VHDL-based synthesis in XST

Peter Marwedel
Informatik XII, U. Dortmund
Xilinx Synthesis Technology (XST)

- Einführung
- SystemC
  - Vorlesungen und Programmierung
- FPGAs
  - Vorlesungen
  - VHDL-basierte Konfiguration von FPGAs mit dem XUP VII Pro Entwicklungssystem
- Algorithmen
  - Mikroarchitektur-Synthese
  - Automatensynthese
  - Logiksynthese
  - Layoutsynthese

- Heute: VHDL-basierte Synthese in XST
  Quelle: //toolbox.xilinx.com/docsan/xilinx9/books/docs/xst/xst.pdf Chapter 6
- Buch: J. Reichardt/B. Schwarz: VHDL-Synthese, Oldenbourg, 2000
VHDL-based synthesis in XST

(VHDL-based) Synthesis: VHDL-model $\rightarrow$ netlist
where netlist=$(\text{components, nets})$ and
components=$\{\text{gates, flip-flops, Xilinx-library elements}\}$

- Ideally, all VHDL language elements would be supported in XST.
- In practice, this is hardly feasible, since VHDL was designed for simulation, not for synthesis (same situation as for SystemC).

♫ Synthesis subset of VHDL!
VHDL IEEE support

XST supports:

- VHDL IEEE std 1076-1987
- VHDL IEEE std 1076-1993
- VHDL IEEE std 1076-2006 (partially implemented) *

VHDL IEEE Conflicts


- In cases where: VHDL IEEE std 1076-1993 requires a construct to be an erroneous case, but VHDL IEEE std 1076-1987 accepts it, XST issues a warning instead of an error.
1. Combinatorial circuits

Combinatorial circuits can be described using

- Simple signal assignments
  Example:
  \( t \leq a \text{ and } b; \) -- after clause ignored
  -- (not precisely synthesizable)!

- Selected signal assignments
- Conditional signal assignments
- Generate statements
- Combinatorial processes
Selected signal assignment

Example (multiplexer description):
library IEEE;
use IEEE.std_logic_1164.all;
entity select_bhv is
  generic (width: integer := 8);
  port (a, b, c, d: in std_logic_vector (width-1 downto 0);
        selector: in std_logic_vector (1 downto 0);
        T: out std_logic_vector (width-1 downto 0) );
end select_bhv;
architecture bhv of select_bhv is
begin
  with selector select
  T <= a when "00",
       b when "01",
       c when "10",
       d when others; -- all cases to be covered (otherwise sequential)
end bhv;
Conditional signal assignment

Example (multiplexer description):
library IEEE;
use IEEE.std_logic_1164.all;
entity select_bhv is
  generic (width: integer := 8);
  port (a, b, c, d: in std_logic_vector (width-1 downto 0);
    selector: in std_logic_vector (1 downto 0);
    T: out std_logic_vector (width-1 downto 0) );
end select_bhv;
architecture bhv of select_bhv is
begin
  with selector select
    T <= a when selector="00" else
      b when selector="01" else
      c when selector="10" else
      d; -- all cases to be covered (otherwise sequential)
end bhv;
Generate statement

Example:

```vhdl
entity EXAMPLE is
    port ( A,B : in BIT_VECTOR (0 to 7);  CIN : in BIT;
        SUM : out BIT_VECTOR (0 to 7); COUT : out BIT);
end EXAMPLE;
architecture ARCHI of EXAMPLE is
    signal C : BIT_VECTOR (0 to 8);
begin
    C(0) <= CIN;
    COUT <= C(8);
    LOOP_ADD : for I in 0 to 7 generate
        SUM(I) <= A(I) xor B(I) xor C(I);
        C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
end ARCHI;
-- VHDL code generates 8-bit adder
```
If `<condition>` generate

entity EXAMPLE is
    generic (N:INTEGER := 8);
    port ( A,B : in BIT_VECTOR (0 to 7);  CIN : in BIT;
        SUM : out BIT_VECTOR (0 to 7); COUT : out BIT);
end EXAMPLE;
architecture ARCHI of EXAMPLE is
    signal C : BIT_VECTOR (0 to 8);
begin
    L1: if (N>=4 and <=32) generate
        C(0) <= CIN;
        COUT <= C(8);
        LOOP_ADD : for I in 0 to N generate
            SUM(I) <= A(I) xor B(I) xor C(I);
            C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
        end generate;
    end generate;
end ARCHI;
Combinatorial Process

Treatment of signal assignments within process different from concurrent signal assignments:
The value assignments are made in a sequential mode. Assignments may cancel previous ones. Example:

```
entity EXAMPLE is port ( A, B : in BIT; S : out BIT);
end EXAMPLE;
architecture ARCHI of EXAMPLE is
  begin
    process ( A, B )
      begin
        S <= '0' ; S<=(A and B);
        S <= '1' ;
      end process;
  end ARCHI;
First the signal S is assigned to 0, but later, the value S is changed to 1.
```
Combinatorial Process

- A process is called **combinatorial** when its inferred HW does not involve any memory elements \((\forall\text{ assigned signals } \in \text{ process}, \forall\text{ paths}: \exists\text{ explicit assignment})\).

- A combinatorial process has a sensitivity list.

- For a combinatorial process, this sensitivity list **must contain all signals** which appear in conditions (if, case, etc.), and any signal appearing on the right hand side of an assignment. **If one or more signals are missing** from the sensitivity list, XST generates a warning for the missing signals, and **adds them to the sensitivity list**. **In this case, the result of the synthesis may be different from the initial design specification (!!!).**
Local variables

A **process** may contain local variables. Due to the absence of delays, variables are handled like signals. **Example:**

```vhdl
library ASYL; use ASYL.ARITH.all;
entity ADDSUB is
    port ( A, B : in BIT_VECTOR (3 downto 0); ADD_SUB : in BIT;
        S : out BIT_VECTOR (3 downto 0));
end ADDSUB;
architecture ARCHI of ADDSUB is
begin
    process ( A, B, ADD_SUB )
        variable AUX : BIT_VECTOR (3 downto 0);
    begin
        if ADD_SUB = ’1’ then AUX := A + B ; else AUX := A - B ; end if;
        S <= AUX;
    end process;
end ARCHI;
```
if ... else statements in combinational processes

library IEEE; use IEEE.std_logic_1164.all;

entity mux4 is
  port (a, b, c, d: in std_logic_vector (7 downto 0);
        sel1, sel2: in std_logic;
        outmux: out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
  process (a, b, c, d, sel1, sel2)
  begin
    if (sel1 = '1') then
      if (sel2 = '1') then
        outmux <= a;
      else
        outmux <= b;
      end if;
    else
      if (sel2 = '1') then
        outmux <= c;
      else
        outmux <= d;
      end if;
    end if;
  end process;
end behavior;

There must be assignments to outmux for all execution paths; otherwise the process would not be combinatorial.
Case statements in combinational processes

Important to avoid sequential behavior.
Example:

```vhdl
library IEEE; use IEEE.std_logic_1164.all;
entity mux4 is port (a, b, c, d: in std_logic_vector (7 downto 0);
    sel: in std_logic_vector (1 downto 0);
    outmux: out std_logic_vector (7 downto 0));
end mux4;
architecture behavior of mux4 is
begin
    process (a, b, c, d, sel)
    begin
        case sel is
            when "00" => outmux <= a;
            when "01" => outmux <= b;
            when "10" => outmux <= c;
            when others => outmux <= d;-- case statement must be complete
        end case;
    end process;
end behavior;
```
For .. Loop statements in combinational processes

The **for ... loop** statement is supported for:

- Constant bounds
- Stop test condition using operators <, <=, > or >=
- Next step computation in one of the specifications:
  - \( var = var + step \)
  - \( var = var - step \)
  (where \( var \): loop variable, \( step \): a constant value).
- **next** and **exit** statements are supported.
for .. loop statements: Example

library IEEE; use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity countzeros is
  port (a: in std_logic_vector (7 downto 0);
     Count: out std_logic_vector (2 downto 0));
end countzeros;

architecture behavior of countzeros is
signal Count_Aux: std_logic_vector (2 downto 0);
begin process (a)
begin
  Count_Aux <= "000";
  for i in a'range loop
    if (a[i] = '0') then
      Count_Aux <= Count_Aux + 1; -- + defined in std_logic_unsigned
    end if;
  end loop;
  Count <= Count_Aux;
end process; end behavior;
Sequential circuits

A process is sequential when it is not a combinatorial process
(∃ signal assigned somewhere, ∃ path:
→ ∃ assignment to this signal on this path)

→ Generated HW has an internal state or memory.

Sequential circuits can be described using sequential processes.
2 types of descriptions are allowed by XST:
1. sequential processes with a sensitivity list
2. sequential processes without a sensitivity list
Sequential processes with a sensitivity list

Template with asynchronous & synchronous parts

```
process ( CLK, RST ) ...
begin
  if RST = <'0' | '1'> then
    -- an asynchronous part may appear here -- optional part
  elsif <CLK'EVENT | not CLK'STABLE> and CLK = <'0' | '1'> then
    -- synchronous part sequential statements may appear here
  end if;
end process;
```

**Note:** Asynchronous signals must be declared in the sensitivity list. Otherwise, XST generates a warning and adds them to the sensitivity list. In this case, the behavior of the synthesis result may be different from the initial specification.
Sequential processes without a sensitivity list - single wait statement -

Inference of registers:
Sequential processes w/o a sensitivity and exactly 1 "wait" statement. It must be the first statement. The condition must be a condition on the clock signal. An asynchronous part can not be specified.

Example:
process ...
  begin
    wait until <CLK’EVENT | not CLK’ STABLE> and CLK = <'0' | '1'>;
    ... -- a synchronous part may be specified here.
  end process;
Description of registers

```vhdl
entity EXAMPLE is port ( DI : in BIT_VECTOR (7 downto 0);
    CLK : in BIT; DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process begin
    wait until CLK'EVENT and CLK = '1';
    DO <= DI ;
  end process;
end ARCHI;
```

No sensivity list

```vhdl
entity EXAMPLE is port ( DI : in BIT_VECTOR (7 downto 0);
    CLK : in BIT; DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( CLK ) begin
    if CLK'EVENT and CLK = '1' then DO <= DI ; end if;
  end process;
end ARCHI;
```

With sensivity list
library ASYL; use ASYL_PKG_ARITH.all;

entity EXAMPLE is
  port ( CLK : in BIT; RST : in BIT;
          DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process ( CLK, RST )
    variable COUNT : BIT_VECTOR (7 downto 0);
  begin
    if RST = '1' then COUNT := "00000000";
    elsif CLK'EVENT and CLK = '1' then
      COUNT := COUNT + "00000001";
    end if;
    DO <= COUNT;
  end process;
end ARCHI;
Sequential processes without a sensitivity list - multiple wait statements -

Sequential circuits can be described with multiple wait statements in a process. When using XST, several rules must be respected:

- The process must only contain one loop statement.
- The first statement in the loop must be a wait statement.
- After each wait statement, a next or exit statement must be defined.
- The condition in the wait statements must be the same for each wait statement.
- This condition must use only one signal—the clock signal.
- This condition must have the following form:

```plaintext
wait [on <clock_signal>] until [(<clock_signal>’EVENT | not <clock_signal>’STABLE) and ] <clock_signal> = <’0’ | ’1’>-;
```
Example

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity EXAMPLE is port (CLK : in STD_LOGIC; RST : in STD_LOGIC 
(DATA1,DATA2,DATA3,DATA4:in STD_LOGIC_VECTOR 
(3 downto 0);RESULT:out STD_LOGIC_VECTOR(3 downto 0); );
end EXAMPLE;
architecture ARCH of EXAMPLE is
begin process begin
  SEQ_LOOP : loop
    wait until CLK'EVENT and CLK = '1';
    exit SEQ_LOOP when RST = '1';
    RESULT <= DATA1;
    wait until CLK'EVENT and CLK = '1';
    exit SEQ_LOOP when RST = '1';
    RESULT <= DATA2;
    wait until CLK'EVENT and CLK = '1';
    exit SEQ_LOOP when RST = '1';
    RESULT <= DATA3;
    wait until CLK'EVENT and CLK = '1';
    exit SEQ_LOOP when RST = '1';
    RESULT <= DATA4;
  end loop; end process; end ARCH;

obviously considered sequential even though values assigned to output in all cases
Functions and procedures

The declaration of a function or a procedure provides a mechanism for handling blocks used multiple times in a design.

Functions and procedures can be declared in the declarative part of an entity, in an architecture, or in packages.

- Parameters can be unconstrained. The content is similar to the combinatorial process content.
- Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package.
- Recursive function/procedure calls are not supported.
package pkg is function add (a,b, c in : bit ) return bit_vector;
end pkg;
package body pkg is
function add (a,b, c in : bit ) return bit_vector is
  variable s, cout : bit; variable result : bit_vector (1 downto 0);
  begin
    s := a xor b xor cin;
    cout := (a and b) or (a and cin) or (b and cin);
    result := cout & s; return result;
  end add; end pkg;
use work.pkg.all; entity example is
port ( a,b : in bit_vector (3 downto 0); cin : in bit;
  s : out bit_vector (3 downto 0); cout: out bit );
end example;
architecture archi of example is
  signal s0, s1, s2, s3 : bit_vector (1 downto 0);
  begin
    s0 <= add ( a(0), b(0), cin ); s1 <= add ( a(1), b(1), s0(1) );
    s2 <= add ( a(2), b(2), s1(1) ); s3 <= add ( a(3), b(3), s2(1) );
    s <= s3(0) & s2(0) & s1(0) & s0(0); cout <= s3(1);
end archi;

The "ADD" function is a single bit adder. This function is called 4 times with the proper parameters to create a 4-bit adder.
State Machines

XST proposes a large set of templates to describe Finite State Machines (FSMs). By default, XST tries to recognize FSMs from VHDL/Verilog code, and apply several state encoding techniques (it can re-encode the user’s initial encoding) to get better performance or less area. However, FSM extraction can be disabled using an `FSM_extract` design constraint.

XST can handle only synchronous state machines. Example:
library IEEE; use IEEE.std_logic_1164.all;

entity fsm is
  port (clk, reset, x1: IN std_logic; outp: OUT std_logic);
end entity;

architecture beh1 of fsm is
  type state_type is (s1, s2, s3, s4);
  signal state: state_type;
begin
  process (clk, reset)
  begin
    if (reset = '1') then
      state <= s1; outp <= '1';
    elsif (clk = '1' and clk'event) then
      case state is
        when s1 => if x1 = '1' then state <= s2; else state <= s3; end if;
               outp <= '1';
        when s2 => state <= s4; outp <= '1';
        when s3 => state <= s4; outp <= '0';
        when s4 => state <= s1; outp <= '0';
      end case;
    end if;
  end process;
end beh1;

Please note, in this example output signal "outp" is a register.
library IEEE; use IEEE.std_logic_1164.all;
entity fsm is port (clk, reset, x1 : IN std_logic; outp : OUT std_logic);
end entity;
architecture beh1 of fsm is
type state_type is (s1,s2,s3,s4);
signal state: state_type;
begin process1: process (clk,reset)
begin
  if (reset =’1’) then state <=s1;
  elsif (clk=’1’ and clk’Event) then
    case state is
      when s1 => if x1=’1’ then state <= s2;
      else state <= s3; end if;
      when s2 => state <= s4;
      when s3 => state <= s4;
      when s4 => state <= s1;
    end case;
  end if;
end process process1;
process2 : process (state)
begin
  case state is
    when s1 => outp <= ’1’;
    when s2 => outp <= ’1’;
    when s3 => outp <= ’0’;
    when s4 => outp <= ’0’;
  end case;
end process process2;
end beh1;
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm is
port ( clk, reset, x1 : IN std_logic;
outp : OUT std_logic);
end entity;

architecture beh1 of fsm is

begin
signal state, next_state: state_type;

process1: process (clk,reset)
begin
if (reset =’1’) then
state <=s1;
elsif (clk=’1’ and clk’Event) then
state <= next_state;
end if;
end process process1;

process3 : process (state)
begin
case state is
when s1 => outp <= ’1’;
when s2 => outp <= ’1’;
when s3 => outp <= ’0’;
when s4 => outp <= ’0’;
end case;
end process;

end beh1;
end architecture;

process2 : process (state, x1)
begin

case state is
when s1 => if x1=’1’ then next_state <= s2;
    else next_state <= s3; end if;
when s2 => next_state <= s4;
when s3 => next_state <= s4;
when s4 => next_state <= s1;
end case;
end process;

end
State Registers

- State Registers must be initialized with an asynchronous or synchronous signal. XST does not support FSMs without initialization signals.

- In VHDL the type of state registers can be of different types: integer, bit_vector, std_logic_vector, for example. It is common and convenient to define an enumerated type containing all possible state values and to declare a state register with that type.
Coding styles for FSMs

Next State Equations
Can be described directly in the sequential process or in a distinct combinational process. The simplest template is based on a Case statement. If using a separate combinational process, its sensitivity list should contain the state signal and all FSM inputs.

FSM Outputs
Non-registered outputs are described either in the combinational process or concurrent assignments. Registered outputs must be assigned within the sequential process.
Packages

XST provides full support for packages. To use a given package, the following lines must be included at the beginning of the VHDL design:

```vhdl
library lib_pack;
-- lib_pack is the name of the library specified where the
-- package has been compiled (work by default)
use lib_pack.pack_name.all;
-- pack_name is the name of the defined package.
```

XST also supports predefined packages; these packages are pre-compiled and can be included in VHDL designs. These packages are intended for use during synthesis, but may also be used for simulation.
Supported packages

**STANDARD Package:**
- contains basic types (bit, bit_vector, and integer).
- Included by default.

**IEEE Packages supported.**
- std_logic_1164: defines types std_logic, std_ulogic, std_logic_vector, std_ulogic_vector, and conversion functions based on these types.
- numeric_bit: supports types unsigned, signed vectors based on type bit, and all overloaded arithmetic operators on these types, conversion and extended functions for these types.
- numeric_std: supports types unsigned, signed vectors based on type std_logic. Equivalent to std_logic_arith.
- math_real: real number constants $e$, $1/e$, $\pi$, $2\pi$, $1/\pi$, … and functions $\sin(x)$, $\cos(x)$, …
Supported packages

Synopsys Packages supported in the IEEE library:

- std_logic_arith: supports types unsigned, signed vectors, and all overloaded arithmetic operators on these types. It also defines conversion and extended functions for these types.
- std_logic_unsigned: defines arithmetic operators on std_ulogic_vector and considers them as unsigned operators.
- std_logic_signed: defines arithmetic operators on std_logic_vector and considers them as signed operators.
- std_logic_misc: defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package (and_reduce, or_reduce, ...)

Objects in VHDL

- **Signals**: can be declared in architecture declarative part & used anywhere within the architecture. Can also be declared and used within a block.

- **Variables**: declared in a process, subprogram, or architecture (shared variable, VHDL’93); **Shared variables** allowed only to denote RAM

- **Constants**: can be declared and used within any region. Their value cannot be changed.

- **Files, Alias, Components**: supported

- **Attributes**: some support
Basic types accepted by XST

- **Enumerated Types:**
  - BIT ('0','1'); BOOLEAN (false, true)
  - STD_LOGIC ('U','X','0','1','Z','W','L','H','-').
    For XST synthesis, the '0' and 'L' values are treated identically, as are '1' and 'H'. The 'X', and '-' values are treated as don’t care. The 'U' and 'W' values are not accepted by XST. The 'Z' value is treated as high impedance.
  - User defined enumerated type:
    type COLOR is (RED,GREEN,YELLOW);

- **Bit Vector Types:**
  - BIT_VECTOR, STD_LOGIC_VECTOR
  - Unconstrained types are not accepted

- **Integer Type:** INTEGER

- **Predefined types:** BIT, BOOLEAN, BIT_VECTOR, INTEGER, REAL

- The following types are declared in the **STD_LOGIC_1164** IEEE Package: STD_LOGIC, STD_LOGIC_VECTOR
Multi-dimensional Array Types

- XST: multi-dimensional array types ≤ 3 dimensions.
- Arrays can be signals, constants, or VHDL variables.
- Assignments & arithmetic operations with arrays allowed.
- Multidimensional arrays can be passed to functions, & used in instantiations.
- Arrays must be fully constrained in all dimensions.

Example:

```vhdl
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```
Example

*Example*  

```vhdl
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);  
type TAB12 is array (4 downto 0) of WORD8;  
type TAB03 is array (2 downto 0) of TAB12;  
signal WORD_A : WORD8;  
signal TAB_A, TAB_B : TAB05;  
signal TAB_C, TAB_D : TAB03;  
constant CST_A : TAB03 := (  
("0000000","0000001","0000010","0000011","0000100")  
("0010000","0010001","0010010","0100011","0010100")  
("0100000","0100001","0100010","0100011","0100100");
```

A multi-dimensional array signal or variable can be completely used:

```
TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CST_A;
```

Just an index of one array can be specified:

```
TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
```

Just indexes of the maximum number of dimensions can be specified:

```
TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
```
Example

subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CST_A : TAB03 := (
("0000000","0000001","0000010","0000011","0000100")
("0010000","0010001","0010010","0100011","0010100")
("0100000","0100001","0100010","0100011","0100100");

Just a slice of the first array can be specified:
TAB_A (4 downto 1) <= TAB_B (3 downto 0);

Just an index of a higher level array and a slice of a lower level array
can be specified:
TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1);
TAB_D (0) (4) (2 downto 0) <= CST_A (5 downto 3)

Indices may be variable.
XST supports record types.

Example:

```vhdl
type REC1 is record
  field1: std_logic;
  field2: std_logic_vector (3 downto 0)
end REC1;
```

Properties:

- Record types can contain other record types.
- Constants cannot be record types.
- Record types cannot contain attributes.
- XST does not support aggregate assignments to record signals.
Entity Declaration

The I/O ports of the circuit are declared in the entity. Each port has a name, a mode (in, out, inout or buffer) and a type.

Types of ports must be constrained, and not more than one dimensional array types are accepted as ports.
Component Configuration

XST supports component configurations in the declarative part of the architecture:

```for instantiation_list: component_name use
   LibName.entity_Name(Architecture_Name);
```

Example:

```for all : NAND2 use entity work.NAND2(ARCHI);
```

All NAND2 components will use the entity NAND2 and architecture ARCHI.

- No configuration: → XST links the component to the entity with the same name (& same interface) & the selected architecture to the most recently compiled architecture.
- If no entity/architecture is found, a black box is generated during the synthesis.
## VHDL Language support

<table>
<thead>
<tr>
<th>Section</th>
<th>VHDL Constructs Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity Header</strong></td>
<td></td>
</tr>
<tr>
<td>Generics</td>
<td>Supported</td>
</tr>
<tr>
<td>Ports</td>
<td>Supported (no unconstrained ports)</td>
</tr>
<tr>
<td>Entity Declarative Part</td>
<td>Supported</td>
</tr>
<tr>
<td>Entity Statement Part</td>
<td>Unsupported</td>
</tr>
<tr>
<td><strong>Architecture Bodies</strong></td>
<td></td>
</tr>
<tr>
<td>Architecture Declarative Part</td>
<td>Supported</td>
</tr>
<tr>
<td>Architecture Statement Part</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Configuration Declarations</strong></td>
<td></td>
</tr>
<tr>
<td>Block Configuration</td>
<td>supported</td>
</tr>
<tr>
<td>Component Configuration</td>
<td>supported</td>
</tr>
<tr>
<td><strong>Subprograms</strong></td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td>Supported</td>
</tr>
<tr>
<td>Procedures</td>
<td>Supported</td>
</tr>
</tbody>
</table>
## Packages

<table>
<thead>
<tr>
<th>Packages</th>
<th>Type TIME</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD</td>
<td>unsupported</td>
<td></td>
</tr>
<tr>
<td>TEXTIO</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>STD_LOGIC_1164</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>STD_LOGIC_ARITH</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>STD_LOGIC_SIGNED</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>STD_LOGIC_UNSIGNED</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>STD_LOGIC_MISC</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>NUMERIC_BIT</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>NUMERIC_EXTRA</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>NUMERIC_SIGNED</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>NUMERIC_UNSIGNED</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>NUMERIC_STD</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>ASYL.ARITH</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>ASYL.SL_ARITH</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>ASYL.PKG_RTL</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>ASYL.ASYL_1164</td>
<td>Supported</td>
<td></td>
</tr>
</tbody>
</table>
Supported types

<table>
<thead>
<tr>
<th>Composite</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_VECTOR</td>
<td>Supported</td>
</tr>
<tr>
<td>STD_ULOGIC_VECTOR</td>
<td>Supported</td>
</tr>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>Supported</td>
</tr>
<tr>
<td>UNSIGNED</td>
<td>Supported</td>
</tr>
<tr>
<td>SIGNED</td>
<td>Supported</td>
</tr>
<tr>
<td>Record</td>
<td>Supported</td>
</tr>
<tr>
<td>Access</td>
<td>supported 1</td>
</tr>
<tr>
<td>File</td>
<td>supported 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enumeration Types</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN, BIT</td>
<td>Supported</td>
</tr>
<tr>
<td>STD_ULOGIC, STD_LOGIC</td>
<td>Supported</td>
</tr>
<tr>
<td>XO1, UX01, XO1Z, UX01Z</td>
<td>Supported</td>
</tr>
<tr>
<td>Character</td>
<td>Supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Integer Types</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>Supported</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>Supported</td>
</tr>
<tr>
<td>NATURAL</td>
<td>Supported</td>
</tr>
<tr>
<td>Physical</td>
<td>Time ignored, real supported for constant calculations</td>
</tr>
<tr>
<td>Floating</td>
<td></td>
</tr>
</tbody>
</table>
Modes and declarations

Table 6-2 Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>In, Out, Inout</td>
<td>Supported</td>
</tr>
<tr>
<td>Buffer</td>
<td>Supported</td>
</tr>
<tr>
<td>Linkage</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

Table 6-3 Declarations

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported for enumerated types, types with positive range having constant bounds, bit vector types, and multi-dimensional arrays</td>
<td></td>
</tr>
<tr>
<td>Subtype</td>
<td>Supported</td>
</tr>
<tr>
<td>Objects</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>Constant Declaration</td>
<td>Supported (deferred constants are not supported)</td>
</tr>
<tr>
<td>Signal Declaration</td>
<td>Supported (“register” or “bus” type signals are not supported)</td>
</tr>
<tr>
<td>Variable Declaration</td>
<td>supported</td>
</tr>
<tr>
<td>File Declaration</td>
<td>supported</td>
</tr>
<tr>
<td>Alias Declaration</td>
<td>Supported</td>
</tr>
<tr>
<td>Attribute Declaration</td>
<td>Supported for some attributes, otherwise skipped (see the “Design Constraints” chapter)</td>
</tr>
<tr>
<td>Component Declaration</td>
<td>Supported</td>
</tr>
</tbody>
</table>
## Specifications and names

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Only supported for some predefined attributes: HIGH, LOW, LEFT, RIGHT, RANGE, REVERSE_RANGE, LENGTH, POS, ASCENDING, EVENT, STABLE, LAST_VALUE, DRIVING_VALUE. Otherwise, ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>Supported only with the “all” clause for instances list. If no clause is added, XST looks for the entity/architecture compiled in the default library.</td>
</tr>
<tr>
<td>Disconnection</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Simple Names</td>
<td>Supported</td>
</tr>
<tr>
<td>Selected Names</td>
<td>Supported</td>
</tr>
<tr>
<td>Indexed Names</td>
<td>Supported</td>
</tr>
<tr>
<td>Slice Names</td>
<td>supported</td>
</tr>
</tbody>
</table>
## Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Logical Operators: and, or, nand, nor, xor, xnor, not</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relational Operators: =, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>&amp; (concatenation)</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Adding Operators: +, -</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>/, mod, rem</td>
<td>Supported if the right operand is a constant power of 2</td>
</tr>
<tr>
<td></td>
<td>Shift Operators: sll, srl, sla, sra, rol, ror</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>abs</td>
<td>Supported for constant operands</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>Only supported if the left operand is 2</td>
</tr>
<tr>
<td></td>
<td>Sign: +, -</td>
<td>Supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operands</th>
<th>Abstract Literals</th>
<th>Only integer literals are supported</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Physical Literals</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>Enumeration Literals</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>String Literals</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Bit String Literals</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Record Aggregates</td>
<td>Unsupported</td>
</tr>
<tr>
<td></td>
<td>Array Aggregates</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Function Call</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Qualified Expressions</td>
<td>Supported for accepted predefined attributes</td>
</tr>
<tr>
<td></td>
<td>Types Conversions</td>
<td>supported</td>
</tr>
<tr>
<td></td>
<td>Allocators</td>
<td>Unsupported</td>
</tr>
<tr>
<td></td>
<td>Static Expressions</td>
<td>Supported</td>
</tr>
</tbody>
</table>
## Wait and loop statements

<table>
<thead>
<tr>
<th>Wait Statement</th>
<th>Wait on \textit{sensitivity_list} until \textit{Boolean_expression}. See the &quot;Sequential Circuits&quot; section for details.</th>
<th>Supported with one signal in the sensitivity list and in the Boolean expression. In case of multiple wait statements, the sensitivity list and the Boolean expression must be the same for each wait statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wait for \textit{time_expression}. See the &quot;Sequential Circuits&quot; section for details.</td>
<td>Unsupported</td>
</tr>
<tr>
<td></td>
<td>Assertion Statement</td>
<td>supported, for static conditions</td>
</tr>
<tr>
<td></td>
<td>Signal Assignment Statement</td>
<td>Supported (delay is ignored)</td>
</tr>
<tr>
<td></td>
<td>Variable Assignment Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Procedure Call Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>If Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Case Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Loop Statement</td>
<td>&quot;for ... loop ... end ... loop&quot;</td>
<td>Supported for constant bounds only</td>
</tr>
<tr>
<td></td>
<td>&quot;while ... loop ... end loop&quot;</td>
<td>supported</td>
</tr>
<tr>
<td></td>
<td>&quot;loop ... end loop&quot;</td>
<td>Only supported in the particular case of multiple wait statements</td>
</tr>
<tr>
<td></td>
<td>Next Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Exit Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Return Statement</td>
<td>supported</td>
</tr>
<tr>
<td></td>
<td>Null Statement</td>
<td>Supported</td>
</tr>
</tbody>
</table>
# Concurrent Statements

<table>
<thead>
<tr>
<th>Concurrent Statement</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Concurrent Procedure Call</td>
<td>Supported</td>
</tr>
<tr>
<td>Concurrent Assertion Statement</td>
<td>Ignored</td>
</tr>
<tr>
<td>Concurrent Signal Assignment Statement</td>
<td>Supported (no “after” clause, no “transport” or “guarded” options, no waveforms)</td>
</tr>
<tr>
<td>Component Instantiation Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>“For ... Generate”</td>
<td>Statement supported for constant bounds only</td>
</tr>
<tr>
<td>“If ... Generate”</td>
<td>Statement supported for static condition only</td>
</tr>
</tbody>
</table>
Summary

Synthesis subset of VHDL: Demonstrated for XST (similar restrictions for other synthesis systems)

- after clause ignored.
- Distinction between combinatorial and sequential models
  - Combinatorial models contain assignments for all execution paths
- XST adds signals to sensivity list.
- Inference of registers
- Restricted support of functions
- Templates for state machines
- No unconstrained types except for functions
- Simplified component configuration
- Subset of types supported