Lecture 9

VHDL, part IV

Hierarchical and parameterized design

Section 1

HIERARCHICAL DESIGN
Dealing with Large Digital System Design

1. Apply hierarchy to the design
   - At the highest level use larger functional blocks that consist of one or more hierarchical blocks = increase abstraction
   - At the lowest level: combinational and sequential networks

2. Apply Register Transfer Level (RTL) description of the functionality
   - Helps to increase abstraction

3. Partition the system according to the nature of performed operations (logical partition)
   - Often used highest level partition: Data and Control

Physical Partition

Division of the physical implementation

Each subsystem is synthesized independently
   - Some circuit should be isolated as independent modules
     - Device-dependent circuit: e.g., memory modules
     - “Non-Boolean” circuit: tri-state buffer, delay-sensitive circuit, clock distribution network, synchronization circuit.
Components

Hierarchical design usually shown as a block diagram (structural description)

VHDL component is the mechanism to describe structural description in text

To use a component
  – Component declaration (make known)
  – Component instantiation (create an instance)

Component declaration

In the declaration section of architecture

Info similar to entity declaration

Syntax:
```vhdl
component component_name
  generic (generic_declaration;
             generic_declaration;
             ...
  );
  port (port_declaration;
        port_declaration;
        ...
  );
end component;
```
Example

A decade (mod-10) counter

entity dec_counter is
  port(
    clk, reset: in std_logic;
    en: in std_logic;
    q: out std_logic_vector(3 downto 0);
    pulse: out std_logic
  );
end dec_counter;

architecture up.arch of dec_counter is
  signal r_reg: unsigned(3 downto 0);
  constant TEB: integer := 10;
begin
  register
  process(clk, reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_reg + 1;
    end if;
  end process;
  -- next-state logic
  process(r_reg)
  begin
    r_next <= r_reg;
    if (en='1') then
      if r_reg(3)'true' then
        r_next <= (others=>'0');
      else
        r_next <= r_reg + 1;
      end if;
    end if;
  end process;
  -- output logic
  process(r_reg)
  begin
    q <= std_logic_vector(r_reg);
    pulse <= '1' when r_reg(TEB) else '0';
  end process;
end up.arch;

Component declaration

Component declaration for dec_counter

component dec_counter
  port(
    clk, reset: in std_logic;
    en: in std_logic;
    q: out std_logic_vector(3 downto 0);
    pulse: out std_logic
  );
end component;
Component instantiation

Instantiate an instance of a component
Provide a generic value
Map formal signals to actual signals

```
instance_label: component_name
  generic map(
    generic_association;
    generic_association;
    ...
  )
  port map(
    port_association;
    port_association;
    ...
  );
```

Port association: `port_name => signal_name`

Example

E.g., 2-digit decimal counter
(00=>01=> . . . =>99 =>00 . . .)
library ieee;
use ieee.std_logic_1164.all;

entity hundred_counter is
  port(
    clk, reset: in std_logic;
    en: in std_logic;
    q_ten, q_one: out std_logic_vector(3 downto 0);
    p100: out std_logic
  );
end hundred_counter;

architecture vhdl_87_arch of hundred_counter is
  component dec_counter
    port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
    );
  end component;
  signal p_one, p_ten: std_logic;
begin
  one_digit: dec_counter
  port map (clk=>clk, reset=>reset, en=>en,
            pulse=>p_one, q=>q_one);
  ten_digit: dec_counter
  port map (clk=>clk, reset=>reset, en=>p_one,
            pulse=>p_ten, q=>q_ten);
  p100 <= p_one and p_ten;
end vhdl_87_arch;

Port name association

The VHDL code is a textual description of a schematic
Positional association

Appeared to be less cumbersome
- E.g., order of port declaration in entity:

```
clk, reset, en, q, pulse
```
- Alternative component instantiation

```
one_digit: dec_counter
  port map (clk, reset, en, q_one, p_one);
ten_digit: dec_counter
  port map (clk, reset, p_one, q_ten, p_ten);
```
- Trouble if the order later changes in entity declaration

Mapping of constant and unused port

Good synthesis software should
- remove the unneeded part
- perform optimization over the constant input

```
one_digit: dec_counter
  port map (clk=>clk, reset=>reset, en=>'1',
            pulse=>p_one, q=>q_one);
ten_digit: dec_counter
  port map (clk=>clk, reset=>reset, en=>p_one,
            pulse=>open, q=>q_ten);
```
Configuration

Bind a component with an entity and an architecture

Type of configuration:
  – Configuration declaration (an independent design unit)
  – Configuration specification (in architecture body)

Flexible and involved

Not supported by all synthesis software

Default binding: (no configuration)
  – Component bound to an entity with identical name
  – Component ports bound to entity ports of same names
  – Most recently analyzed architecture body bound to the entity

Library

A virtual repository to stored analyzed design units

Physical location determined by software

Design units can be organized and stored in different libraries

Declaration

```plaintext
library lib_name, lib_name, ... , lib_name;
```

– E.g. library IEEE;
Package

Organize and store declaration information, such as data types, functions etc.
Divided into
- Package declaration
- Package body (implementation of subprograms)
Both are design units

```pascal
package package_name is
    declaration item;
    declaration item;
    ...
end package_name;

package body package_name is
    subprogram;
    subprogram;
    ...
end package_name;
```

Section 2

PARAMETERIZED DESIGN
Introduction

Why parameterized design?

– Design reuse: the same design block can be used in different situation
– Possibility of reuse is higher if the block is “customizable”
  • i.e. different data widths or features
– With parameterized design the characteristics of a block can be customized at design-time

Type of parameters

Width parameters
– Specify the size of relevant data signals
– This enables scalable design

Feature Parameters
– Specify small variation inside a design
  • i.e. include or not an output buffer
– In theory can be used to select totally different implementations
  • Instead of using different architectures
  • There is no specific rules about this, but as general rule parameters are supported by any design tool
Methods for parameter definition

There are two methods to enable parameterized design
- Generics
- Unconstrained arrays

The parameter can be directly called when using generics, while should be referred to array attributes when using unconstrained arrays.

Generics

Mechanism to pass info into an entity/component

Declared in entity declaration and then a constant can be assigned to it in port declaration and architecture body.

Assigned a value when the component is instantiated.

Like a parameter, but has to be constant.
8-bit Reduced-xor circuit

\[ a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \]

library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor is
  port (a: in std_logic_vector(7 downto 0); y: out std_logic);
end reduced_xor;

architecture cascade1_arch of reduced_xor is
begin
  y <= a(0) xor a(1) xor a(2) xor a(3) xor a(4) xor a(5) xor a(6) xor a(7);
end cascade1_arch;

Block Diagram

Diagram of the 8-bit Reduced-xor circuit showing the logic gates and connections.
Example: Reduced XOR with generics

Listing 14.1 Parameterized reduced-xor circuit using a generic

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity reduced_xor is
  generic(WIDTH: natural); -- generic declaration
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    y: out std_logic
  );
end reduced_xor;

architecture loop_linear_arch of reduced_xor is
  signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  process(a,tmp)
  begin
    tmp(0) <= a(0); -- boundary bit
    for i in 1 to (WIDTH-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
  end process;
  y <= tmp(WIDTH-1);
end loop_linear_arch;
```

Unconstrained Arrays

Unconstrained array has no defined bounds or range, but only the index type is specified
- Type unc_array is array(natural range<>) of integer;

In some part of the project the array size MUST be defined

The parameters are taken directly from the array

Unconstrained arrays are more flexible and compact

Generics need the designer to specify every parameter
- Cumbersome but safe
- Generics are preferable
Fixed-size references

One key to develop parameterized design is to avoid fixed-size references

All the array ranges should be linked to one/more parameters
- i.e. signal s = std_logic_vector(WIDTH – 1 downto 0)

Some specific techniques should be used for signal initialization/assignment
- This kind of assignment must be avoided in parameterized design
  - q<= “00000000”
- Because it doesn’t accomodate any array size

Fixed value assignments

Alternative to “direct” assignments are several

Named associations
- q <= (others => ‘0’);
- q <= (others => ‘1’);
- q <= (0=>'1', others => ‘0’);

Integer conversions
- q<= to_unsigned(6, WIDTH)
- If q std_logic_vector: q<= std_logic_vector(to_unsigned(6, WIDTH))
Section 3

GENERATE STATEMENTS

Generate Statements

Concurrent statements which can be considered as circuit parts

These circuit parts can be instantiated iteratively (for generate)

These circuit part can be instantiated according to the value of some parameters (if generate)
For Generate

Iterative instantiation of circuit parts

Syntax:

```
Gen_label:
    for loop_index in loop_range generate
        Concurrent statements;
    end generate;
```

It repeats the loop body of concurrent statements for a definite number of times

The range has to be static (defined at compilation time)

- It is normally related to some width parameters

The body contains concurrent statements (can be other generates)

---

Example: reduced XOR circuit

Listing 14.12  Parameterized reduced-xor circuit using a for generate statement

```vhdl
architecture gen_linear_arch of reduced_xor is
    signal tmp : std_logic_vector(WIDTH-1 downto 0);
begin
    tmp(0) <= a(0);
    xor_gen:
        for i in 1 to (WIDTH-1) generate
            tmp(i) <= a(i) xor tmp(i-1);
        end generate;
    y <= tmp(WIDTH-1);
end gen_linear_arch;
```
Conditional Generate Statement

Conditional instantiation of concurrent statements

Syntax:

\[
\text{Gen_label:}
\]
\[
\text{if condition generate}
\]
\[
\text{Concurrent statements;}
\]
\[
\text{end generate;}
\]

It instantiates the circuit described by the body of concurrent statements if the condition is true.
There is no else branch.

The condition must be static for synthesis (evaluable at compile time).

Example: reduced XOR

Listing 14.15 Parameterized reduced-xor circuit with a conditional generate statement

architecture gen_if_arch of reduced_xor is
signal tmp: std_logic_vector(WIDTH-2 downto 1);
begins
xor_gen:
for i in 1 to (WIDTH-1) generate
-- leftmost stage
left_gen: if i=1 generate
    tmp(i) <= a(i) xor a(0);
    end generate;
-- middle stages
middle_gen: if (i < 1) and (i < (WIDTH-1)) generate
    tmp(i) <= a(i) xor tmp(i-1);
    end generate;
-- rightmost stage
right_gen: if i=(WIDTH-1) generate
    y <= a(i) xor tmp(i-1);
    end generate;
end gen_if_arch;
Section 4

TWO-DIMENSIONAL ARRAYS

Motivation

Parameterization can lead to inefficiency
- Reduced language constructs => not enough flexibility for the designer
- Very high level construct => too far from the underlying hardware

Loops can describes one-dimensional cascading structures
- Long propagation delay, difficult for placement
- Tool cannot perform optimization at that level

To develop efficient design we should care more about the overall topology
- Tree instead of cascade
- In this lecture some examples…
Two-dimensional signals

To overcome the inefficiency of single loop constructs, advanced parameterized design focuses on regular two-dimensional structures

- Two-dimensional signals are required for such use

Two convenient requirements

- The two-dimensional signal must be based on std_logic, std_logic_vector, signed, unsigned types
- To improve portability and readability, an user-defined data type should be introduced

Three alternatives:

- Genuine two-dimensional data type
- Array-of-arrays
- Emulated two-dimensional array

Genuine two-dimensional data type

Collection of elements of the same type

```vhdl
type data_type_name is array (range_1, range_2, ...) of element_data_type;
```

Data type is a constrained array if the range is fixed, unconstrained array otherwise

- Constrained
  ```vhdl
type array32x8 is array (31 downto 0, 7 downto 0) of std_logic;
  ```
  - Range is fixed
  - An index pair (i, j) can be used to access one specific element

- Unconstrained
  ```vhdl
type std_logic_2d is array (natural <> , natural <> ) of std_logic;
  ```
  - The range is not defined in the type, but at the signal declaration
  - Sometimes not accepted by synthesis tools
Array-of-arrays data type

One dimensional array whose element are one-dimensional arrays

- The data type must be a constrained array

```vhdl
type array32x8 is array (31 downto 0) of std_logic_vector(7 downto 0);
```

Same assignment of the two-dimensional constrained array

Access of a row is much easier:

```vhdl
signal s : std_logic_vector(7 downto 0);
signal t : array32x8;
s <= t(7);
```

More synthesis software accept this form

2D Arrays: Requirements

Ideally the two-dimensional representation should be

- as close as possible to the underlying hardware
- accepted by any synthesis software

Both requirements cannot be easily achieved

However conversion between different representation is usually straightforward
Synthesis of Parameterized Modules

The parameter is set at design-time
   – The synthesizer performs the synthesis of a fixed-size module

Parameterization requires general and repetitive structures
   – More “preparation work” for the designer

Synthesis can be more demanding
   – Flattening the multi-dimensional arrays
   – Processing the static expression

A good synthesis software should be able to interpret the multi-dimensional arrays and make optimizations where needed

Section 5
MISTAKES ONE USUALLY DOES...
VHDL Pitfalls

1. Identifiers
   - VHDL isn’t case sensitive (e.g. Input and input are the same)

2. If statement
   - ELSIF written ELSE IF

3. Wrong string delimiters
   - ’0001’ instead of “0001”

4. Reserved words
   - reserved words used as object names: IN, OUT, BUFFER, AND, NAND, OR

5. Case statement
   - VHDL requires all conditions to be presented

VHDL Pitfalls (2)

6. Expression evaluation
   - following operations have equal precedence
     AND, OR, NAND, NOR, XOR
   - following expression is illegal
     a OR b AND c
   - correct expression
     a OR (b AND c)

7. Signal assignment from multiple sources
   - E.g. signal is reset in sequential process, but it is assigned outside the process
   - Only one driver for a signal!
     • A process, a concurrent statement …
Typical mistakes with syntax

Signal assignment is <=, variable assignment :=
- Depends on the object on the left hand side (signal <= variable, variable := signal)

Entity ports and generics
- Lines end with ";" except the last one is left without ";"

Component instantiation
- Assignments end with comma, except the last one is without comma

```
n2h2_rx_chan_1 : n2h2_rx_chan
  generic map (data_width_g => data_width_g,
               ...)
  port map (clk => clk,
            rst_n => rst_n,
            avalon_addr_in => mem_addr_r(i),
            ...
            irq_out => irq_chan_r(i));
```

Mistakes (2)

Using don’t care-operator:
- E.g. when "1--" => ...
- May simulate well (modelsim supports)
- Does not synthesize!

The process sensitivity list is incomplete!
- May hide bugs in the design
- Synthesis ignores sensitivity list but simulation relies on its completeness
Top 20 Errors in VHDL

This is a list of the most common VHDL errors. The Top 10 account for about 40% of all errors. The Top 20 account for about 60% of all errors.

Source: Doulous VHDL Golden reference guide

VHDL Pitfall: Null-Statement in CASE

-- Null statement in case statement
-- at combinatorial process
-- produces latch in synthesis.

ENTITY null_statement IS
  PORT (
    sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    output : OUT STD_LOGIC);
END null_statement;
ARCHITECTURE example OF null_statement IS
BEGIN -- Example
  PROCESS (sel)
  BEGIN -- PROCESS
    CASE sel IS
      WHEN "00" => output <= '0';
      WHEN "10" => output <= '1';
      WHEN OTHERS => NULL;
    END CASE;
  END PROCESS;
END example;
General guidelines and hints

Use only synthesizable code!
- E.g. "After x ns" is prohibited!
- If you fix your code with e.g. after, it won’t work after synthesis
- Only place to use non-synthesizable code is testbenches

Use std_logic data types and numeric_std package

Use only descending range in the arrays (e.g. downto)
- Signal write_r : std_logic_vector(data_width_g-1 downto 0)
- Signal write_out : std_logic_vector(0 to data_width_g-1)

Parenthesis to show the order of evaluation
- A and ( x or b)

Check the VHDL coding rules used in the course
- Not just tidyness, affects also performance/area of the design

Not supported by synthesis

Not supported:
- physical types
  - time, for example
- access types
- file types
- guard expression
- signals in packages (global signals)
- signal and variable initialization
  - Typically ignored (there are exceptions, e.g. Xilinx FPGA synthesis)
- multiple wait statements
- more than one ‘event in a process
Guidelines

Asynchronous reset is used only to initialize
- Not part of the functionality

Modularize the design to components
- Easier to design single components
  - Better results
- Easier to upgrade

Get rid of red signals

Red signals in wave form viewer indicate serious problems!
Signal does not have any value - undefined 'U'
- Any logic reading that signal will malfunction

Signal has conflicting values - result is ‘X’
- It is driven in 2+ places in code that will be executed in parallel
- two processes (incl. reset part of sync. process)
- in process and concurrent assignment (=outside all processes)
- multiple concurrent assignments
- port of subcomponent and signal in higher-level component

Your first task is to remove all of them!
Get rid of red signals (2)

Undefined value 'U' at input ruins both outputs

Conflict between '1' driven by DUV and '0' driven by TB.
Short-circuit between GND and VDD. zap!

Conflict between '0'
driven by DUV and
'1' driven by TB.

Values 'Z', 'H' or 'L' driven by TB do not cause conflicts.

Conflict between values given in process and in concurrent assignment.

Summary

Large system design: need for hierarchical methodology
- System partition
- Components instantiation
- Libraries and packages

Generate Statements

Parameterized design: scalability and reusability
- Genrics
- Unconstrained arrays
- Two-dimensional arrays

General mistakes