HIERARCHICAL DESIGN
Outline

1. Introduction
2. Components
3. Generics
4. Configuration
5. Other supporting constructs
1. Introduction

- How to deal with 1M gates or more?
- Hierarchical design
  - Divided-and-conquer strategy
  - Divide a system into smaller parts
sequential multiplier

control path

state register

next-state logic

data path

data register

functional units

routing network

adder

decrementor
Benefits of hierarchical design

• Complexity management
  – Focus on a manageable portion of the system, and analyze, design and verify each module in isolation.
  – Construct the system in stages by a designer or concurrently by a team of designers.
  – Help synthesis process
• Design reuse
  – Use predesigned modules or third-party cores
  – Use the same module in different design
  – Isolate device-dependent components (e.g., SRAM)
Relevant VHDL constructs

- Component
- Generic
- Configuration
- Library
- Package
- Subprogram
2. 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 entity
- Info similar to entity declaration
- Syntax:

```vhdl
component component_name
  generic ( generic_declaration;
              generic_declaration;
              ...
  );
  port ( port_declaration;
         port_declaration;
         ...
  );
end component;
```
• E.g., a decade (mod-10) counter

```vhdl
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);
    signal r_next: unsigned(3 downto 0);
    constant TEN: 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_next;
        end if;
    end process;
    -- next-state logic
    process(en, r_reg)
    begin
        r_next <= r_reg;
        if (en='1') then
            if r_reg=(TEN-1) then
                r_next <= (others=>'0');
            else
                r_next <= r_reg + 1;
            end if;
        end if;
    end process;
    -- output logic
    q <= std_logic_vector(r_reg);
    pulse <= '1' when r_reg=(TEN-1) else '0';
end up_arch;
• Component declaration for dec_counter

```vhdl
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;
```

![Diagram of dec_counter component]
Component instantiation

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

```vhdl
instance_label: component_name
generic_map(
generic_association;
generic_association;
...
)
port_map(
port_association;
port_association;
...
);
```

• Port association (named association)

```vhdl
port_name => signal_name
```
• 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;

- 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:

    \[
    \text{clk, reset, en, q, pulse}
    \]

  – Alternative component instantiation

    \[
    \text{one_digit: dec_counter}
    \text{\quad port map (clk, reset, en, q_one, p_one)};
    \text{ten_digit: dec_counter}
    \text{\quad port map (clk, reset, p_one, q_ten, p_ten)};
    \]

  – Trouble if the order later changes in entity declaration
• Mapping of constant and unused port
  – E.g.,
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);

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

- Mechanism to pass info into an entity/component
- Declared in entity declaration and then can be used as a constant in port declaration and architecture body
- Assigned a value when the component is instantiated.
- Like a parameter, but has to be constant
• e.g., parameterized binary counter
  – Note that the generic is declared before the port and thus can be used in port declaration

```vhdl
entity para_binary_counter is
  generic (WIDTH: natural);
  port(
    clk, reset: in std_logic;
    q: out std_logic_vector(WIDTH-1 downto 0)
  );
end para_binary_counter;
```
architecture arch of para_binary_counter is
  signal r_reg, r_next: unsigned(WIDTH-1 downto 0);
begin
  process(clk, reset)
  begin
    if (reset='1') then
      r_reg <= (others => '0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  r_next <= r_reg + 1;
  q <= std_logic_vector(r_reg);
end arch;
• e.g., to use the parameterized counter

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity generic_demo is
  port(
    clk, reset: in std_logic;
    q_4: out std_logic_vector(3 downto 0);
    q_12: out std_logic_vector(11 downto 0)
  );
end generic_demo;
```
architecture vhdl_87_arch of generic_demo is

component para_binary_counter
    generic (WIDTH: natural);
    port (
        clk, reset: in std_logic;
        q: out std_logic_vector(WIDTH-1 downto 0)
    );
end component;

begin
    four_bit: para_binary_counter
        generic map (WIDTH=>4)
        port map (clk=>clk, reset=>reset, q=q_4);
    twee_bit: para_binary_counter
        generic map (WIDTH=>12)
        port map (clk=>clk, reset=>reset, q=q_12);
end vhdl_87_arch;
• e.g., parameterized mod-n counter
  – Count from 0 to n-1 and wrap around
  – Note that WIDTH depends on N

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod_n_counter is
generic ( 
  N: natural;
  WIDTH: natural
);
port ( 
  clk, reset: in std_logic;
  en: in std_logic;
  q: out std_logic_vector(WIDTH-1 downto 0);
  pulse: out std_logic
);
end mod_n_counter;
```
architecture arch of mod_n_counter is
  signal r_reg: unsigned(WIDTH-1 downto 0);
  signal r_next: unsigned(WIDTH-1 downto 0);
begin
  -- register
  process(clk, reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  process(en, r_reg)
  begin
    r_next <= r_reg;
    if (en='1') then
      if r_reg=(N-1) then
        r_next <= (others=>'0');
      else
        r_next <= r_reg + 1;
      end if;
    end if;
  end process;
  -- output logic
  q <= std_logic_vector(r_reg);
pulse <= '1' when r_reg=(N-1) else '0';
end arch;
• E.g., the 2-digit decimal counter again
one_digit: mod_n_counter
  generic map (N=>10, WIDTH=>4)
  port map (clk=>clk, reset=>reset, en=>en,
             pulse=>p_one, q=>q_one);

ten_digit: mod_n_counter
  generic map (N=>10, WIDTH=>4)
  port map (clk=>clk, reset=>reset, en=>p_one,
             pulse=>p_ten, q=>q_ten);

p100 <= p_one and p_ten;
• Another useful application of generic: passing delay information

\[ y \leq a + b \text{ after } T_{pd} \text{ ns}; \]
4. Configuration

• Bind a component with an entity and an architecture
• Flexible and involved.
• Only simple binding of entity and architecture is needed in synthesis
  – Entity: like a socket in a printed circuit board
  – Architecture: like an IC chip with same outline
• Not supported by all synthesis software
• Application of binding:
  – E.g., adder with different speed: Fast but large adder or small but slow adder
  – E.g., Test bench descriptions at different stages
• Type of configuration:
  – Configuration declaration (an independent design unit)
  – Configuration specification (in architecture body)

• 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
• Configuration declaration
  – An independent design unit
  – Simplified syntax

```plaintext
configuration conf_name of entity_name is
  for architecture_name
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
  for instance_label: component_name
    use entity lib_name.bound_entity_name(bound_arch_name);
  end for;
  . . .
end for;
end;
```
• E.g., create two architecture bodies for the decade counter (one up and one down)

```vhdl
architecture down_arch of dec_counter is
  -- next-state logic
  process (en, r_reg)
  begin
    r_next <= r_reg;
    if (en='1') then
      if r_reg=0 then
        r_next <= to_unsigned(TEN-1,4);
      else
        r_next <= r_reg - 1;
      end if;
    end if;
  end process;
  -- output logic
  q <= std_logic_vector(r_reg);
  pulse <= '1' when r_reg=0 else '0';
```
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;

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;
configuration count_down_config of hundred_counter is
  for vhdl_87_arch
    for one_digit: dec_counter
      use entity work.dec_counter(down_arch);
    end for;
    for ten_digit: dec_counter
      use entity work.dec_counter(down_arch);
    end for;
  end for;
end;
• Configuration specification
  – Included in the declaration section of architecture body

• Syntax:

```vhdl
for instance_label: component_name
  use entity lib_name.bound_entity_name(bound_arch_name);
for instance_label: component_name
  use entity lib_name.bound_entity_name(bound_arch_name);
```
• E.g.,

```vhdl
architecture vhdl_87_config_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;
for one_digit: dec_counter
    use entity work.dec_counter(down_arch);
for ten_digit: dec_counter
    use entity work.dec_counter(down_arch);
signal p_one, p_ten: std_logic;
begin
```
• Component instantiation and configuration in VHDL 93
  – Remove component and configuration declaration
  – Usually satisfactory for RT-level synthesis
  – Syntax:

```vhdl
instance_label:
  entity lib_name.bound_entity_name(bound_arch_name)
    generic map ( . . . )
    port map ( . . . );
```
• E.g.,

```vhdl
architecture vhdl_93_arch of hundred_counter is
  signal p_one, p_ten: std_logic;
begin
  one_digit: entity work.dec_counter(up_arch)
    port map (clk=>clk, reset=>reset, en=>en,
              pulse=>p_one, q=>q_one);
  ten_digit: entity work.dec_counter(up_arch)
    port map (clk=>clk, reset=>reset, en=>p_one,
              pulse=>p_ten, q=>q_ten);
  p100 <= p_one and p_ten;
end vhdl_93_arch;
```
5. Other constructs for developing large system

- Library
- Subprogram
- Package
Library

- A virtual repository to stored analyzed design units
- Physical location determined by software
- Design units can be organized and stored in different libraries
• Default library: work
  – E.g.,

  ```vhdl
two_digit: entity work.dec_counter(up_arch)
```

• Non-default library has to be declared:
  – syntax:

  ```vhdl
library lib_name, lib_name, ..., lib_name;
```

  – E.g., library ieee;
• E.g.,

```vhdl
library c_lib;  -- make c_lib visible
configuration clib_config of hundred_counter is
  for vhdl_87_arch
    for one_digit: dec_counter
      use entity c_lib.dec_counter(down_arch);  -- c_lib
    end for;
    for ten_digit: dec_counter
      use entity c_lib.dec_counter(down_arch);  -- c_lib
    end for;
  end for;
end clib_config;
```
Subprogram

• Include function and procedure
• Made of sequential statement
• Is not a design unit; must be declared
• Aimed for software hierarchy not hardware hierarchy
• We only use function
  – Shorthand for complex expression
  – “House-keeping tasks; e.g., type conversion
• Syntax of function

```vhdl
function func_name(parameter_list) return data_type is
  declarations;
begin
  sequential statement;
  sequential statement;
  ...
  return(expression);
end;
```
• E.g.,

```vhdl
architecture arch of ...

-- declaration

function maj(a, b, c: std_logic) return std_logic is
  variable result: std_logic;
begin
  result := (a and b) or (a and c) or (b and c);
  return result;
end maj;

signal i1, i2, i3, i4, x, y: std_logic;
begin
  ...
  x <= maj(i1, i2, i3) or i4;
  y <= i1 when maj(i2, i3, i4)=’1’ else ...
  ...
```
• E.g.,

```vhdl
function to_boolean(a: std_logic) return boolean is
  variable result: boolean;
begin
  if a='1' then
    result := true;
  else
    result := false;
  end if;
  return result;
end to_boolean;
```
E.g., \([\log_2 n]\)

```plaintext
function log2c(n: integer) return integer is
    variable m, p: integer;
begin
    m := 0;
p := 1;
    while p < n loop
        m := m + 1;
p := p * 2;
    end loop;
return m;
end log2c;
```
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
• Syntax

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

package body package_name is
    subprogram;
    subprogram;
    ...
end package_name;
```
• E.g.,

```vhdl
-- package declaration
library ieee;
use ieee.std_logic_1164.all;
package util_pkg is
  type std_logic_2d is
      array (integer range <>, integer range <>) of std_logic;
  function log2c (n: integer) return integer;
end util_pkg;
```
package body

package body util_pkg is
    function log2c(n: integer) return integer is
        variable m, p: integer;
        begin
            m := 0;
            p := 1;
            while p < n loop
                m := m + 1;
                p := p * 2;
            end loop;
        return m;
    end log2c;
end util_pkg;
• Improved mod-n counter

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity better_mod_n_counter is
  generic (N: natural);
  port(
    clk, reset: in std_logic;
    en: in std_logic;
    q: out std_logic_vector(log2c(N)-1 downto 0);
    pulse: out std_logic
  );
end better_mod_n_counter;

architecture arch of better_mod_n_counter is
  constant WIDTH: natural := log2c(N);
  signal r_reg: unsigned(WIDTH-1 downto 0);
  signal r_next: unsigned(WIDTH-1 downto 0);
begin
```
6. Partition

• Physical partition:
• Division of the physical implementation
• Each subsystem is synthesized independently
• Partition too small: loose optimization opportunity
• Partition too large: require too much resource
  – e.g., $O(n^3)$ algorithm 1000 gates for 1 sec;
  – 35 hours ($50^3$ sec) for one 50,000 gate circuit
  – 21 min ($10 \times 5^3$ sec) for 10 5,000 gate circuit
• 5000 to 50,000 gates for today’s synthesizer
• Logical partition:
  – Help development and verification process for human designers
  – Logical partitions can be merged later in synthesis

• Some circuit should be isolated as independent modules
  – Device-dependent circuit: e.g., memory modules