technische universität
dortmund

fakultät für informatik
informatik 12

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=(components, nets) and components={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

- VHDL IEEE std 1076-1987 constructs are accepted if they do not conflict with VHDL IEEE std 1076-1993. In case of a conflict, VHDL IEEE Std 1076-1993 behavior overrides VHDL IEEE std 1076-1987.
- 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 <= a **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:

```
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
  ($\forall$ assigned signals $\in$ process, $\forall$ paths:
  $\exists$ 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:

```
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:

```
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
($\exists$ signal assigned somewhere, $\exists$path:
$\neg\,\exists$ assignment to this signal on this path)

$\rightarrow$ 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

```
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

```
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

Description of counters

```
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:

  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.

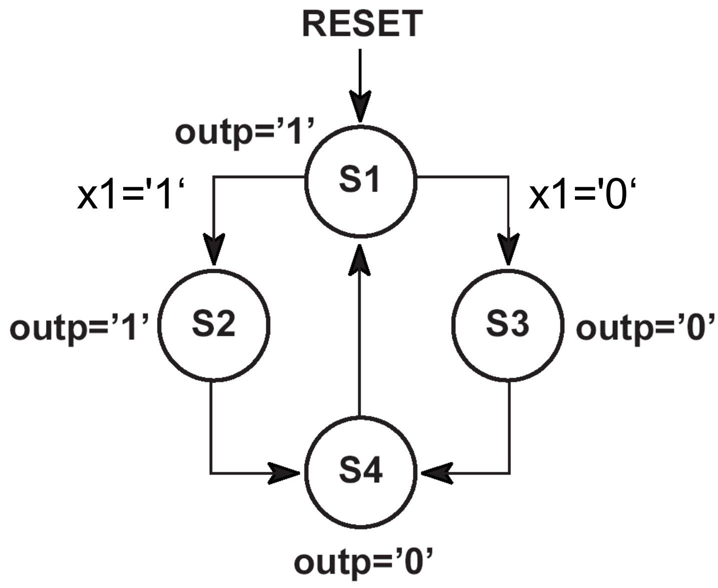
- Parameters can be unconstrained. The content is similar to the combinatorial process content.
- Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package.
- Recursive function/procedure calls are not supported.

```
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.

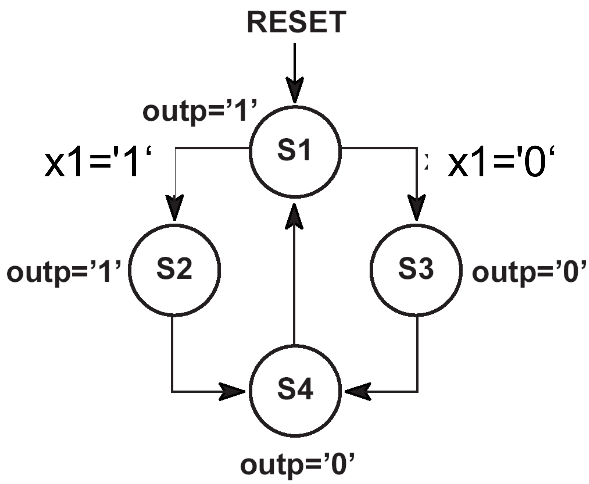
State Machines

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:



FSM with 1 Process

```
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*.

FSM with 2 Processes

```
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;
```

FSM with 3 Processes

```
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, next_state: state_type ;
begin
 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;
```

```
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 process2;
```

```
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;
```

Coding styles for FSMs

State Registers

- State Registers must to 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:

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 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*, π, 2 π, 1/π, … 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:

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

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 <= CNST_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) <= CNST_A (5 **downto** 3)

Indices may be variable.

Record Types

XST supports record types.

Example:

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);

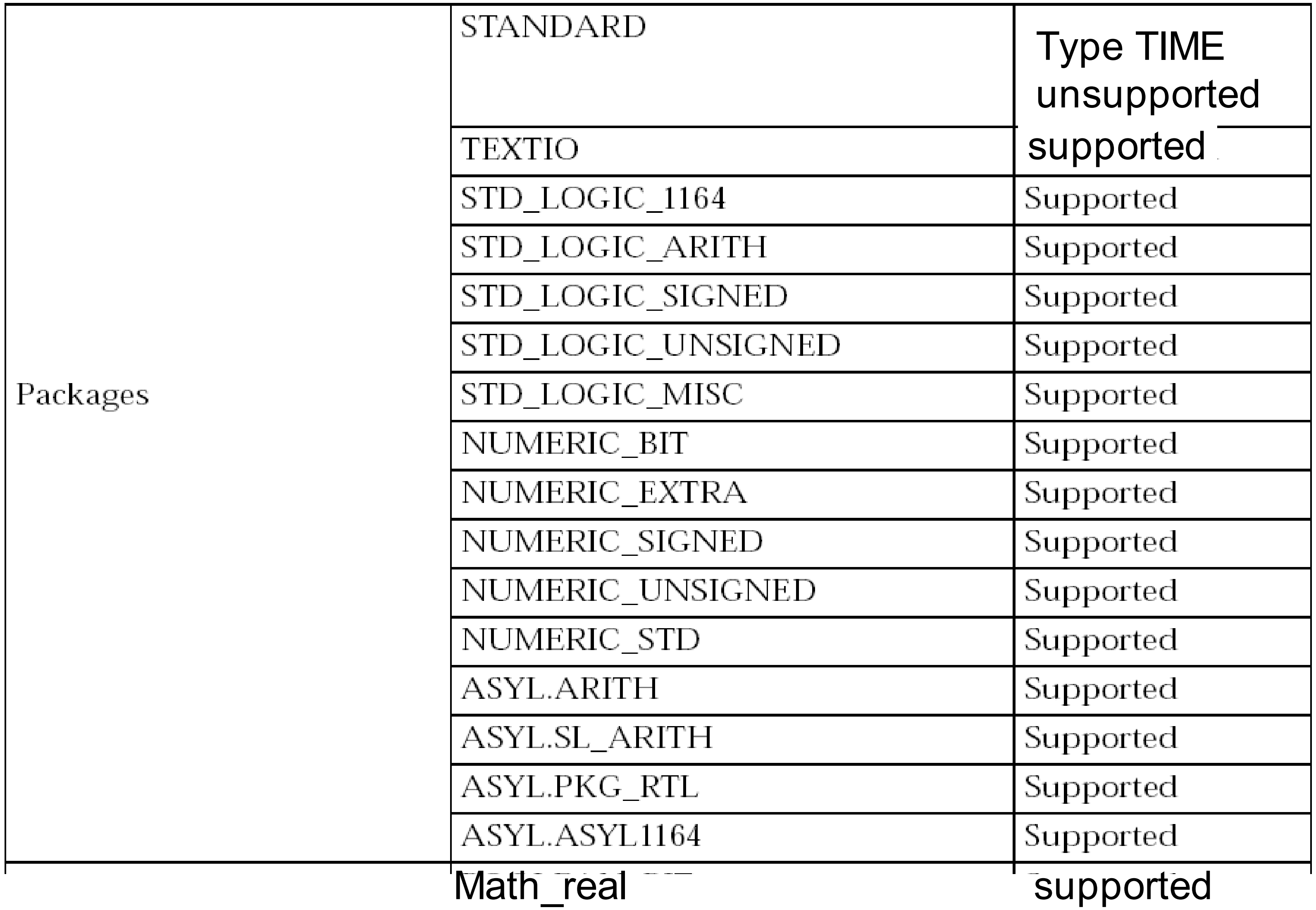
All NAND2 components will use the entity NAND2 and architecture ARCHI.

- 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

Entity Header	Generics	Supported
	Ports	Supported (no unconstrained ports)
	Entity Declarative Part	Supported
	Entity Statement Part	Unsupported
Architecture Bodies	Architecture Declarative Part	Supported
	Architecture Statement Part	Supported
Configuration Declarations	Block Configuration	supported
	Component Configuration	supported
Subprograms	Functions	Supported
	Procedures	Supported

Packages



Packages	STANDARD	Type TIME unsupported
	TEXTIO	supported
	STD_LOGIC_1164	Supported
	STD_LOGIC_ARITH	Supported
	STD_LOGIC_SIGNED	Supported
	STD_LOGIC_UNSIGNED	Supported
	STD_LOGIC_MISC	Supported
	NUMERIC_BIT	Supported
	NUMERIC_EXTRA	Supported
	NUMERIC_SIGNED	Supported
	NUMERIC_UNSIGNED	Supported
	NUMERIC_STD	Supported
	ASYL.ARITH	Supported
	ASYL.SL_ARITH	Supported
	ASYL.PKG_RTL	Supported
	ASYL.ASYL1164	Supported
	Math_real	supported

Supported types

Composite	BIT_VECTOR	Supported
	STD_ULOGIC_VECTOR	Supported
	STD_LOGIC_VECTOR	Supported
	UNSIGNED	Supported
	SIGNED	Supported
	Record	Supported
	Access	supported i
	File	supported i
Enumeration Types	BOOLEAN, BIT	Supported
	STD_ULOGIC, STD_LOGIC	Supported
	XO1, UX01, XO1Z, UX01Z	Supported
	Character	Supported
Integer Types	INTEGER	Supported
	POSITIVE	Supported
	NATURAL	Supported
	Physical	Time ignored, real supported for constant calculations
	Floating	

Modes and declarations

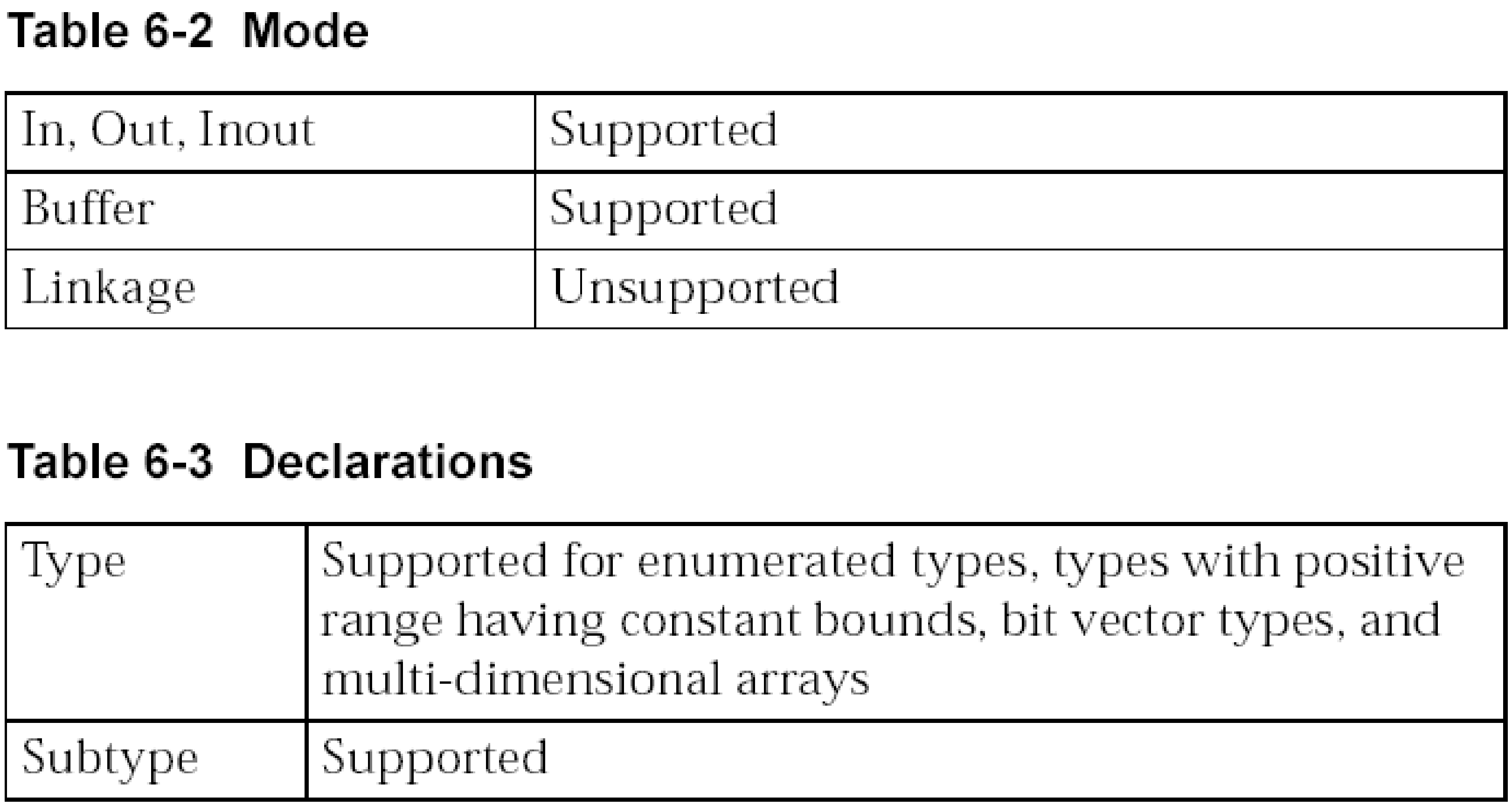


Table 6-2 Mode

In, Out, Inout	Supported
Buffer	Supported
Linkage	Unsupported

Table 6-3 Declarations

Type	Supported for enumerated types, types with positive range having constant bounds, bit vector types, and multi-dimensional arrays
Subtype	Supported

Objects

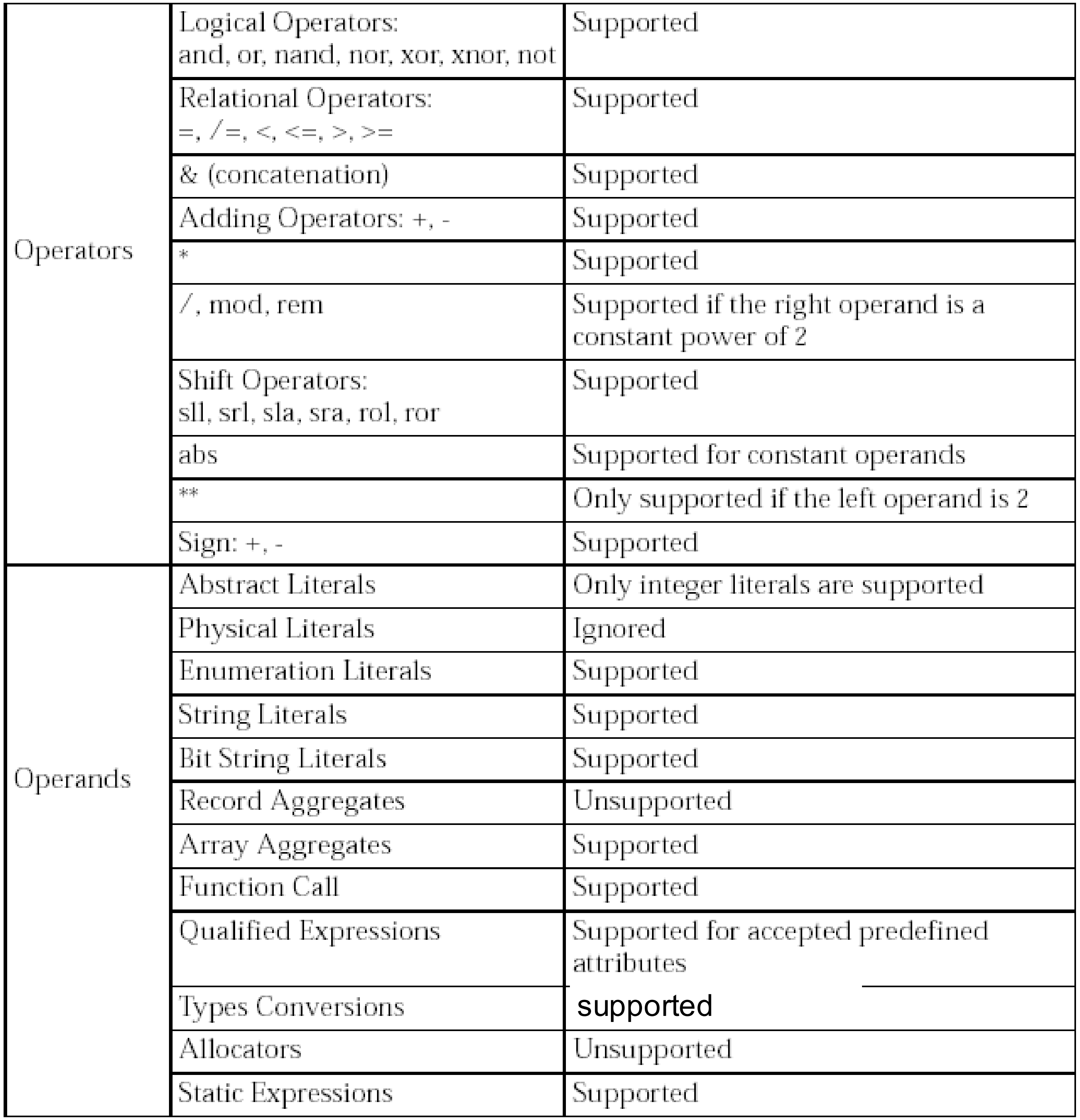
Constant Declaration	Supported (deferred constants are not supported)
Signal Declaration	Supported (“ register” or “ bus” type signals are not supported)
Variable Declaration	supported
File Declaration	supported
Alias Declaration	Supported
Attribute Declaration	Supported for some attributes, otherwise skipped (see the “ Design Constraints” chapter)
Component Declaration	Supported

Specifications and names

Attribute	Only supported for some predefined attributes: HIGH, LOW, LEFT, RIGHT, RANGE, REVERSE_RANGE, LENGTH, POS, ASCENDING, EVENT, STABLE, LAST_VALUE, DRIVING_VALUE. Otherwise, ignored.
Configuration	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.
Disconnection	Unsupported

Simple Names	Supported
Selected Names	Supported
Indexed Names	Supported
Slice Names	supported

Operators

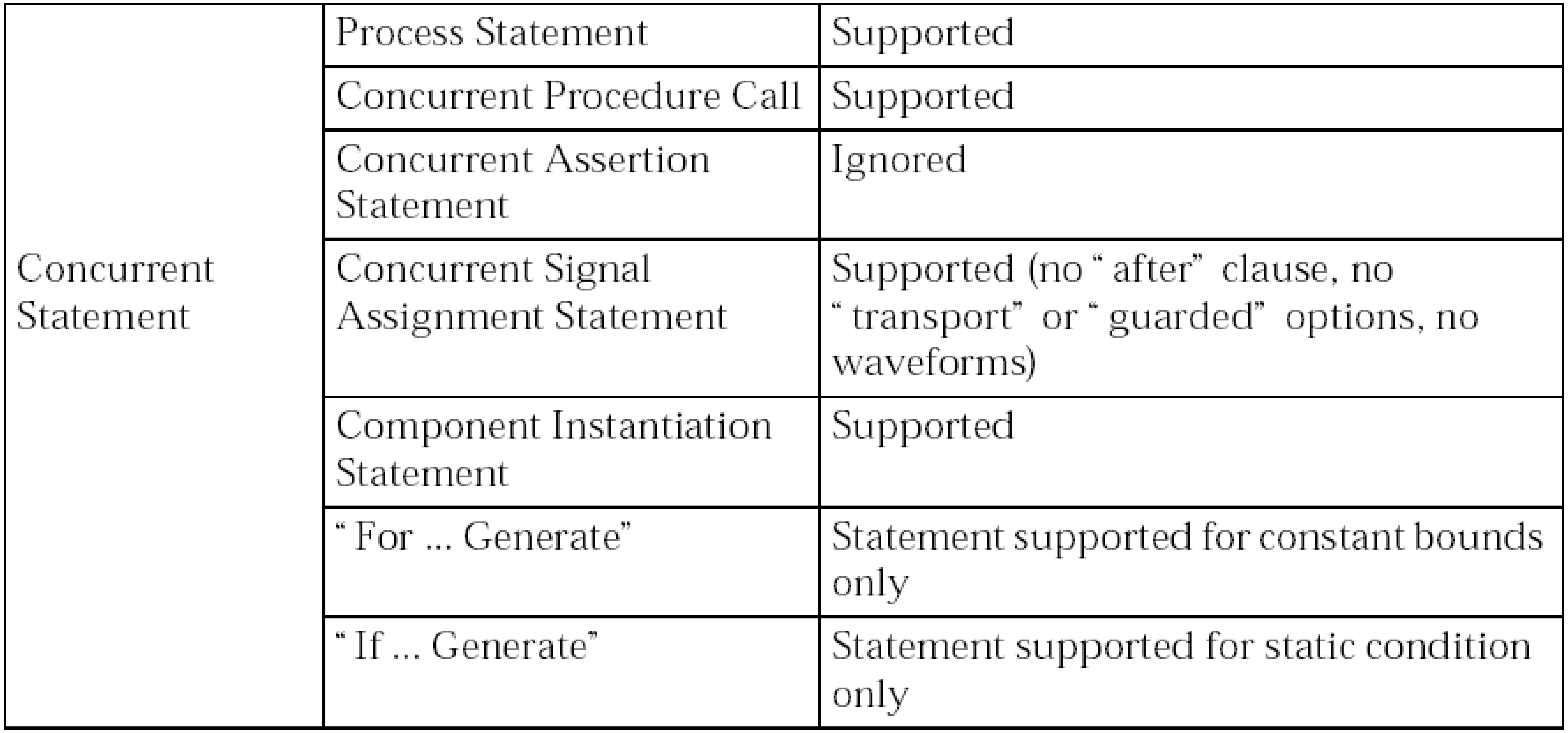


Operators	Logical Operators: and, or, nand, nor, xor, xnor, not	Supported
	Relational Operators: =, /=, <, <=, >, >=	Supported
	& (concatenation)	Supported
	Adding Operators: +, -	Supported
	*	Supported
	/, mod, rem	Supported if the right operand is a constant power of 2
	Shift Operators: sll, srl, sla, sra, rol, ror	Supported
	abs	Supported for constant operands
	**	Only supported if the left operand is 2
	Sign: +, -	Supported
Operands	Abstract Literals	Only integer literals are supported
	Physical Literals	Ignored
	Enumeration Literals	Supported
	String Literals	Supported
	Bit String Literals	Supported
	Record Aggregates	Unsupported
	Array Aggregates	Supported
	Function Call	Supported
	Qualified Expressions	Supported for accepted predefined attributes
	Types Conversions	supported
	Allocators	Unsupported
	Static Expressions	Supported

Wait and loop state-ments

Wait Statement	Wait on *sensitivity_list* until *Boolean_expression*. See the " Sequential Circuits" section for details.	Supported with one signal in the sensi-tivity 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.
	Wait for *time_expression*. . See the " Sequential Circuits" section for details.	Unsupported
	Assertion Statement	supported, for static conditions
	Signal Assignment Statement	Supported (delay is ignored)
	Variable Assignment Statement	Supported
	Procedure Call Statement	Supported
	If Statement	Supported
	Case Statement	Supported
Loop Statement	" for ... loop ... end ... loop"	Supported for constant bounds only
	" while ... loop ... end loop"	supported
	" loop ... end loop"	Only supported in the particular case of multiple wait statements
	Next Statement	Supported
	Exit Statement	Supported
	Return Statement	supported
	Null Statement	Supported

Concurrent Statements



Concurrent Statement	Process Statement	Supported
	Concurrent Procedure Call	Supported
	Concurrent Assertion Statement	Ignored
	Concurrent Signal Assignment Statement	Supported (no “after” clause, no “transport” or “guarded” options, no waveforms)
	Component Instantiation Statement	Supported
	“For ... Generate”	Statement supported for constant bounds only
	“If ... Generate”	Statement supported for static condition only

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