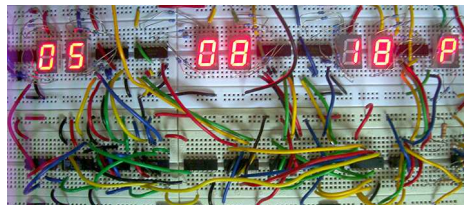
Set of concurrent assignments, describing the flow of data.



Design units

architecture statement – *structural style*

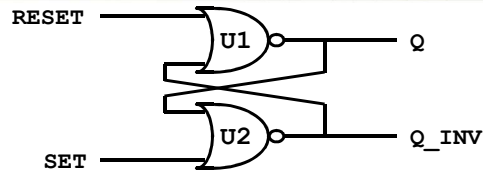
```
architecture arch_structural of COMPARE is
signal I: bit;
  component XOR2 port (x,y: in bit; z: out bit);
  end component;
  component INV port (x: in bit; z: out bit);
  end component;
begin
  U0: XOR2 port map (A,B,I);
  U1: INV port map (I,C);
end arch_structural;
```



Set of connected modules, describing the structure of a model.



Structural description



```
architecture STRUCT of RS_FLOP is
-- component declaration
  component NOR2 port (A,B: in bit; X: out bit);
  end component;
begin
-- component instantiation
  U1: NOR2 port map (RESET,Q_INV,Q);
  U2: NOR2 port map (Q,SET,Q_INV); -- positional association
end STRUCT;

-- or named association, eg:
-- U1: NOR2 port map (A => RESET, X => Q, B => Q_INV);
```



Design units configuration statement

- allows choosing one of the architectures for a given **entity**
- provides a convenient way of documenting the project versions
- eliminates the need to recompile the entire project when you need to change only a few components

```
configuration TESTBENCH_FOR_top of top_tb is
  for TB_ARCHITECTURE
    for UUT : top
      use entity work.top (structure);
    end for;
  end for;
end TESTBENCH_FOR_top;
```

Annotations for the configuration statement:

- name of configuration: TESTBENCH_FOR_top
- entity being configured: top_tb
- architecture being configured: TB_ARCHITECTURE
- component being configured: UUT
- name of library: work
- architecture chosen: top



Design units package statement

Groups together the common: declarations, subprograms, components or types.

```
package my_constans is
  constant unit_delay: time := 1 ns;
end my_constans;

Y <= '0' after work.my_constans.unit_delay;
```

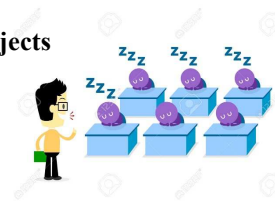
package STANDARD

In every VHDL tool, this package defines, among others, data types, such as: bit, boolean, bit_vector, character, string, text etc.



Lexical elements

- **literals**
inscriptions which represent the data, the way they are written implies all their properties, including their value
- **identifiers (names)**
strings of letters and digits, identifying the objects
- **objects**
signals, variables, constants, parameters
- **expressions**
formulas including operators and arguments, determining the way of calculation or specification of values





Lexical elements Literals

Single literals (scalars)

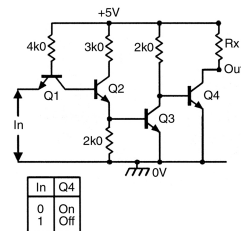
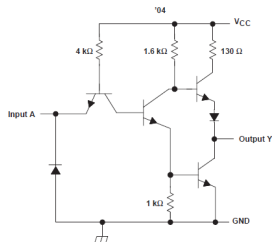
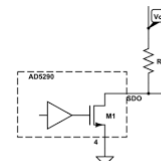
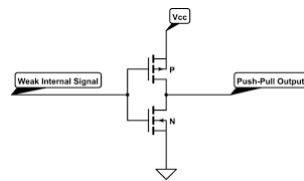
character - single character between apostrophes, eg: **'A'** or **'a'**
bit - represents the binary value **'1'** or **'0'**
std_logic - represents the value of the signal by IEEE 1164:

- 'U'** uninitialized
- 'X'** forcing an unknown
- '0'** forcing 0
- '1'** forcing 1
- 'Z'** Hi-Z
- 'W'** weak unknown
- 'L'** weak 0
- 'H'** weak 1
- '-'** don't care



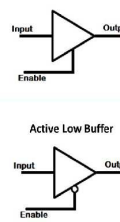
std_logic

- 'U'** uninitialized
- 'X'** forcing an unknown
- '0'** forcing 0
- '1'** forcing 1
- 'Z'** Hi-Z
- 'W'** weak unknown
- 'L'** weak 0
- 'H'** weak 1
- '-'** don't care



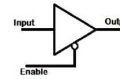
In	Q4
0	On
1	Off

Active High Buffer



When Enable is High- Buffer is active. Therefore logic level of output is the same as that of input
 When Enable is Low- Buffer goes into high impedance state (Disabled)

Active Low Buffer



When Enable is Low- Buffer is active. Therefore logic level of output is the same as that of input
 When Enable is High- Buffer goes into high impedance state (Disabled)



Lexical elements Literals

AGH

boolean - represents two discrete values:

`true TRUE True`
`false FALSE False`

real - represents a floating-point value, eg: `1.3` or

`-344.0E+23`, typically from `-1.0E+38` to `1.0E+38`

with a precision of at least six digits after the decimal point

integer - represents an integer value, eg: `+1,862` or `-257`,

`+123_456`, `16#00FF#`, typically from `-2,147,483,647` to
`+ 2,147,483,647`

time - represent the only physical quantity defined, i.e. time:

`62 fs`, (`ps`, `ns`, `us`, `ms`, `sec`, `min`, `hr`)

Multiple literals (arrays, vectors)

string - string of characters, covered by quotes, eg: `"x"`, `"T hold"`

bit_vector - `"0001_1100"`, `x"00FF"`

std_logic_vector - `"101Z"`, `"UUUUUU"`



Lexical elements Literals

AGH

Decimal literals:

`14`
`7755`
`156E7`
`188.993`
`88_670_551.453_909`
`44.99E-22`

Based literals:

`16#FE#` -- `254`
`2#1111_1110#` -- `254`
`8#376#` -- `254`
`16#D#E1` -- `208`
`16#F.01#E+2` -- `3841.00`
`2#10.1111_0001#E9` -- `1506.00`

Physical literals:

`60 sec`
`100 m`
`5 kohm`
`177 A`

`b"11111110"` - binary representation
`B"1111_1110"` - equivalent binary representation
`x"FE"` - equivalent hexadecimal representation
`O"376"` - equivalent octal representation



Lexical elements Identifiers, ranges

AGH

Basic identifiers

Must begin with a letter. Subsequently there may occur letters, numbers or underscore (`_`). Underscore may not be the last character, nor there may be two neighbouring underscores. VHDL is not case sensitive: `XYZ` \Leftrightarrow `xyz`. Identifiers may not be the same as keywords (approximately 100).

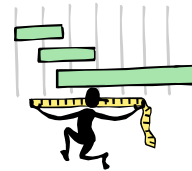
Example: `XYZ`, `X3`, `S(3)`, `S(1 to 4)`, `my_defs`.

The range of variability of the type can be limited:

`range` {*low_val* `to` *high_val* | *high_val* `downto` *low_val*}

np: `integer range 1 to 10;`

`real range 1.0 to 10.0;`



Lexical elements Declarations, signal declarations

AGH

Most of the objects must be declared explicitly. Some objects (eg, iteration identifiers in a loop, signals arising from other signals by the use of attributes) are declared implicitly.

Declaration of objects (their names and types) include declarations of: constants, variables, signals, or files.

Signal declarations

- **scalar**: `signal name(s) : type[range][:= expression] ;`
- **array**: `signal name(s) : array_type [(index)][:= expression] ;`
- **as a port (eg, scalar)**:
`port (name(s) : direction type [range][:= expression] ; ...) ;`



Lexical elements Variable declarations

Variable declarations (within the process)

- scalar:

```
variable name(s): type[(range)] := expression;
```

- array:

```
variable name(s): array_type [(range)] := expression;
```

eg:

```
variable Index: integer range 1 to 50;  
variable Adder_Delay: time range 10ns to 50ns := 10ns;  
variable MEMORY: bit_vector (0 to 7);  
variable x,y: integer;
```

VHDL'93 introduced a shared variable
for communication between processes.



Lexical elements Expressions

Arguments of expressions must respond in terms of the types

Types conversions:

integer (3.0)	=>	integer
real (3)	=>	real
integer * time	=>	time
nanos + picos	=>	time
nanos / picos	=>	integer

```
variable My_Data, My_Sample: integer;
```

```
...
```

```
My_Data := integer(74.94 * real(My_Sample));
```

```
Vector <= CONV_STD_LOGIC_VECTOR(Integer_Variable);
```



Lexical elements Operators

AGH

Operators in expressions:

logical	<code>and</code>	<code>or</code>	<code>nand</code>	<code>nor</code>	<code>xor</code>	<code>not</code>	
relational	<code>=</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>	
concatenation &							
arithmetic	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>**</code>		
	<code>mod</code>	<code>rem</code>	<code>abs</code>				
shift ('93)	<code>sll</code>	<code>srl</code>	<code>sla</code>	<code>sra</code>	<code>rol</code>	<code>ror</code>	<code>xnor</code>

Required types of arguments:

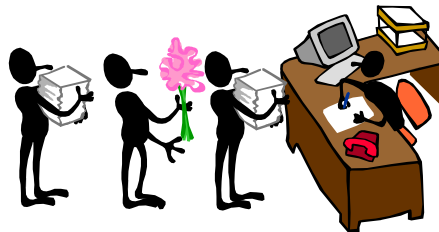
the same	:	<code>and or nand nor xor not</code>
	:	<code>= /= < <= > >= + - * /</code>
integer	:	<code>mod rem</code>
integer exp	:	<code>**</code>
numerical	:	<code>abs</code>



Sequential statements

AGH

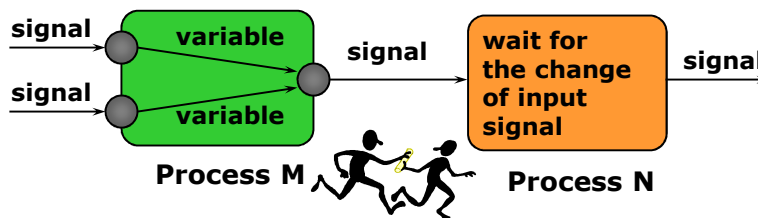
- `process` (*concurrent!*)
- sequential assignment
- `wait`
- `if`
- `case`
- `null`
- `loop`
- `next`
- `exit`
- `assert`
- subprograms





Sequential statements process statement

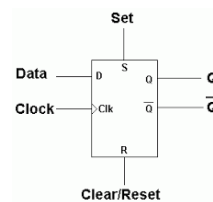
- is a concurrent statement!
- defines a part of an architecture, where instructions are interpreted sequentially
- contains only sequential statements
- must contain either a list of signals that activate (ie, sensitivity list) or **wait** instruction
- provides the ability to 'programming-like' definition of behaviour
- has the ability to change the signals defined in **architecture** and / or **entity**



Sequential statements process statement assignment statement

Syntax:

```
[label : ]
process [ (sensitivity list) ]
[subprograms]
[types]
[constants]
[variables]
[other declarations]
begin
    sequential statements
end process [label];
```



Variable assignment statement

```
variable := expression;
```

Signal assignment statement

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



Sequential statements wait statement

Syntax:

```
wait [on sensitivity list]
     [until condition]
     [for time-expression]
```

Examples:

```
wait on a,b; -- process activation
wait until x > 10;
wait for 10 ns;
wait; -- waits forever
```



The following scripts are equivalent :

```
process                                process (a,b);
...
wait on a,b; <=> ...
end process;                            end process;
```



Sequential statements if statement

Syntax:

```
if condition then sequential_statements;
  [elsif condition then sequential_statements] ;
  [else sequential_statements];
end if;
```

Example:

```
process (R, CLK)
begin
  if R = '0' then
    operand(7 downto 0) <= "00000000";
  elsif CLK = '1' and CLK'event then
    operand(7 downto 0) <= DATAB;
  end if;
end process;
```



Sequential statements case statement

AGH

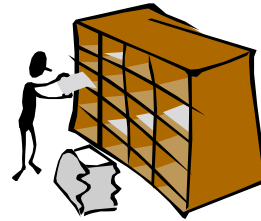
Especially convenient to decode: the codes, the states of finite state machine or the buses states.

Syntax:

```
case expression is
  when val           => sequential_statements ;
  [when val1 | val2  => sequential_statements ;]
  [when val3 to val4 => sequential_statements ;]
  [when others       => sequential_statements ;]
end case;
```

Example:

```
case BCD_int is
  when 0 => LED <= "1111110";
  when 1 => LED <= "0110000";
  ...
end case;
```



Concurrent statements

AGH

- signal assignment statements
 - unconditional
 - conditonal
 - selected
- subprograms
- block





Concurrent statements Unconditional signal assignment

In both examples below, the result of the assignment is the same:

```
architecture sequential of MULTIPLEXER is
begin
    process (A, INDEX)
    begin
        OUTPUT <= A (INDEX); -- sequential
    end process;
end sequential;
```

```
architecture concurrent of MULTIPLEXER is
begin
    OUTPUT <= A (INDEX); -- concurrent
end concurrent;
```



Concurrent statements Conditional signal assignment

Syntax:

signal <= {*expression when condition else*} *expression* ;

Example:

```
DATA <= ROM when ADR < x"2000" else
        RAM when ADR < x"6000" else
        "ZZZZZZZZ";
```

Analogous to the sequential **if** statement, but:

- execute without taking the order into account,
- used in *dataflow* i *structural* description styles,
- are synthesizes to the combinational logic.

Note! Can not be used inside the **process**.



Concurrent statements Selected signal assignment

Syntax:

```
with expression select  
signal <= {expression when choices,};
```

Example:

```
with digit select  
out <= '1' when 0 | 9,  
       '0' when 1 to 8,  
       'Z' when others;
```



Analogous to the sequential **case** statement.

Note! Can not be used inside the **process**.



Advanced data types

- Predefined types
- Extended Types
 - Enumerated Types
 - Subtypes
- Composite Types
 - Arrays
 - one-dimensional (vectors)
 - multidimensional
 - Records
- Other Predefined Types
 - Files
 - Lines



Advanced data types Arrays

AGH

- Consist of elements of the same type.
- Used to describe the buses, registers and other sets of hardware components.
- Array elements can be scalars or other composite objects.
(It is not possible to create eg arrays of files!)
- Access to the specific elements through the use of pointers.



The only predefined array types are:

- `bit_vector` (`package STANDARD`)
- `string` (`package STANDARD`)
- `std_logic_vector` (`package STD_LOGIC_1164`)

If needed, the user has to declare the new types of arrays for the real and integer elements himself.



Advanced data types – Arrays Vectors – index range declaration

AGH

The way of access depends on the way of declaration.

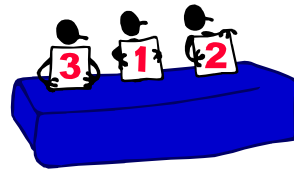
Examples:

```
variable c: bit_vector (0 to 3);
variable d: bit_vector (3 downto 0);
c := "1010";
d := c;
```

`c(0)` `c(1)` `c(2)` `c(3)`

1	0	1	0
---	---	---	---

`d(3)` `d(2)` `d(1)` `d(0)`



- `b(4 to 7)` any range
- `c(4)` index out of range
- `c(1.0)` wrong type of index



Advanced data types – Arrays Vectors – assignments

AGH

Example: for the `bit_vector` type:

```
c := "1010";           constant of the bit_vector type
c := x"A";             as above, note the length!
c := S & T & M & W;     4 concatenated 1-bit signals
c := ('1', '0', '1', '0'); 4-bit aggregate
c := 10;               forbidden
```

Vector slices

Assignments may be executed between the slices (fragments) of vectors.

Examples:

```
variable a: bit_vector (3 downto 0);
variable c: bit_vector (8 downto 1);
c(6 downto 3) := a;
```

`c(6 downto 3)`, not the `c(3 to 6)` - direction of the indexes must be the same as in the declaration!



Advanced data types – Arrays Vectors – aggregates

AGH

Literal may contain a list of array elements in positional and / or named association, creating the so-called aggregate.

Syntax:

```
[type_name' ] ([choice =>] expression1 {, [others =>] expression2})
```

Examples:

```
variable a,b: bit := '1';
variable x,y: bit_vector (1 to 4);
-- positional association
x := bit_vector' ('1', a NAND b, '1', a OR b);
-- named association
y := (1 => '1', 4 => a OR b, 2 => a NAND b, 3 => '1');
```

=> read: "receives"

Using the named association, [*choice =>*] points to the one or many elements.

[*choice =>*] may include an expression (eg: (*i mod 2*) =>), pointing to one element or contain the range (eg: *3 to 5 =>* or *7 downto 0 =>*), pointing to the sequence of elements. The positional association has to be used before the named one.

Example:

```
variable b: bit;  
variable c: bit_vector (8 downto 1);  
c := bit_vector('1', b, 5 downto 2 => '1', others => '0');
```

Very convenient and frequently used:

```
Counter <= (others => '0');  
Data_Bus <= (others => 'Z');
```

To be continued...

