



VHDL - VHSIC Hardware Description Language

VHSIC - Very High Speed Integrated Circuit

- **VHDL is**
 - **Non-proprietary language (IEEE-1076 1987/1993)**
 - **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), development continues**
 - **VHDL-2000 (2000&2002), VHDL-200X (ongoing), VHDL-AMS (1076.1-1999)**



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



Types, packages

- ***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 which allows the specification of groups of logically related declarations.**



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**
- **A *class* is relevant to the nature of the object and represents HOW the object is used in the model**



Hello, world!

```
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!

- Synopsys

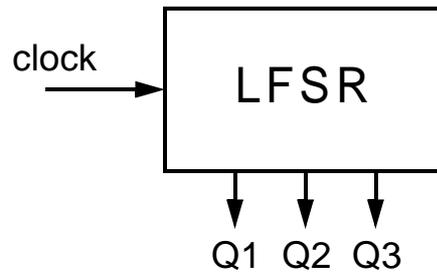
```
# run
0 NS
Assertion NOTE at 0 NS in design unit TEST(HELLO)
  from process /TEST/_P0
    "Hello, world!"
(vhdlsim): Simulation complete, time is 0 NS.
#
```

- ModelSim

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

LFSR behavior

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



- $Q3 := Q2 \parallel Q2 := Q1 + Q3 \parallel Q1 := Q3$

LFSR behavior

```
architecture Kaitumine 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 Kaitumine;
```



LFSR dataflow

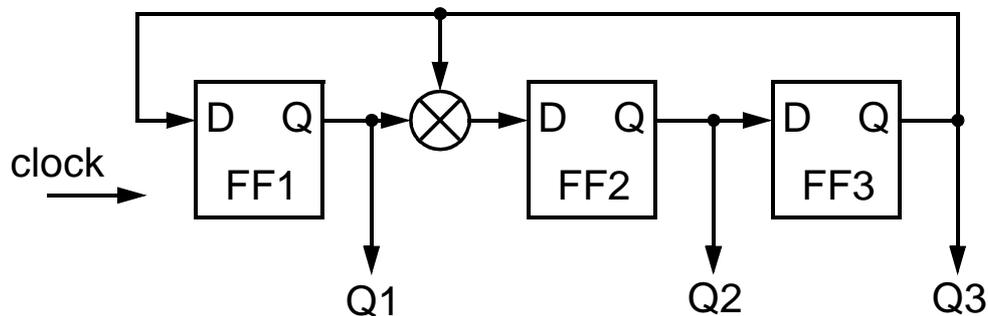
```

architecture Andmevoog 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 Andmevoog;

```



LFSR structure





LFSR structure

```

architecture Struktuur 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 Struktuur;

```

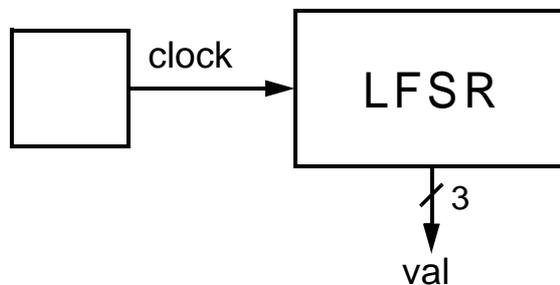


LFSR test-bench

```

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 configuration

```

configuration STRUCTconf of LFSRstim is
  for test
    for L1: LFSR use entity WORK.LFSR(Struktuur);
      for Struktuur
        for all: FF use entity WORK.FF(Kaitumine);
        end for;
        for xor1: XORgate use
          entity WORK.XORgate(Andmevoog);
        end for;
      end for;
    end for;
  end for;
end STRUCTconf;

```



Design Units

	Design unit	Comments
1	ENTITY	Interface specification
2	ARCHITECTURE	function or composition of an entity
3	PACKAGE DECLARATION	declarations, subprograms
4	PACKAGE BODY	subprogram bodies
5	CONFIGURATION	binding architectures/entities



VHDL writing style

- **Comments** (-- this is a comment)
- **Header**
- **Generic**
- **Indentation**
- **Line lengths (80 characters)**
- **Statements per line (1)**
- **Declarations per line (1)**
- **Alignment of declarations**



Header

- **Project name**
- **File name**
- **Title of the design unit (DU)**
- **Description of DU, purpose and limitations**
- **Intended design library**
- **Analysis dependencies (packages, components)**
- **Model initialization (Resets, initial values)**
- **Notes or other items (synthesis aspects)**
- **Author(s) and full address(es) (+ e-mail)**
- **Simulator used (version NO)**
- **Revision list (author, date, changes etc.)**



Example

```

-----
-- Project           : ATEP CLASS
-- File name        : count_e.vhd
-- Title            : COUNTER_Nty
-- Description      : Counter Entity Description
-- Design Library   : Atep_Lib
-- Analysis Dependency: none, does not require any library
-- Simulator(s)    : Model Technology 4.2f on PC
--                 : Cadence Leapfrog on Sun workstation
-- Initialization   : Model does not include a RESET. Initialization
--                 : is dependent on port initialization values.
-- Notes           : This model is not designed for synthesis.
-----
-- Revisions      :
--           Date           Author  Revision    Comments
-- Tue Jul 19 22:52:20 1994  cohen   Rev A      Creation
--                                     VhdlCohen@aol.com
-----

```

NAND example

```
entity NAND_GATE is
  port(A,B: in bit; Y: out bit);
end NAND_GATE;
```

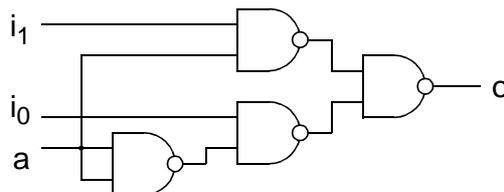
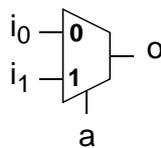
```
entity AND_GATE is
  port(A,B: in bit; Y: out bit);
end AND_GATE;
```

```
architecture Beh of NAND_GATE is
  signal Pass_bit: bit := '1';
begin
  Pass_bit <= transport A and B;
  Y <= transport not Pass_bit;
end Beh;
```

```
architecture Str of NAND_GATE is
  signal Pass_bit: bit := '1';
  component AND_GATE
    port(A,B:in bit; Y:out bit);
  end component
begin
  AND_Phase: AND_GATE
    port map (A, B, Pass_bit);
  Y <= transport not Pass_bit;
end Str;
```

MUX

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

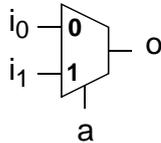




MUX

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;
```

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

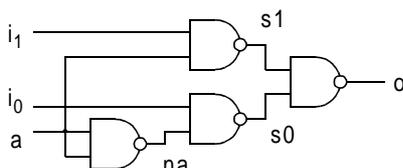


MUX

dataflow

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

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



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;
```



VHDL Elements

- **Basic language elements**
- **Configuration**
- **Elements of entity / architecture**
- **Subprograms**
- **Packages**
- **Control structures**
- **Drivers**
- **VHDL timing**



Basic language elements

- **Lexical elements, Identifiers**
- **Syntax**
 - **Delimiters, Literals, Operators and Precedence**
- **Types and Subtypes**
 - **Scalar type - integer, enumerate (IEEE Std. 1164), physical, real**
 - **Composite type - arrays, records**
 - **Access type**
- **File**
- **Attributes**
- **Aliases**



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 := (E1 => 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;



Identifiers

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



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
 - 'A' '**' -- character literals
 - "This is a string" -- string literal
 - X"FFF" B"1111" -- bit string literal



Operators

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



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 ... to ...; 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;
```



Physical type

```
type Time is range -(2**31-1) to (2**31-1)
  units
    fs; -- femtosecond
    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 type

- Composite consists of arrays and records.
- An *array* is a type, the value of that consists of elements that are all the same type (subtype).

```
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;
```

- A *record* is a composite type whose values consist of named elements.

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



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);
      ...
    end loop;
  end process;
end architecture;
```



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.

```

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

```