VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

- VHDL is
  - Non-proprietary language (IEEE-1076 1987/1993/2002/2008)
  - Widely supported Hardware Description Language (HDL)
  - Programming language
    - similar to Ada – typing definitions, type checking, overloading
  - Simulation language
  - Documentation language
  - Usable for register-transfer level and logic synthesis
    - algorithmic level supported in part

VHDL properties

- Openness and availability
- Support of different design methodologies and technologies
- Independence from technologies and production processes
- Large variety of description possibilities
- Worldwide design process, and project interchangeability and reuse

VHDL history

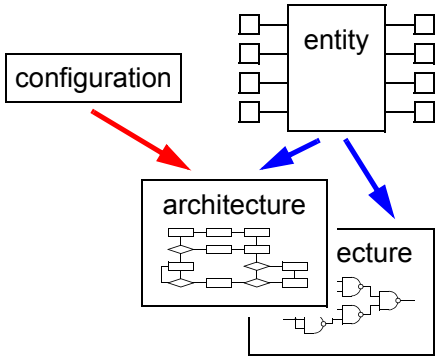
- June 1981 - brainstorm in Massachusetts (state, DoD, academy)
- 1983 - USA government contest
- 1985 - ver. 7.2 (IBM, TI, Intermetics), first software tools
- 1986 - IEEE started standardization. Standard IEEE--1076 (VHDL'87)
- 1987 - fully functional software from Intermetics
- 1994 - version VHDL'93 (IEEE 1076-1993), new features added
- 1999 - VHDL-AMS (IEEE 1076.1-1999), Analog and Mixed Signal
- 2002 - VHDL'2000 (IEEE 1076-2002), bug fixes and clarifications,
- 2009 - VHDL 4.0 (IEEE 1076-2008), enhanced synthesizability, development continues...

Some VHDL basics

- VHDL allows one to model systems where more than one thing is going on at a moment
  - concurrency!
  - discrete event simulation
- VHDL allows modelling to occur at more than one level of abstraction
  - behavioral
  - RTL
  - boolean equations
  - gates
- Why do we care about that?
  - Formal Specification
  - Testing & Validation using simulation
  - Performance prediction
  - Automatic synthesis
  - Top-Down modelling:
    behavior –> RTL –> boolean –> gates

Hello, world!

- entity – interface description
- architecture – behavior & structure
- configuration – composition / abstraction level
- package – datatypes, etc.




```
entity test is
end test;

architecture hello of test is
begin
    process begin
        assert false
            report "Hello world!"
            severity note;
        wait;
    end process;
end hello;
```

Hello, world!

- **ModelSim**

```
run -all
# ** Note: Hello world!
#    Time: 0 ps  Iteration: 0  Instance: /test
```

- **Xilinx ISim**

```
Time resolution is 1 ps
Simulator is doing circuit initialization process.
at 0 ps: Note: Hello world! (/test/).
Finished circuit initialization process.
```

- **Synopsys**

```
# run
0 NS
Assertion NOTE at 0 NS in design unit TEST(HELLO) from process /TEST/_P0
    "Hello, world!"
(vhdlsim): Simulation complete, time is 0 NS.
#
```


Design file structure

- **The highest-level VHDL construct is the design file**
  - **a design may consist of multiple design files**
- **A design file contains design units that contain one or more library units**
- **Library units in turn contain: entity, configuration, and package declarations (primary units); and architecture and package bodies (secondary units).**

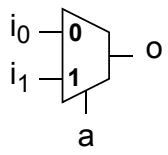
```
design_file    ::= {library_clause | use_clause} library_unit
                   {{library_clause | use_clause} library_unit}
library_unit   ::= primary_unit | secondary_unit
primary_unit   ::= entity_declaration |
                   configuration_declaration |
                   package_declaration
secondary_unit ::= architecture_body | package_body
```

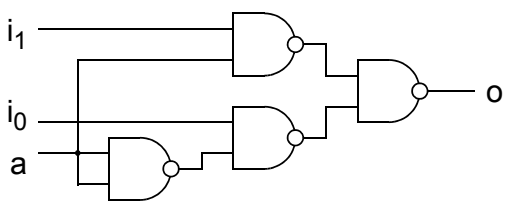
VHDL design units & basic elements

- Entity – interface specification (of a “black box”)
- Architecture – functionality and/or composition of an entity
  - content of the “black box”
  - signals – “wires” between concurrent statements (modules/components/...)
    - value changes at the end of simulation cycles – all simulators give always the same result
    - VHDL’93 allows shared variables in architectures that may introduce non-deterministic behavior
  - concurrent statements
    - processes, components, signal assignments, blocks ... – executed in “parallel”
  - processes – sequential behavior of a “module”
    - variables – value changes immediately
    - sequential statements
      signal/variable assignments, procedure calls, conditional statements, timing control, ...
- Package declaration – declarations, sub-programs
- Package body – bodies of sub-programs
- Configuration – binding entities/architectures, components/entities

Multiplexer – an example

- $o = \overline{a}\, i_0 + a\, i_1$
- De Morgan’s law
  - $\overline{a}\ \&\ \overline{b} = \overline{a + b}$ $\overline{a} + \overline{b} = \overline{a\ \&\ b}$
  - $a' \ \&\ b' = (a + b)'$ $a' + b' = (a\ \&\ b)'$
- $o = ((a' \ \&\ i_0)' \ \&\ (a \ \&\ i_1)')'$

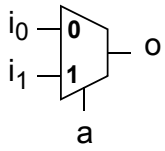




Multiplexer – an example

entity

```
entity MUX is
  port ( a, i0, i1 : in bit;
         o : out bit );
end MUX;
```



behavioral

```
architecture behave of MUX is
begin
  process ( a, i0, i1 ) begin
    if   a = '1'   then
      o <= i1;
    else
      o <= i0;
    end if;
  end process;
end behave;
```

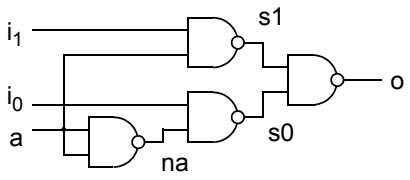
- $o = \bar{a}\, i_0 + a\, i_1$

Multiplexer – an example

dataflow

```
architecture dataflow of MUX is
begin
  o <= ( (not a) and i0 ) or
       ( a and i1 );
end dataflow;
```

- $o = \bar{a}\, i_0 + a\, i_1$



structural

```
architecture struct of MUX is
  component NANDg
    port ( i0, i1 : in bit;
           c : out bit );
  end component;
  signal na, s1, s0 : bit;
begin
  U1: NANDg port map (a,a,na);
  U2: NANDg port map (i1,a,s1);
  U3: NANDg port map (i0,na,s0);
  U4: NANDg port map (s1,s0,o);
end struct;
```

Types, packages, libraries

- ***Types*** **and** ***subtypes*** **present** ***the set of values and a set of operations*****, structure, composition, and storage requirement that an object, such as a variable or a constant or a signal, can hold.**
  - **There** ***exists*** **a set of predefined types in package** ***Standard***
  - **A** ***package*** **is a design unit that allows to group logically related declarations**
  - **A** ***library*** **is a collection of design units (library units)**

Object classes

- **VHDL categorizes objects into four classes:**
  - ***constant*** **- an object whose value may not be changed**
  - ***signal*** **- an object with a past history**
  - ***variable*** **- an object with a single current value**
  - ***file*** **- an object used to represent file in the host environment**
- **The** ***type*** **of an object represents its structure, composition, and storage requirements**
- **The** ***class*** **is relevant to the nature of the object and represents HOW the object is used in the model**

TTÜ1918
Department of Computer Engineering

Basic language elements

- **Lexical elements, identifiers**
- **Syntax – delimiters, literals, statements, operators**
- **Types and subtypes, attributes, aliases**

- **Identifiers**
  - **identifier ::= basic_identifier | extended_identifier**
  - **basic_identifier ::= letter{[underline]letter_or_digit}**
  - **extended_identifier ::= \graphic_character{graphic_character}**
  - **Examples:**
    - **INTGR9, Intgl_5 -- legal**
    - **Intgrl-5, Acquire_, 8to3, Abc@adr -- illegal**
    - **\1 2bs#_3\ -- legal in VHDL'93**

Delimiters

	Name	Example
&	concatenator	FourB_v := TwoB_v & ”10”
|	vertical bar	when ’Y’ | ’y’ =>
#	enclosing based literals	Total_v := 16#2AF#
:	separates data object	variable Sw_v : OnOff_Typ;
.	dot notation	OnOff_v := Message_v.Switch_v;
=>	arrow (read as ”then”)	when On1 => Sun_v := 6;
=>	arrow (read as ”gets”)	Arr_v := (El1 => 5, others => 100);
:=	variable assignment	Sum_v := Numb_s +7;
<=	signal assignment	Count_s <= 5 after 5 ns;
<>	box	type S_Typ is array (integer range <>) ...
--	comment	-- this is definitely a comment;

Literals

- **A literal is a value that is directly specified in the description of a design.**
- **A literal can be a bit value, string literal, enumeration literal, numeric literal, or the literal null.**
- **Examples**
  - **12 0 1E6 -- integer literals**
  - **12.0 1.34E-12 -- real literals**
  - **16#E# 2#1110_0000# -- based literals (also ‘:’ instead of ‘#’ in VHDL’93)**
  - **’A’ ’*’ -- character literals**
  - **”This is a ””string””” -- string literal (also ‘%’ instead of ‘”’ in VHDL’93)**
  - **X”FFF” B”1111” -- bit string literal**

Operators

- ***logical***: and, or, nand, nor, xor (VHDL'87) xnor (VHDL'93)
- ***relational***: =, /=, <, >, <=, >=
- ***shift***: sll, srl, sla, sra, rol, ror (VHDL'93)
- ***adding***: +, -, &
- ***sign***: +, -
- ***multiplying***: *, /, mod, rem
- ***miscellaneous***: **, abs, not

- **Examples**
  - & : concatenation, '1' & "10" = "110"
  - ** : exponentiation, 2**3 = 8
  - mod : modulus, 7 mod (-2) = -1 -- A=B*N+(A mod B)
  - rem : remainder, 7 rem (-2) = 1 -- A=(A/B)*B+(A rem B)

TTÜ1918
Department of Computer Engineering

Expressions & statements

- **Assignments**
  - **signals –** **s <=** [**transport** | **inertial**] *expression* [**after** *time*] **;**
  - **variables –** **v :=** *expression*;
  - **expressions –** *expression operation expression*
variable | *signal*
function-call

- **Control flow statements**
  - **conditional – if-then-else, case**
  - **loops – for-loop, while-loop**
  - **procedure calls**
  - **timing control**

Types

- Scalar type – discrete (integer, enumeration, physical) and real types
- Composite type – array and record types
- Access type – only simulation
- File type – only simulation
  - A VHDL *subtype* is a type with a constraint which specifies the subset of values for the type. Constraint errors can be detected at compile and/or run time.
  - Example:
    - type Bit_position is range 7 downto 0;
    - subtype Int0_5_Typ is integer range 0 to 5;
    - type Color is (Green,Yellow,Red);

Package STANDARD

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  type BIT is ('0','1');
  type CHARACTER is (NUL,SOH,...,'a','b','c',...,DEL);
  type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
  type INTEGER is range -(2**31-1) to (2**31-1);
  type REAL is range ...;
  type TIME is range ...
    units fs; ps=1000 fs; ...  hr=60 min; end units;
  function NOW return TIME;
  subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
  subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
  type STRING is array (POSITIVE range <>) of CHARACTER;
  type BIT_VECTOR is array (NATURAL range <>) of BIT;
end STANDARD;
```

VHDL'93 types

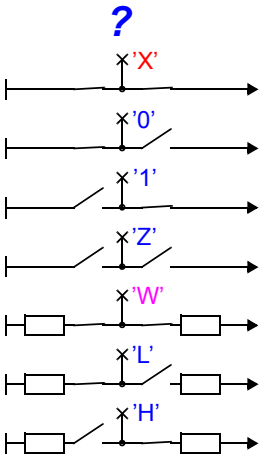
```
attribute FOREIGN: STRING; -- for links to other languages
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
function NOW return DELAY_LENGTH;
type FILE_OPEN_KIND is (READ_MODE,WRITE_MODE,APPEND_MODE);
type FILE_OPEN_STATUS is
  (OPEN_OK,STATUS_ERROR,NAME_ERROR,MODE_ERROR);
```

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;
```


Std_Ulogic Type (IEEE 1164)

State	Definition	Synthesis interpretation
'U'	uninitialized	model behavior
'X'	forcing unknown	model behavior
'0'	forcing 0	logic level ("transistor")
'1'	forcing 1	logic level ("transistor")
'Z'	high impedance	disconnect
'W'	weak unknown	model behavior
'L'	weak 0	logic level ("resistor")
'H'	weak 1	logic level ("resistor")
'-'	don't care	match all



Composite types – array & record

- *Array* – all elements are of the same (sub)type, multiple dimension possible

```
subtype X01_Typ is STD_Logic range 'X' to '1';
type Vect8_Typ is array (7 downto 0) of X01_Typ;
type IntArr is array (integer range <>) of integer;
```

- *Record* – consist of named elements with arbitrary types

```
type Task_typ is record
  Task_Numb    : integer;
  Time_tag     : time;
  Task_mode    : Task_mode_Typ;
end record;
```

Alias – part of a type/variable/signal

```
variable Nmbr: BIT_VECTOR (31 downto 0);
alias Sign : BIT is Nmbr(31);
alias Mantissa : BIT_VECTOR (23 downto 0) is Nmbr (30 downto 7);
alias Exponent : BIT_VECTOR ( 6 downto 0) is Nmbr ( 6 downto 0);
```


Composite type – examples

- Defining 3D array (hyper-cube)

```
type look_up_1 is array (bit) of bit;
type look_up_2 is array (bit) of look_up_1;
type look_up_3 is array (bit) of look_up_2;
constant output_a: look_up_3 :=
  ( ( ( '0', '1' ), ( '1', '0' ) ), ( ( '1', '0' ), ( '0', '1' ) ) );
type look_up_b is array (bit,bit,bit) of bit;
constant output_b: look_up_b :=
  ( ( ( '0', '1' ), ( '1', '0' ) ), ( ( '1', '0' ), ( '0', '1' ) ) );
signal s_a: bit := output_a('1')('0')('1');
signal s_x: look_up_1 := output_a('1')('0');
signal s_b: bit := output_b('1','0','1');
```

- Record

```
type my_rec is record   num: integer;   tm: time;   end record;
constant rec1: my_rec := ( 10, 5 ns );
signal s_n: integer := rec1.num;
signal s_t: time := rec1.tm;
```

Access type

- Access types are used to declare values that access dynamically allocated variables. Such variables are referenced not by name but by an access value that acts like a pointer to the variable.
- The variable being pointed to can be one of the following:
  - *Scalar object*, e.g. enumerated type, integer
  - *Array objects*, e.g. procedures READ and WRITE in package Std.TextIO make use of the access type
  - *Record objects*, e.g. for representing linked lists
- Examples:

```
type AInt_Typ is access integer;
Aint1_v := new integer'(5);     -- new integer with value 5
Deallocate(Aint1_v);            -- Aint1_v now points to null
```


File type (VHDL'87)

- Main difference in VHDL'87 and VHDL'93 - explicit open/close
- VHDL'87 string vector read example:

```
use std.textio.all;
architecture stimulus of testfig is
    Read_input: process
        file vector_file: text is in "testfib.vec";
        variable str_stimulus_in: string(34 downto 1);
        variable file_line: line;
    begin
        while not endfile(vector_file) loop
        readline(vector_file, file_line);
        read(file_line,str_stimulus_in);
        ...
```

File type (VHDL'93)

- File types are used to access files in the host environment.

```
type Stringfile_Typ is file of string;
type IntegerFile_Typ is file of integer;
type BitVector_Typ is file of Bit_Vector;
```

- The following procedures exist for file types (FT) handling:
  - File_Open (file F: FT; External_Name: in string;
    Open_Kind: in File_Open_Kind := Read_Mode);
  - File_Close (file F: FT);
  - Read (F: in FT, Value: out TM); -- TM ::= type|subtype
  - Write (F: out FT; Value: in TM);
  - function EndFile (F: in FT) return boolean;

Type conversion

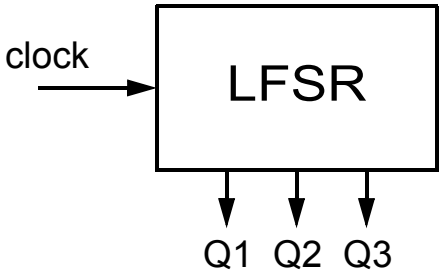
- For type conversion types use:
  - a type conversion function, or
  - a type casting.

```
library IEEE;
use IEEE.std_logic_1164.all;       -- for std_logic_vector
use IEEE.std_logic_arith.all;      -- for signed and unsigned
...
    signal k: std_logic_vector(7 downto 0):= "11110000";
    signal a, b: signed(7 downto 0);
    signal c: unsigned(15 downto 0);
    ...
    a <= conv_signed(100,8);       -- conversion function
    c <= conv_unsigned(65535,16);-- conversion function
    b <= signed'("00001111");      -- type casting
    a <= a + signed'(k);           -- type casting
```

LFSR – Linear Feedback Shift Register

Description styles ~~ abstraction levels

```
entity LFSR is
    port ( clock : in bit; Q1,Q2,Q3 : out bit );
end LFSR;
```



- **Q3 := Q2 || Q2 := Q1 + Q3 || Q1 := Q3**

LFSR behavior

```
entity LFSR is
  port ( clock: in bit; Q1,Q2,Q3: out bit );
end LFSR;
```

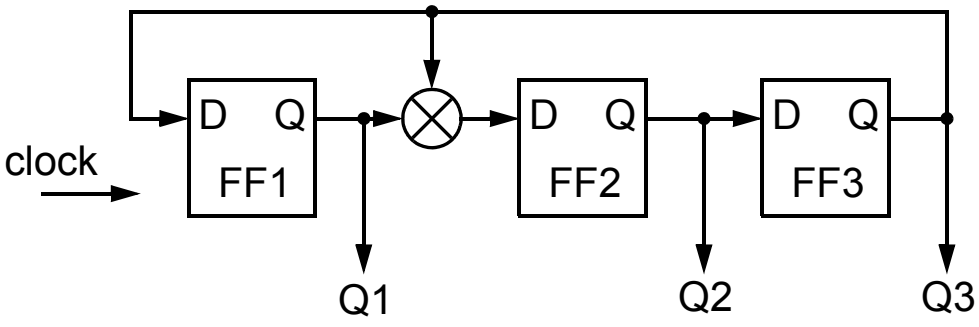
```
architecture Behavior of LFSR is
begin
    process
        variable Olek: bit_vector(3 downto 0):="0111";
    begin
        Q3 <= Olek(2) after 5 ns;
        Q2 <= Olek(1) after 5 ns;
        Q1 <= Olek(0) after 5 ns;
        wait on clock until clock = '1';
        Olek := Olek(2 downto 0) & '0';
        if Olek(3) = '1'
            then Olek := Olek xor "1011";
        end if;
    end process;
end Behavior;
```

LFSR dataflow

```
entity LFSR is
  port ( clock: in bit; Q1,Q2,Q3: out bit );
end LFSR;
```

```
architecture DataFlow of LFSR is
    signal FF1, FF2, FF3: bit := '1';
begin
b1: block (clock = '1' and not clock'stable)
    begin
        FF3 <= guarded FF2 after 5 ns;
        FF2 <= guarded FF1 xor FF3 after 5 ns;
        FF1 <= guarded FF3 after 5 ns;
    end block;
    Q3 <= FF3;
    Q2 <= FF2;
    Q1 <= FF1;
end DataFlow;
```


LFSR structure



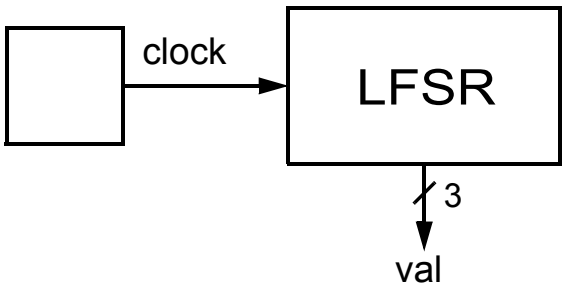
LFSR structure

```
entity LFSR is
  port ( clock: in bit; Q1,Q2,Q3: out bit );
end LFSR;
```

```
architecture Structure of LFSR is
    signal xor_out: bit;
    signal SR1, SR2, SR3: bit := '1';
    component FF
        port ( clock, data: in bit;  Q: out bit );
    end component;
    component XORgate
        port ( a, b: in bit;  x: out bit );
    end component;
begin
    FF1: FF port map ( clock, SR3, SR1 );
    FF2: FF port map ( clock, xor_out, SR2 );
    FF3: FF port map ( clock, SR2, SR3 );
    xor1: XORgate port map ( SR1, SR3, xor_out );
    Q3 <= SR3;    Q2 <= SR2;    Q1<= SR1;
end Structure;
```


LFSR test-bench

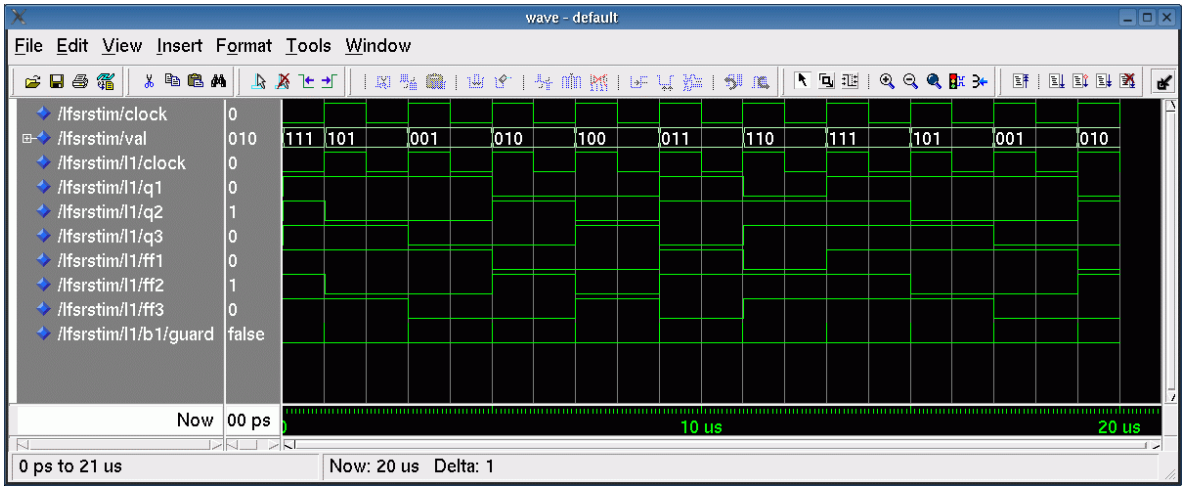
```
entity LFSRstim is
end LFSRstim;
```



LFSR test-bench

```
architecture test of LFSRstim is
    component LFSR
        port ( clock: in bit;  Q1, Q2, Q3: out bit );
    end component;
    signal clock: bit := '0';
    signal val: bit_vector (3 downto 1);
begin
    L1: LFSR port map ( clock, val(1), val(2), val(3) );
    process begin
        for I in 1 to 20 loop
            wait for 1 us;    clock <= not clock;
        end loop;
        wait;
    end process;
end test;
```


LFSR simulation results



LFSR configuration

```
configuration STRUCTconf of LFSRstim is
    for test
        for L1: LFSR use entity WORK.LFSR(Structure);
            for Structure
                for all: FF use entity WORK.FF(Behavior);
                end for;
                for xor1: XORgate use
                    entity WORK.XORgate(DataFlow);
                end for;
            end for;
        end for;
    end for;
end STRUCTconf;
```


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

- 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

Using attributes - #1

- **Inflexible code**

```
signal s: bit_vector (7 downto 0);
...
for i in 0 to 7 loop ...
```

- **More flexible code**

```
signal s: bit_vector (sz-1 downto 0);
...
for i in 0 to sz-1 loop ...
```

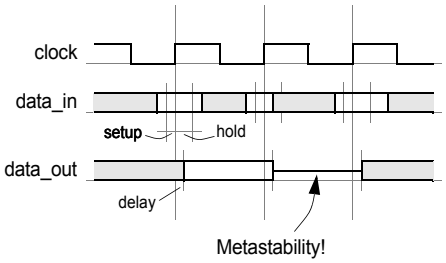
- **Most flexible code**

```
signal s: bit_vector (7 downto 0);
...
for i in s'low to s'high loop ...
for i in s'reverse_range loop ...
```

Using attributes - #2

- Setup & hold times in memory elements (triggers)
  - setup - the input data must be fixed for certain time before the active clock flank
  - hold - the input data must be stable for certain time after the active clock flank
- Caused by non-equal signal paths in memory elements
- Violation may cause metastability -- output between '0' and '1'

```
process (clock,data_in) begin
  if clock'event and clock='1' then
    assert data_in'last_event >= 3 ns
      report "setup time violation" severity warning;
    data_out <= data_in after 3 ns;
  end if;
  if data_in'event and clock='1' then
    assert clock'last_event >= 5 ns
      report "hold time violation" severity warning;
  end if;
end process;
```



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'

VHDL timing

- The “wait” statement
  - *wait on* sensitivity list
  - *wait until* condition
  - *wait for* time_expression
- Process’ sensitivity list
- Simulation Engine
  - simulation cycles – new values are assigned to signals in the end of the cycle
  - concurrency – the process execution order is not important
- Delta delay (Δ-delay)
  - event after an event at the same time moment
- Inertial/transport delay

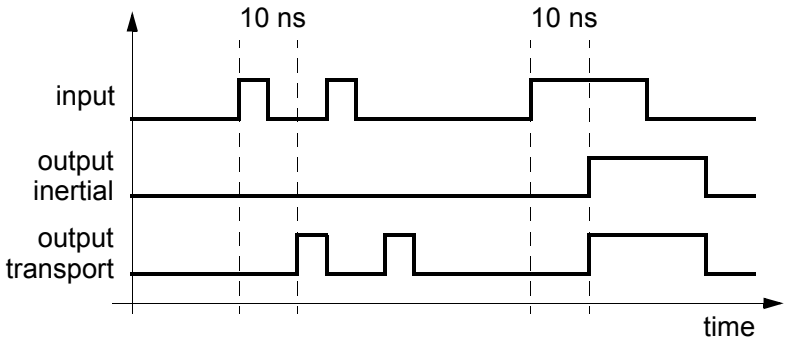
Timing control

- 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 change on signal *a* only
    - wait until a=’1’ and b=’0’; -- sensitive to changes on 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**
  - **sub-models 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

  - **Modularity features – functions & procedures**
  - **Partitioning features – libraries, packages, components, blocks, configurations, ...**

Elements of *entity*

- **Declarations, generic and port clauses, etc.**

```
entity_declaration ::=
    entity identifier is
      [ generic ( formal_generic_interface_list ) ; ]
      [ port ( formal_port_interface_list ) ; ]
      { entity_declarative_part }
  [ begin
      { concurrent_assertion_statement
      | passive_concurrent_procedure_call
      | passive_process_statement } ]
    end [ entity ] [ identifier ] ;
```


TTÜ1918
Department of Computer Engineering

Generics – a way to pass parameters

```
-- Address generator - entity
library IEEE;
use IEEE.std_logic_1164.all
use IEEE.std_logic_arith.all;
entity agener is
  generic ( bitwidth: positive );
  port (
    clock: in bit;
    reset, enable: in std_logic;
    start_address, stop_address: in unsigned(bitwidth-1 downto 0);
    address: out unsigned(bitwidth-1 downto 0) );
end agener;

  -- ... and somewhere in the architecture
  signal count: unsigned(bitwidth-1 downto 0);
```

Architecture

- Architecture – declarations, statements
  - Process statement
  - Concurrent signal assignment
  - Component instantiation statement
  - Concurrent procedure call
  - Generate statement
  - Concurrent Assertion Statement
  - Block statement

Process

- Behavior of the model
- Contains timing control
  - either wait statement(s) or sensitivity list must exist in a process
- 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

```
name1: process
  variable tmp: bit;
begin
  wait on a, b;
  tmp := a and b;
  wait for 5 ns;
  x <= tmp;
end process name1;
```

Equivalent processes

- **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**

```
[label:] if conditional-expression then statements...
elsif conditional-expression then statements...
else statements...
end if [label];
```

  - *conditional-expression* - must return boolean value

- **case**

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

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 contain timing control statements
  - input parameters only (as constants)
  - operator overloading
  - resolution functions – multiple drivers of a signal
- Procedure
  - used as a statement (sequential or concurrent) – can contain timing control statements
  - input parameters (constants)
  - output parameters (variables/signals)
- Declaration (prototype)
  - package or declarative part of architecture, process, function, procedure, etc.
- Content (body)
  - package body
  - declarative part of architecture, process, function, etc. (together with declaration)

Functions

```
-- Conversion version example
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;

-- Operator overloading example
function "and" (l,r: signed) return signed is begin
  return signed(std_logic_vector(l) and std_logic_vector(r));
end;


-- Architecture - declarative part
  signal a, b, x: signed (7 downto 0);
  signal y: boolean;

-- Architecture/process/... - body
  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

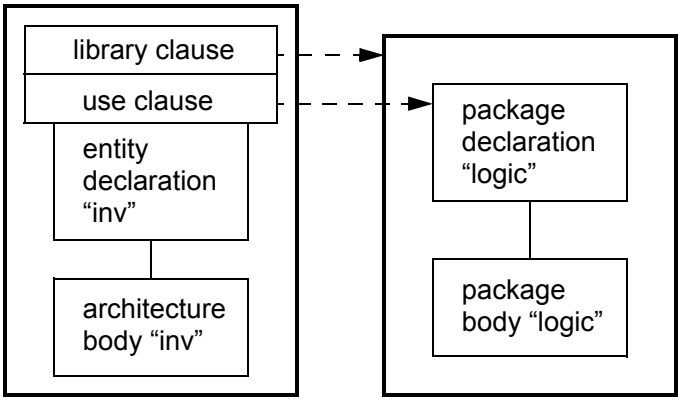
Packages

- 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**

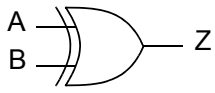
```
architecture toparch of topunit is
    component child1
        port( ... );
    end component;
    component child2 ...
begin
    COMP1: child1 port map(...);
    COMP2: child2 port map(...);
end toparch;
```

Structural replication

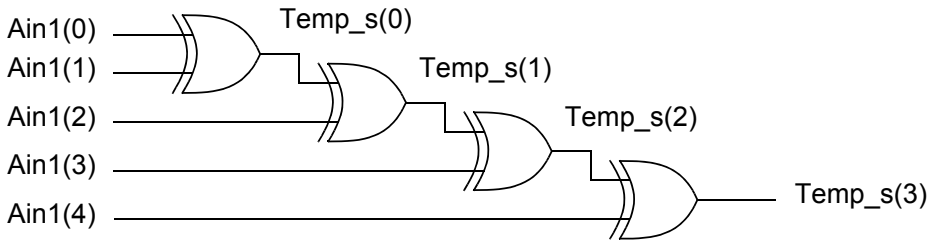
- A *generate* statement provides a mechanism for iterative or conditional elaboration of a portion of a description
- Typical application - instantiation and connecting of multiple identical components (half adders to make up full adder, trees of components etc.)

```
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

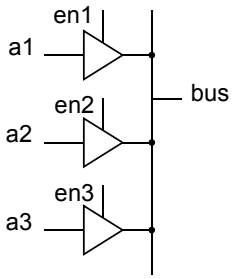



```
entity XOR_Nty is
  port ( A : in Std_Logic;
         B : in Std_Logic;
         Z : out Std_Logic );
end XOR_Nty;
```



Resolution functions

- **Multiple drivers of a signal**

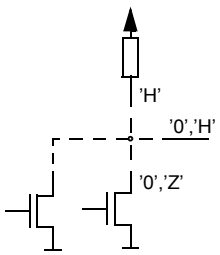



```
type tri_value is ('0','1','Z');
type tri_val_vector is array (natural range <>) of tri_value;
function resolve_tri (b:tri_val_vector) return tri_value is ...
    ...
signal bus: resolve_tri tri_value;
```

- **Driving values – 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;
```



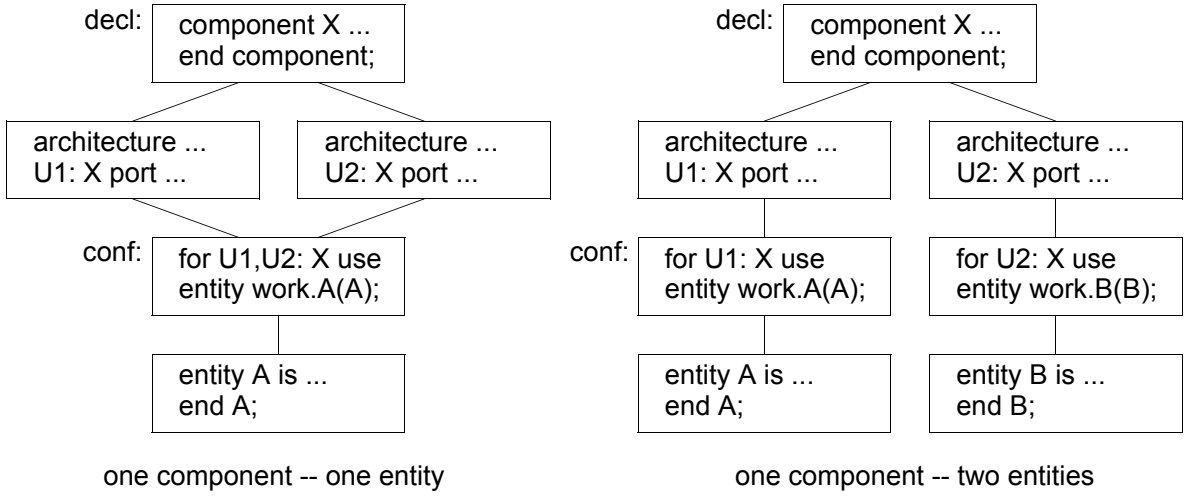
Resolution function – how to use

```
library IEEE;    use IEEE.std_logic_1164.all;
entity driver is
  port ( i1: in std_logic_vector(1 downto 0);
         o1, o2: inout std_logic_vector(1 downto 0) );
end entity driver;
library IEEE;    use IEEE.std_logic_1164.all;
architecture driving of driver is begin
  o1 <= i1;    o2 <= not i1;
end architecture driving;
-------------------------------------------------------------------
entity test is    end entity test;
library IEEE;    use IEEE.std_logic_1164.all;
architecture bench of test is
  signal i1, i2, o1, o2: std_logic_vector(1 downto 0);
  component driver
    port ( i1: in std_logic_vector(1 downto 0);
           o1, o2: inout std_logic_vector(1 downto 0) );
  end component;
begin
  process begin
    i1 <= "01";    i2 <= "01";    wait for 10 ns;
    i1 <= "01";    i2 <= "11";    wait for 10 ns;
    i1 <= "ZZ";    i2 <= "01";    wait for 10 ns;
    i1 <= "10";    i2 <= "ZZ";    wait for 10 ns;
    i1 <= "HL";    i2 <= "00";    wait for 10 ns;
    i1 <= "11";    i2 <= "HL";    wait for 10 ns;
    i1 <= "HZ";    i2 <= "LZ";    wait for 10 ns; wait;
  end process;
  u1: driver port map (i1, o1, o2);
  u2: driver port map (i2, o1, o2);
end architecture bench;
```

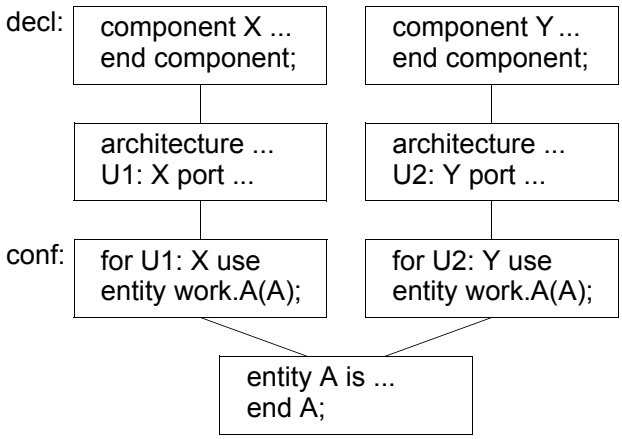
Configuration declaration

- **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



Example #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)
```


Example #4 (http://www10.edacafe.com/book/ASIC/ASICs.php)

- components

```
entity AD2 is port (A1, A2: in BIT; Y: out BIT); end;
architecture B of AD2 is begin Y <= A1 and A2; end;
entity XR2 is port (X1, X2: in BIT; Y: out BIT); end;
architecture B of XR2 is begin Y <= X1 xor X2; end;
```

- component declaration & configuration specification

```
entity Half_Adder is port (X, Y: BIT; Sum, Cout: out BIT); end;
architecture Netlist of Half_Adder is use work.all;
  component MX port (A, B: BIT; Z:out BIT); end component;
  component MA port (A, B: BIT; Z:out BIT); end component;
  for G1:MX use entity XR2(B) port map(X1 => A,X2 => B,Y => Z);
begin
  G1:MX port map (X, Y, Sum); G2:MA port map (X, Y, Cout);
end;
```

- configuration declaration, block configuration, component configuration

```
configuration C1 of Half_Adder is
  use work.all;
  for Netlist
    for G2:MA
      use entity AD2(B) port map(A1 => A,A2 => B,Y => Z);
    end for;
  end for;
end;
```

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