TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of VLSI Design, Diagnostics & Architecture

VHDL – REUSE

A Language for Modeling and Designing Hardware

Dr. Thomas B. Preußer

thomas.preusser@tu-dresden.de

Dresden, July 2, 2014

Goals of this Lecture

- Cover the different aspects of reuse in VHDL:
  - structural reuse by component instantiation,
  - functional reuse through functions, and
  - behavioral reuse through procedures.
- How to enhance reusability by means of generics.

Reusing a Module: Running Example

- Assume the following module:

```
entity parity is
  port(
    arg : in  std_logic_vector(1 to 8);
    par : out std_logic
  );
end parity;

architecture rtl of parity is
begin
  par <= arg(1) xor arg(2) xor arg(3) xor arg(4) xor
         arg(5) xor arg(6) xor arg(7) xor arg(8);
end rtl;
```

- Note the ports establishing the external signal interface!
- How can you integrate it in another module implementation?

Reusing a Module: Instantiation

```
architecture rtl of xyz is

  -- Component declaration - analogous to the declaration of the entity
  component parity is
    port(
      arg : in  std_logic_vector(1 to 8);
      par : out std_logic
    );
  end component;

  -- Connectivity
  signal arg1, arg2 : std_logic_vector(7 downto 0);
  signal par1, par2 : std_logic;
  ...
begin
  -- Instantiation
  p1 : parity     -- Named association of signals (RECOMMENDED)
    port map(
      arg => arg1,  -- note: only array lengths must match
      par => par1
    );
  p2 : parity     -- Positional association of signals
    port map(arg2, par2);
  ...
end rtl;
```

Generics: Enhancing Flexibility

- Reuse potential increases through generic parameters, e.g.:

```
entity parity is
  generic(
    N : positive          -- Bit Width of Input
  );
  port(
    arg : in  std_logic_vector(1 to N);
    par : out std_logic
  );
end parity;
```

- The module is available for an *arbitrary* bit width.
- Each concrete instance has *constant* parameters, which are bound at its corresponding instantiation site:

```
p1 : parity
  generic map(          -- Binding of generics
    N   => 8
  )
  port map(
    arg => arg1,
    par => par1
  );
```

Implementing with Generics

Generics require an adaptive module implementation:

```
architecture rtl of parity is
  signal tmp : std_logic_vector(1 to N);
begin
  tmp(1) <= arg(1);
  xor_chain: for i in 2 to N generate
    tmp(i) <= tmp(i-1) xor arg(i);
  end generate;
  par <= tmp(N);
end rtl;
```

- Generic parameters can be used just like constants.
- **generate** statements:
  - provide adaptive structure
  - expanding a *concurrent* context.

The generate-statement

. . . enables the specification of conditional or repetitive structures:

```
<Label>: if <Condition> generate
[ [local declarations]
begin]
   ...
end generate [Label];
```

```
<Label>: for <Identifier> in <range> generate
[ [local declarations]
begin]
   ...
end generate [Label];
```

Array and Type Attributes

. . . help to keep code generic:

Example: **signal** arr : std_logic_vector(7 **downto** 0);

arr'length	Array length	8
arr'**range**	Index range	7 **downto** 0
arr'reverse_range	Reverse index range	0 **to** 7
arr'left	Left boundary of index range	7
arr'right	Right boundary of index range	0
arr'high	Highest array index	7
arr'low	Lowest array index	0

- Instead of variable names, also type names may be used.
- The desired dimension of multidimensional array types has to be selected by argument:
  arr'length(1) and arr'length are equivalent.

Functional Reuse

Functions:

- are sequential specifications of *instant* computations,
- do (ideally) not have side effects, and
- cannot modify their arguments.

The specification of the computation may use local **variable**s with *instant* assignment semantics using the operator := (unlike **signal**s!).

```
function log2ceil(arg : positive) return natural is
  variable tmp : positive;
  variable log : natural;
begin
  tmp := 1;
  log := 0;
  while arg > tmp loop
    tmp := tmp * 2;
    log := log + 1;
  end loop;
  return log;
end log2ceil;
```

Behavioral Reuse

Procedures:

- encapsulate timed sequential behavior,
- may have **out** and **inout** parameters, which are **variable**s or **signal**s, and
- capture declarations from their enclosing scopes.

```
procedure cycle is
begin
  clk <= '0';      -- signal captured from surrounding scope
  wait for 5 ns;
  clk <= '1';
  wait for 5 ns;
end cycle;
```

Procedures are mostly used in behavioral models not destined for synthesis.

Variables

- may be declared in any *sequential* context, i.e. in a **process**, a **function**, or a **procedure**.
- are updated through *instant* assignments: :=

When code is synthesized, a **variable** may unfold to a single or to multiple physical nodes, e.g.:

```
process(arg)
  variable tmp : std_logic;
begin
  tmp := '0';
  for i in arg'range loop
    tmp := tmp xor arg(i);
  end loop;
  par <= tmp;     -- final signal assignment
end process;
```

Variables are often considered internal implementation details and cannot be tapped for visualization in waveforms.

Parameter Passing

Parameters to **procedure**s and **function**s are passed in different modes:

Mode	Class	Parameter	
		Procedure	**Function**
in[1]	**constant**[2]	Expression	Expression
	signal	Signal	Signal
	variable	Variable	⊘
out / **inout**	**signal**	Signal	⊘
	variable[2]	Variable	⊘

[1] Default
[2] Default for Mode

Packages

. . . are useful for the collection of:

- non-local types and constants,
- component declarations,
- functions and procedures.

Separation into interface and implementation:

```
package utils is
  subtype tALUOp is std_logic_vector(1 downto 0);
  constant ALU_OP_ADD : tALUOp := "00";
  ...
  function max(arg1 : integer; arg2 : integer) return integer;
end package utils;

package body utils is
  function max(arg1 : integer; arg2 : integer) return integer is
  begin
    if arg1>arg2 then return arg1; end if;
    return arg2;
  end;
end utils;
```

Assertions

- are used for checking
  - static assumptions, e.g. about generics, or
  - dynamic assumptions during the simulation.
- are not synthesized into logic.

```
assert <Condition>                                  — Assertion
  [report   <String>]                               — Message
  [severity (note|warning|error|failure)];          — Severity
```

- If the assertion does not apply:
  - the message is output if it is provided, and
  - the simulation aborts depending on the declared severity.

Assertion Example

```
architecture tb of parity_tb is
  signal arg : std_logic_vector(1 to 4);
  signal par : std_logic;
begin
  par : parity
    generic map(N => arg'length)
    port    map(arg => arg, par => par);

  process
  begin
    for i in 0 to 2**arg'length-1 loop
      arg <= std_logic_vector(to_unsigned(i, arg'length));
      wait for 10 ns;
      assert par = (arg(1) xor arg(2) xor arg(3) xor arg(4))
        report "Parity Mismatch"
        severity error;
    end loop;
    report "Test complete";
    wait; -- forever
  end process;
end tb;
```

This is a *testbench*!