COMIT SYSTEMS
The Contract Engineering Company
Design Advantage®
Comit Systems, Inc. Newsletter
We build them for YOU.SM
Volume 4 Number 3 May - Jun 2001

Recursive VHDL structures in FPGA synthesis

By Rangarajan Sundaravaradan

New standards in VHDL-93 permit the writing of VHDL components that instantiate themselves. Implementation of recursive structures leads to designs easily pipelined by the addition of registers at the outputs of the recursively instantiated components. In general, the recursive structures are easier to develop and can be expressed clearly, making them easier to understand. This article explores practical applications of recursive structures in FPGA synthesis.

Most hardware structures that are designed consist of a number of instances of a basic component interconnected in a regular pattern. In order to describe such repetitive structures, VHDL provides a mechanism called the generate statement. The language permits both conditional and repetitive forms of generate statement. A recursive structure is one, which is parameterized with respect to its size and is described in terms of smaller instances of the same structure. The examples in this article cover recursion using subprograms and entities.

The Wide XOR component

XOR component using a simple function:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity XOR_reduce is
  generic (N: natural := 5);
  port (data_in: in std_logic_vector(N-1
downto 0);
        data_out: out std_logic);
end XOR_reduce;
architecture one of XOR_reduce is
  function
XOR_reduce_func(data:std_logic_vector)retu
rn std_logic is
    variable result : std_logic;
  begin
    result := '0';
    for I in data'RANGE loop
      result := result XOR data(I);
    end loop;
  return result;
end;
begin
  data_out <= XOR_reduce_func(data_in);
end one;
```

XOR component using a recursive function:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity XOR_tree is
  generic (N: natural := 4);
  port (data_in: in std_logic_vector(N
downto 0);
        data_out: out std_logic);
end XOR_tree;
architecture one of XOR_tree is
  function XOR_tree_func(data:
std_logic_vector) return std_logic is
    variable UPPER_TREE, LOWER_TREE: std_logic;
    variable MID, LEN: natural;
    variable result: std_logic;
    variable i_data:
std_logic_vector(data'LENGTH-1 downto 0);
begin
    i_data := data;
    LEN := i_data'LENGTH;
      if LEN = 1 then
        result := i_data(i_data'LEFT);
      elsif LEN = 2 then
        result := i_data(i_data'LEFT) XOR
i_data(i_data'RIGHT);
      else
        MID := (LEN + 1)/2 + i_data'RIGHT;
        UPPER_TREE :=
XOR_tree_func(i_data(i_data'LEFT downto MID));
        LOWER_TREE := XOR_tree_func(i_data(MID-
1 downto i_data'RIGHT));
        result := UPPER_TREE XOR LOWER_TREE;
      end if;
    return result;
end;
begin
  data_out <= XOR_tree_func(data_in);
end one;
```

Wide XOR using recursive components:
In this example the different levels of logic are pipelined.

```
    entity xor_gate is
      generic(pipeline_delay:natural:=0 ;);
       port ( X : in std_logic_vector ;
                 q : out std_logic ;
                 clk : in std_logic ;
                 enable : in std_logic
               ) ;
    end xor_gate ;
architecture xor_gate_a of xor_gate is
begin

a: if  X'length < N generate
   --basic Xor gate component is instantiated
   here
    end generate a;

b: if X'length > N generate
constant Y: natural := (X'length -1)/N ;
signal temp : std_logic_vector(0 to Y);

begin
  C : for I in 0 to Y-1 generate
    D: entity xor_gate generic map
(pipeline_delay => pipeline_delay)
       Port map (X => X(N*I to (N*(I+1) -1) ),q
=> temp(i) ,
```

```
-- Now for the end input bits
      E : entity xor_gate generic map
(pipeline_delay => pipeline_delay )
             Port map (X => X(N*Y to X'length
-1 ),q => tmp(Y),
                              Clk => clk
,enable => enable );
      F : entity xor_gate generic map
(pipeline_delay => pipeline_delay )
              Port map (X => temp ,q => q
,clk=> clk ,enable => enable);

    End generate b;

End xor_gate_a ;
```

In the above case the basic instance is parameterized to size N bits.

Wide XOR component targeted for ORCA2

Implementing the basic instance of the wide XOR component targeted to a Lucent ORCA@ device:

Each PFU in ORCA2 uses three input data buses (A[4:0], B[4:0], WD[3:0]), four control inputs (C0, CK, CE, LSR), and a carry input (CIN); the last is used for fast arithmetic functions. There is a 5-bit output bus (O[4:0]) and a carry-out (COUT) from the PFU.

A(4:0) and B(4:0) are inputs to the lookup table and WD(4:0) are direct inputs to the flipflops.

Basic XOR gate instance can be constructed upto 11 bits in width using a(4:0),b(4:0) and the carry input(CIN).A single PFU can implement XOR gate upto 11 bits with 0 or 1 cycle of clock delay.

Construction of larger XOR functions require another component which instantiates the base 11 bit XOR gate as shown in the example above.

The advantages with recursion are

- The component is broken into different levels of logic; pipelining the individual stages is easier.
- Timing can be improved by adopting recursive structures.
- Recursive structures are easier to develop.
- High performance in terms of speed and area can be achieved by adjusting the base component to fit the target architecture. ■

References.

1. Designer's Guide to VHDL – Peter J. Ashenden
2. Recursive and Repetitive Hardware Models in VHDL – Peter J. Ashenden
3. Vector Pipeline Library manual – John McCluskey.