VHDL in more detail

Bert Molenkamp
Department of Electrical Engineering, Mathematics and Computer Science
University of Twente
PO Box 217
7500 AE Enschede
the Netherlands
email: e.molenkamp@utwente.nl

Contents

Data types; enumeration

- Scalar types
  - Enumeration type

    predefined

    TYPE character IS ...(ASCII VHDL'87, ISO-8859-1 since VHDL'93)

    TYPE bit IS ('0', '1');

    TYPE boolean IS (false,true);

    TYPE severity_level IS (note,warning,error,failure)

    file_open_kind and *file_open_status* (since VHDL'93)

    user-defined

    in library IEEE

    TYPE std_ulogic IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

    in your own library perhaps:

    TYPE traffic_light_color IS (red,yellow,green);

Order is important; e.g. left most element is the default initial value. !

Data types; integer

- The standard does not include, explicitly nor implicitly, a bit representation!
- The minimal range implies that 1-complement representation with 32 bits is possible.
- Most tools use a 2-complement representation.

- Integer types

  predefined

  TYPE integer IS …. minimal range -2147483647 to +2147483647 (bounds included)

  user-defined

  TYPE byte IS RANGE 0 TO 255;

Strongly typed language

```
TYPE Int0To255 IS RANGE 0 TO 255;
TYPE byte IS RANGE 0 TO 255;

VARIABLE a : Int0To255;
VARIABLE b : byte;
```

illegal

```
a := b;
```

valid

```
a := Int0To255 (b);
b := byte(a);
```

Type conversion functions <u>implicitly</u> declared for closely related types.

Strongly typed language /2

```
TYPE Int0To255 IS RANGE 0 TO 255;
SUBTYPE funny IS Int0To255 RANGE 0 TO 10;

VARIABLE a : Int0To255;
VARIABLE b : funny;
```

valid

```
b := a;
```

Do you need a sub range of an integer?

```
VARIABLE i : INTEGER RANGE 0 TO 9;
```

Data types; physical

- scalar types, cont'd
  - **Physical types**

    predefined

    time, implementation defined,

```
TYPE time IS RANGE implementation defined UNITS
    fs;                          -- primary unit
    ps     = 1000 fs;            -- secondary units
    ns     = 1000 ps;            --  "
    us     = 1000 ns;            --  "
    ms     = 1000 us;
    sec    = 1000 ms;
    min    = 60 sec;
    hr     = 60 min;
END UNITS;
```

- minimum range guaranteed $-2^{31}-1$ $+2^{31}-1$
  only -2.147 us , 2.147 us (with 32 bits)
- base unit selectable
- most tools support 64 bits, -2.5 hours + 2.5 hours

user-defined

Data types; floating point

- Floating point types

  predefined

  real

  Since VHDL Std 1076-2002 it is required to support the IEEE Std 754 or IEEE Std 854 floating point standard with a minimum of 64 bits.

  user-defined

  TYPE probability IS RANGE 0.0 TO 1.0;

Data types; array

- Composite types
  - **Array types**

    constrained and unconstrained

```
TYPE word IS ARRAY (15 DOWNTO 0) OF bit;
TYPE memory IS ARRAY (natural RANGE <>) OF word;
VARIABLE w1 : word;
VARIABLE mem : memory (0 TO 255);
```

```
mem(1) := "1111111111111111";
mem(2)(5 downto 1) := "00000";
```

predefined

```
TYPE bit_vector IS ARRAY (natural RANGE <>) OF bit;
TYPE string IS ARRAY (positive RANGE <>) OF character;
```

```
SUBTYPE natural IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS integer RANGE 1 TO integer'HIGH;
```


Data types; array /2

- Array types cont'd

  user defined

```
TYPE multi IS ARRAY (0 TO 3, 0 TO 4, .. ) OF bit;
  VARIABLE m : multi;
  m(1,2,...) := '1';
VARIABLE byte : bit_vector (7 DOWNTO 0);
```

index

```
byte (2) := '1';
```

slice

```
byte(4 DOWNTO 2) := "010";
```

concatenate; rotate left

```
byte := byte (6 DOWNTO 0) & byte (7);
```

others

```
byte:= (7=>'1', OTHERS =>'0');
```

Data types; record

- Record type

```
TYPE date IS RECORD
   day          : integer RANGE 1 TO 31;
   month        : name_of_month;
   year         : integer;
END RECORD;
VARIABLE date1, date2, date3 : date;
```

```
date1.day := 2;
```

```
date3 := date2;
```

positional association

```
date2 := (2, March, 2001);
```

named association

```
date2 := (month => March, year => 2001, day => 2);
```


Data types; access

- Access type (pointer)

```
SUBTYPE byte IS bit_vector(7 DOWNTO 0);
TYPE field;                    -- incomplete type decl.
TYPE memory IS ACCESS field;
TYPE field IS RECORD
   address      : natural;
   content      : byte;
   nxt          : memory;
END RECORD;
```

```
VARIABLE a : memory := NULL;
```

```
VARIABLE b : memory := NEW field;
```

```
VARIABLE c : memory := NEW field'(120,(OTHERS=>'0'),NULL);
```

An object declared as access type must be an object of class variable !

Data types; file

- File types
  - VHDL'87:
    - File is not an object.
    - Opening and Closing of a file depends on the location of the file declaration.
  - Since VHDL'93:
    - Files are really supported.
    - File is an object.
      - Can be used as parameter in functions/procedures
    - During simulation files can be opened and closed.

Files types are available in a way like in many other programming languages !

Operators; enumeration

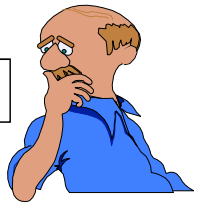
enumeration type

relational: =, /=, <, <=, >, >=
result of type boolean.

boolean and bit

logical: and, or, nand, nor, xor, xnor, not
And, or, nand, nor are short-circuit

And how to deal with logical operators for our own 'logical' types, like std_ulogic?



Overloading

FUNCTION "and" (l,r : std_ulogic) RETURN std_ulogic;

- Write a function (or a procedure)
- If it is an operator add quotes, like "and"

overloading

The selection of a function / procedure is determined by:

- the name, and
- the types

Infix operators may also be used as prefix operators adding quotes

y := a AND b; -- infix

y := "AND"(a,b); -- prefix

!

Bit string literal

Three base specifiers are supported

- B Binary, B"01_10";
- O Octal, O"03_7"; 3 bits for each digit
- X Hex, X"7_F"; 4 bits for each digit

underscores for readability only

In VHDL'87 the bit string literal must be a bit_vector

In VHDL'93 the bit string literal should be a type with at least the elements '0' and '1', e.g. also std_logic_vector

Operators; integer

integer types

relational: =, /=, <, <=, >, >= result of type boolean.
arithmetic: +, -, *, /, **, mod, rem, abs
- division truncates towards 0, like "DIV" in PASCAL

mod versus rem

5 mod 3 = 2	5 rem 3 = 2
(-5) mod 3 = 1	(-5) rem 3 = -2
(-5) mod (-3) = -2	(-5) rem (-3) = -2
5 mod (-3) = -1	5 rem (-3) = 2

Operators; floating point

floating point types

relational: =, /=, <, <=, >, >= result of type boolean.
arithmetic: +, -, *, /, abs

** with right operand of integer type

Operators /4

physical types

relational: =, /=, <, <=, >, >= result of type boolean.

arithmetic: +, -, *, /, abs

+,-,abs: operands of the same type only

* physical type × integer(or real) → physical type

* integer(or real) × physical type → physical type

/ physical type × integer(or real) → physical type

/ physical type × physical type → universal Integer

operator | Left operand | Right operand | Result type

Operators; array

array types

relational: =, /=, <, <=, >, >= result of type boolean.

concatenation: & single elements and one dimensional arrays only
'1' & '0' results in "10"
"11" & '0' results in "110"

array types of bit and boolean

logical: and, or, nand, nor, xor, xnor, not
example "1010" AND "1100" results in "1000"

Shift operators for bit_vector

sll, srl, sla, sra, rol, ror

Operators; record

record types

relational: =, /= result of type boolean.

```
TYPE demo1 IS RECORD a : integer; END RECORD;
TYPE demo2 IS RECORD a,b : integer; END RECORD;
   VARIABLE x1 : demo1;
   VARIABLE x2 : demo2;
```

valid

```
x1:=(a=>5);
x2:=(5,7);
x2:=(a=>5,b=>7);
```

illegal

```
x1:=(5);
```

If an array or a record has one element a *named association* is required for assignment. !

Operators; precedence

operators in increasing precedence

- and, or, nand, nor, xor, xnor
- =, /=, <, <=, >, >=
- sll, srl, sla, sra, rol, ror
- +, -, &
- +, - (as sign operator)
- *, /, mod, rem
- **, abs, not

illegal

```
a**-B
a NAND b NAND c
```

'0' nand '0' nand '1' ???
Not associative

(0 nand 0) nand 1
= 1 nand 1 = 0

0 nand (0 nand 1)
=0 nand 1 = 1

valid

```
a** (-B)
a AND b AND c
```

Subprograms

Two kind of subprograms

- functions
- procedures

Signals may not be declared in subprograms.

functions

- exactly one result
- can drive a signal <u>and</u> a variable
- no wait statements allowed

procedures

- any number of results
- can drive only a signal <u>or</u> only a variable
- wait statements are allowed

Function, an example

```
FUNCTION reduce_or (inp : bit_vector( 3 DOWNTO 0)) RETURN bit IS
  VARIABLE tmp : bit;
BEGIN
  tmp := '0';
  FOR i IN 3 DOWNTO 0 LOOP
    tmp := tmp OR inp(i);
  END LOOP;
  RETURN tmp;
END reduce_or;
```

But what if the length of the input vector is changed?

Function, an example of an unconstrained array

```
FUNCTION reduce_or (inp : bit_vector) RETURN bit IS
  VARIABLE tmp : bit;
BEGIN
  tmp := '0';
  FOR i IN inp'RANGE LOOP
    tmp := tmp OR inp(i);
  END LOOP;
  RETURN tmp;
END reduce_or;
```

Simulates faster

```
FUNCTION reduce_or (inp : bit_vector) RETURN bit IS
BEGIN
  FOR i IN inp'RANGE LOOP
    IF inp(i)='1' THEN RETURN '1'; END IF;
  END LOOP;
  RETURN '0';
END reduce_or;
```

Not allowed is:

FUNCTION <name><input> RETURN bit_vector(1 DOWNTO 0) ….

Use (or a subtype):

FUNCTION <name><input> RETURN bit_vector ….

Procedure, an example

```
PROCEDURE reduce_or (inp : bit_vector; res : OUT bit) IS
  VARIABLE tmp : bit;
BEGIN
  tmp := '0';
  FOR i IN inp'RANGE LOOP
    tmp := tmp OR inp(i);
  END LOOP;
  res := tmp;
END reduce_or;
```

Return statement allowed in a procedure

Package and package body

- Packages can encapsulate subprograms and types to be used elsewhere.
- The interface of a subprogram in the package
- The body of a subprogram in the package body (the interface declaration is repeated)
- Local subprograms and types allowed in package body
  - no interface declaration in the package for these subprograms

Package and package body /2

```
PACKAGE demo IS
 FUNCTION reduce_or (inp : bit_vector) RETURN bit;
END demo;
```

```
PACKAGE BODY demo IS
 FUNCTION reduce_or (inp : bit_vector) RETURN bit IS
   VARIABLE tmp : bit;
 BEGIN
   tmp := '0';
   FOR i IN inp'RANGE LOOP
     tmp := tmp OR inp(i);
   END LOOP;
   RETURN tmp;
 END reduce_or ;
END demo;
```

Package and package body /3

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY work;
USE work.demo.ALL;
ENTITY usedemo IS
  PORT (a, b : IN std_logic);
....
```

- "Library work" is default library clause. It is not necessary to write it explicitly.
- Implicitly declared above each design unit is:

```
LIBRARY std, work;
USE std.standard.ALL;
-- package 'standard' contains predefined environment
```


Analysis order

- ***Primaries*** **(entities, packages) must be analyzed before their *secondaries* (resp. architecture and package body).**
- **If a *primary* is analyzed <u>all places</u> where that primary is used should be analyzed again too.**
- **If a *secondary* is changed then only that is to be analyzed.**

Place primaries and secondaries in different files.
Use logical names for the files, e.g.

mux.vhd	**entity**
mux-behaviour.vhd	**architecture behaviour**

!

Sequential statements

- In processes and subprograms
- Order dependent

```
if statement
case statement
loop statement
next statement
exit statement
procedure call
statement
return statements
null statement
wait statement
assert statement
```


If statement

IF cond THEN ... END IF;

IF cond THEN ... ELSE ... END IF;

IF cond THEN ... ELSIF cond1 THEN ... END IF;

```
IF i='1'
  THEN y := '0';
  ELSE y := y;
END IF;
```

```
IF i='1'
  THEN y := '0';
END IF;
```

```
IF i='1'
  THEN y := '0';
  ELSE NULL;
END IF;
```

case statement

```
CASE selection IS
  WHEN choice1 => ...
  WHEN choice2 => ...
  WHEN OTHERS => ...
END CASE;
```

- selection must be
  - discrete type, or
  - one dimensional array
- no overlap in the choices
- each value in the selection should occur in the choices
  - WHEN OTHERS is optional (last choice!)
- locally static expression required for selection

```
VARIABLE sel : integer;
...
CASE sel IS
  WHEN 0 TO 5 => ...          -- values 0 up to 5
  WHEN 7 | 10 | 2**10 => ...  --  'or'
  WHEN OTHERS => ...
END CASE;
```


case statement /2

Illegal; legal since VHDL'2008

```
-- a and b of type bit
CASE a&b IS
  WHEN "00" => y <= i0;
  WHEN "01" => y <= i1;
  WHEN "10" => y <= i2;
  WHEN "11" => y <= i3;
END CASE;
```

Illegal since the type of **a&b** is unknown. Legal since 2008 because we are interested in the pattern only.

valid solution

```
VARIABLE sel : bit_vector(1 DOWNTO 0);
.....
sel := a&b;
CASE sel IS
  WHEN "00" => ...
```

Loop, next, and exit statements

- Loop variable is implicitly declared in a *for scheme*.
  - Tip: don't declare an object with the same name
- 'For' and 'While' scheme are optional.

```
y := 0;
FOR i IN 0 TO 5 LOOP
  y:=y+i;
END LOOP;
```

```
y := 0;
FOR i IN 0 TO 10 LOOP
 <statement(s)>
 NEXT WHEN i>=6;
 y:=y+i;
END LOOP;
```

```
VARIABLE i : integer;
y := 0; i:= 0;
WHILE i < 5 LOOP
  i := i + 1;
  y:=y+i;
END LOOP;
```

```
y := 0;
FOR i IN 0 TO 10 LOOP
  y:=y+i;
  EXIT WHEN i=5;
  <statement(s)>
END LOOP;
<statement>
```


next, and exit statements

Optional in the *next* and the *exit* statement is:

- label
- condition

```
a:LOOP
  b:LOOP
    EXIT WHEN cond1;      -- out of loop with label b if cond1 is true
    EXIT a WHEN cond2;    -- out of loop with label a if cond2 is true
    EXIT a;               -- out of loop with label a
  END LOOP b;
END LOOP;
```

Wait statement

- suspends a process or procedure.
- wait on <sensitivity list>
  - WAIT ON a,b;
  - resume if a and/or b is changed
- wait until <condition>
  - WAIT UNTIL clk='1';
  - resume if clock becomes '1', an event is required!
    This is equivalent with:

```
LOOP
  WAIT ON clk;
  EXIT WHEN clk='1';
END LOOP;
```

- wait for <time-out>
  - WAIT FOR 10 ns;
  - resume after 10 ns.

Wait statement /2

General case:

WAIT ON a UNTIL b='1' FOR 10 ns;

Process or *procedure* resumes if

- time-out interval is expired, or
- an event on signal a AND condition b='1' is true.

In the example above an event on signal b is <u>not</u> required !

WAIT UNTIL b='1';	similar as	WAIT ON b UNTIL b='1';
WAIT ON a UNTIL b='1';	not similar as	WAIT ON a,b UNTIL b='1';

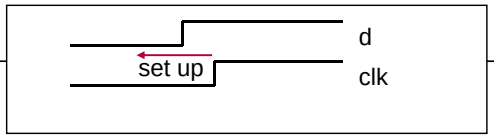
Assert statement

ASSERT cond REPORT “message” SEVERITY <severity_level>

If the condition is not fulfilled then

- **message is shown, and**
- **depending on the severity level simulation stops**

TYPE severity_level IS (note,warning,error,failure);




```
PROCESS
BEGIN
  WAIT UNTIL clk='1';
  ASSERT d'STABLE( 10 ns ) REPORT "setup violation" SEVERITY error;
  q <= d;
END PROCESS;
```


Concurrent statements

- **Hardware is “concurrent”**
- **Order independent**
- **Execute once initially**

concurrent signal assignment statement

concurrent assert statement

component instantiation statement

process statement

block statement

generate statement

concurrent procedure call

Conc. signal assignment statement

- Conditional
- Selected

  has same requirements as the case statement.

conditional

```
y <= a;
z <= a WHEN inp= '1' ELSE
     b WHEN inp= '0' ELSE
     'X';
```

selected

```
WITH inp SELECT
   z <= a WHEN '1',
        b WHEN '0',
        'X' WHEN OTHERS;
```


Concurrent assert statement

ASSERT cond REPORT “message” SEVERITY <severity_level>

If the condition is not fulfilled then

- message is shown, and
- depending on the severity level simulation stops

```
TYPE severity_level IS (note,warning,error,failure);
```

```
ASSERT a>b REPORT "a is not greater then b" SEVERITY error;
```

Component instantiation statement

```
  COMPONENT i8086
    PORT (..)
  END COMPONENT
BEGIN
  inst : i8086 PORT MAP (..);
```

- flexibility for the future; easy replacement of an old component with a new one.
- Top-down design possible; late binding (configuration)

Component instantiation statement /2

VHDL'93 also allows direct instantiation of a design entity.

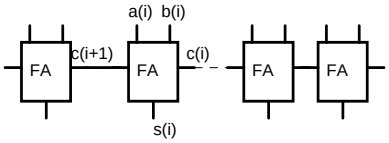
```
BEGIN
  inst : ENTITY work.i8086(behaviour) PORT MAP (..);
```

or

```
BEGIN
  inst : CONFIGURATION work.conf PORT MAP (..);
```

Generate statement

A mechanism for *iterative* elaboration of a description.




```
label_required:FOR i IN a'RANGE GENERATE
  fulladder : fa PORT MAP (a(i), b(i), c(i), c(i+1), s(i));
END GENERATE;
```

Since VHDL'93 a declarative region is optional

```
......GENERATE
  SIGNAL c : bit_vector(...
BEGIN
  fulladder : FA .....
```


Process statement

```
label_optional:PROCESS (sensitivity_list_optional)
  <declarative region>
BEGIN
  <seq. statements>
END PROCESS;
```

- Signal declarations are not allowed in the declarative region.
- A process statement should have at least one wait statement.
- A process statement with a sensitivity list after the keyword PROCESS may have no other wait statement.

Following processes are equivalent

```
PROCESS(a)
BEGIN
  b <= a;
END PROCESS;
```

```
PROCESS
BEGIN
  b <= a;
  WAIT ON a;
END PROCESS;
```

Block statement

```
label_required:BLOCK
  <declarative region>
BEGIN
  <concurrent statements>
END BLOCK;
```

- Variable declarations are not allowed in the declarative region.
- The label is required for configuration specification.
- Block statements can have a guard. Like
  label:BLOCK(a='1') .....
  Not part of this course.

Concurrent procedure call

- A procedure can also be used as a concurrent procedure call.
- The default object class of mode *input* of a procedure is *constant* and that of the mode *output* is *variable*.

```
PROCEDURE wire ( a : IN bit; b : OUT bit) IS
BEGIN
  b := a;
END wire;
```

Object explicitly given

```
PROCEDURE wire (CONSTANT a : IN bit;
                VARIABLE b : OUT bit) IS
BEGIN
  b := a;
END wire;
```

Concurrent procedure call /2

Object explicitly given

```
PROCEDURE wire (          a : IN bit;
                     SIGNAL b : OUT bit) IS
BEGIN
  b <= a; -- note the changing of the assignment operator symbol
END wire;
```

concurrent procedure call

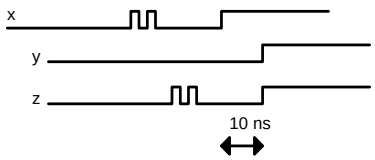
```
ARCHITECTURE ..
BEGIN
  wire(c,d); -- concurrent
```

is equivalent with

```
ARCHITECTURE ..
BEGIN
  PROCESS
  BEGIN
    wire(c,d); -- sequential
    WAIT ON c;
  END PROCESS;
```


Modeling delay

- Inertial delay y <= x AFTER 10 ns;
  - Default delay mechanism
  - Signal y only changes if the change of signal x is stable for the given time period (10 ns).
  - Modeling delay of physical components.
- Transport delay z <= TRANSPORT x AFTER 10 ns;
  - Signal z is equal x with a delay of 10 ns.
  - Modeling wire delay.



Modeling delay /2

- Multiple assignments to a signal is possible.
- Time must be in increasing order.
- Time is relative to the current simulation time.

```
y <=  '0' AFTER 10 ns,
      '1' AFTER 20 ns,
      '0' AFTER 30 ns;
```

Generic descriptions

- Most descriptions have similar behaviour, only the constraints are different
- VHDL supports:
  - unconstrained arrays

    FUNCTION reduce_or (inp : bit_vector) RETURN bit;
  - generics in the entity declaration are *global constants*. "Global" means that during elaboration the constant value is determined. (*Local constants;* value of constant known at analysis time).

A selection expression in the case statement need to be a *locally* static expression.

Generic descriptions /2

- OPEN, then default value is used.
- No semicolon after GENERIC MAP (if a PORT MAP follows).
- If named association is used then the order is not important

```
ENTITY inv IS
  GENERIC (delay : time := 10 ns;
           width : positive := 1);
  PORT (a : IN bit_vector(1 TO width);
        b : OUT bit_vector(1 TO width));
END inv;

ARCHITECTURE behaviour OF inv IS
BEGIN
  b <= NOT a AFTER delay;
END behaviour;
```

examples of instantiations

```
inst1:inv GENERIC MAP(20 ns,4) PORT MAP(x, y);

inst2:inv GENERIC MAP(30 ns,5) PORT MAP(v, u);
```


Attributes

The language has a rich set of attributes. The often used attributes are:

- 'event exa: clk'event
  Is a *function* that results in a boolean. The boolean is true if there is an event on the signal clk.
- 'stable(<time_period_is_optional>) exa: clk'stable(10 ns)
  Is a *signal* of type boolean. The boolean is true if the signal is stable for the given time period.
  If no time period is given the value is the inverse of the 'event attribute (but still a signal)

Attributes /2

- 'left and 'right
  Returns the left/right bound of an array
- 'range
  Returns the range, from *left* to *right*, of the array
- 'reverse_range
  Returns the range, from *right* to *left*, of the array
- 'length
  Returns the length of the array

```
VARIABLE a : bit_vector (4 DOWNTO 2);
VARIABLE b : bit_vector (8 TO 10);

a'LEFT          is   4
a'RANGE         is   4 DOWNTO 2
b'REVERSE_RANGE is   10 DOWNTO 8
b'LENGTH        is   3
```

```
VARIABLE c : bit_vector (5 TO 2);
is legal; it is a null array!
```

Multiple driven signal

- Each process creates exactly <u>one</u> driver for every signal assigned to in that process.
- What should be the final value of the signal?
  - The user has to define a function with the desired behaviour.
  - A subtype is needed with this function; a resolution function
- Often these subtypes are already available in a package.

```
process1:x <= '0' WHEN a='1' ELSE
                '1';
```

```
process2:PROCESS (a)
BEGIN
  x <= '1';
  x <= TRANSPORT '0' AFTER 10 ns;
END PROCESS;
```

Std_logic_1164

Types in Std_logic_1164

- std_ulogic (std_ulogic_vector)
  - no resolution function, hence no multiple driven signal allowed.
- std_logic (std_logic_vector)
  - a resolution function is attached to the type
  - procedure as follows (see figure):

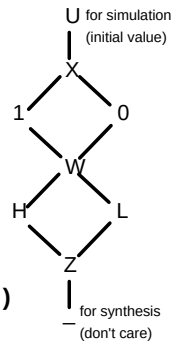
    values at same level, then

    not equal values → value one level higher

    equal values → value is the same

    ('-' → value 'X' in combination with 'U' the result is 'U')

    values not at same level, then the highest value is the resolved value.



('0', '1') → 'X'
('H', '0') → '0'
('0', 'Z', '1') → (('0', 'Z'), '1') → ('0', '1') → 'X' The order is not defined!

Std_logic_1164 /2

- Std_logic is a subtype of std_ulogic
- Std_logic_vector is a <u>not</u> subtype of std_ulogic_vector.

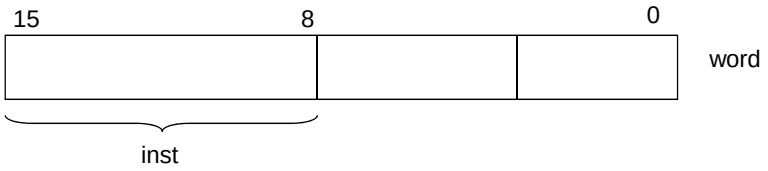
```
VARIABLE ulog : std_ulogic;
VARIABLE log : std_logic;
VARIABLE ulogv : std_ulogic_vector(2 DOWNTO 0);
VARIABLE logv : std_logic_vector(2 DOWNTO 0);
.....
log := ulog; -- valid, same base type!
ulogv := logv; -- illegal, different types
```

Conversion functions available in package std_logic_1164, e.g.

```
FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector;
FUNCTION To_StdULogicVector ( s : std_logic_vector  ) RETURN std_ulogic_vector;
```

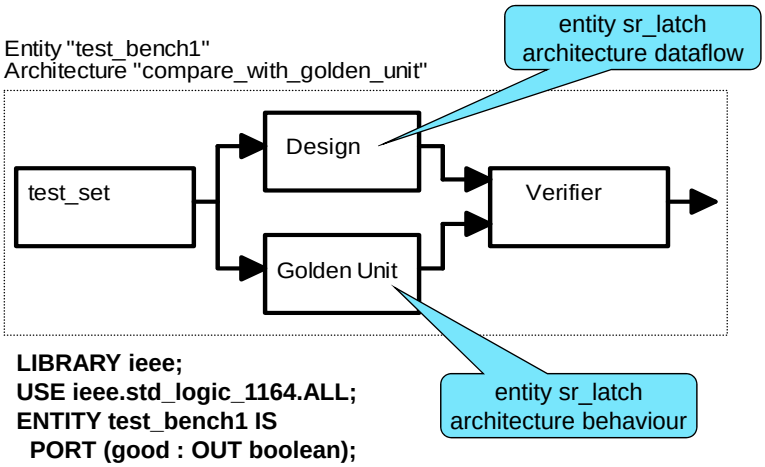
Alias

- In a processor often parts of a *word* have different meanings, e.g.:
  - the instruction
  - operands
- An alias can be used to select these parts using the logical names.
- An alias is <u>not</u> an object!




```
VARIABLE word : bit_vector (15 DOWNTO 0);
ALIAS inst : bit_vector (7 DOWNTO 0) IS word(15 DOWNTO 8);
ALIAS op1 : bit_vector (3 DOWNTO 0) IS word(7 DOWNTO 4);
ALIAS op2 : bit_vector (3 DOWNTO 0) IS word(3 DOWNTO 0);
...
op1 := "0010";
```


Test Bench /5




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY test_bench1 IS
  PORT (good : OUT boolean);
END test_bench1;
```

Test Bench /6

```
ARCHITECTURE compare_with_golden_unit OF test_bench1 IS
  COMPONENT sr_latch
    PORT (s, r    : IN  std_ulogic;
          q, qn : OUT std_ulogic);
  END COMPONENT;
  COMPONENT test_set
    PORT (set, reset : OUT std_ulogic := '0');
  END COMPONENT;
  COMPONENT verifier
    PORT (q_des,q_gold,qn_des,qn_gold : IN std_ulogic;
          good : OUT boolean);
  END COMPONENT;

  SIGNAL s,r,q_des,qn_des,q_gold,qn_gold : std_ulogic;
```

cont'd on next slide

Test bench /7

Configuration specification

```
  FOR design:sr_latch USE ENTITY work.sr_latch(dataflow);

  FOR golden_unit:sr_latch USE ENTITY work.sr_latch(behaviour);

BEGIN

  design:sr_latch PORT MAP(s,r,q_des,qn_des);

  golden_unit:sr_latch PORT MAP (s,r,q_gold,qn_gold);

  testb:test_set PORT MAP(s,r);

  ver:verifier PORT MAP (q_des,q_gold,qn_des,qn_gold,good);

END compare_with_golden_unit ;
```

Configuration

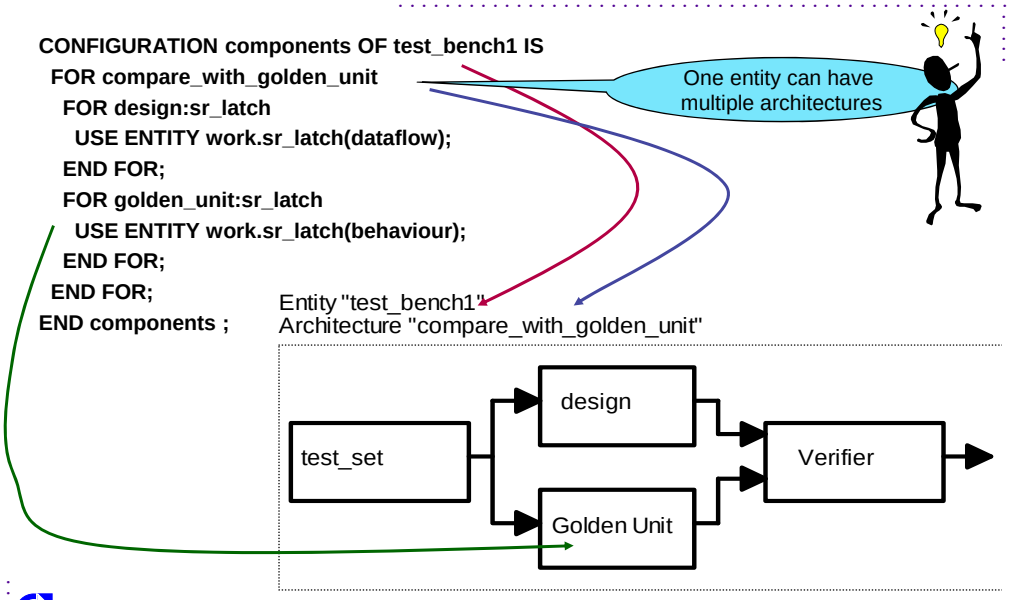
- Component specification in an architecture

```
ARCHITECTURE ..
    FOR inst:inverter USE ENTITY work.inv(behaviour);
```

- Default *binding*
  - If no configuration is given.
  - The last compiled architecture corresponding to the entity. Note: the entity declaration must exactly be the same (typographic!) as the component declaration.
  - The entity declaration must be exactly the same (the same names, and the same order, ..) as the component declaration.
  - Some tools require for default binding an <u>empty</u> configuration declaration.

Test Bench /8

```
CONFIGURATION components OF test_bench1 IS
  FOR compare_with_golden_unit
    FOR design:sr_latch
      USE ENTITY work.sr_latch(dataflow);
    END FOR;
    FOR golden_unit:sr_latch
      USE ENTITY work.sr_latch(behaviour);
    END FOR;
  END FOR;
END components ;
```



Test Bench /9

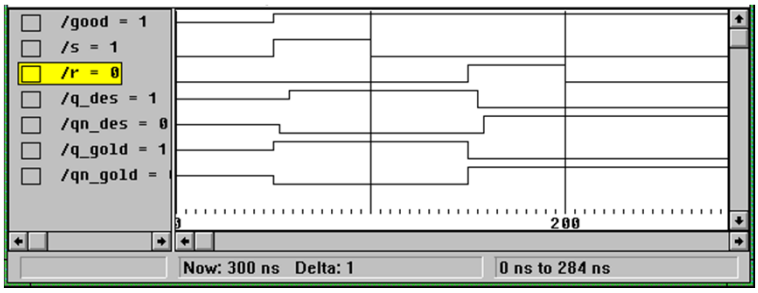
Problem of the verifier:

When is the output stable?

```
ARCHITECTURE behaviour OF verifier IS
BEGIN
  PROCESS
  BEGIN
    WAIT FOR 50 ns;
    good <=(q_gold=q_des) AND (qn_gold=qn_des);
  END PROCESS;
END behaviour;
```

In a synchronous system the wait statement can be replaced with
Wait until clk='1';

Test Bench /10



Port map / pitfalls

The actual in the association list of a port map must be signal.

```
Entity funny is
        port (z : bit_vector(0 to 1) …
signal a1, a2 : bit;
Signal v : bit_vector (0 to 1);
```

Not allowed is:

```
        lbl : funny port map (a1 & a2, ..)
```

Since a1 & a2 is an expression, not a signal

Allowed is:

```
        v<= a1 & a2;      lbl : funny port map (v,..)
        or
        lbl : funny port map (z(0)=>a1, z(1)=>a2,..)
```


Port map / pitfalls

The actual in the association list of a port map must be signal.

```
Entity funny is
        port (z : bit …
```

Allowed is (since 1993)

```
        lbl : funny port map ('1', ..)
```

```
Entity funny is
        port (z : out bit …
```

Output not connected:

```
        lbl : funny port map (OPEN, ..)
```

Qualification

If the language can not determine the type a qualification is necessary.

```
PROCEDURE Do_something (c : character);
PROCEDURE Do_something (c : bit);
```

Illegal: is '0' a bit or a character?

```
Do_something('0');
```

qualification

```
Do_something( bit' ( '0' ) );
```


Qualification /2

illegal

```
-- a and b of type bit
CASE a&b IS
  WHEN "00" => ...
```

valid solution

```
CASE bit_vector'( a&b ) IS
  WHEN "00" => ...
```