IAS0600 LAB REPORT 5
PARAMETERIZABLE MULTIPLIER

Kayode Hadilou ADJE

194360MAHM

Introduction

This lab dealt with iterative or conditional elaboration of a portion of a hardware description using *generate statements*. *Generate statements* simplifies description of designs, it helps in specifying a group of identical components using one component specification and repeating through the generate mechanism or turning on/off blocks of logic in description. In this lab generate statement is practiced with parameterizable multiplier.

Background

For this practical lab, the background sources I benefitted from are listed and detailed below:

-Lecture Explanation and Slides: I relied on and used lecture slides and explanations about *generic* and *generate statements*.

-Lab Sheet: The explanations in the lab sheet was a guide for the completion of the tasks;

-The cumulated knowledge gained in previous lab work and first trainings practical classes about VHDL and Vivado were of great use.

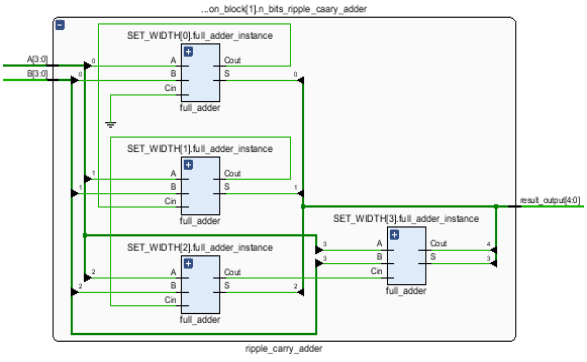
Workflow

This lab was about implementing a parameterizable multiplier using generate statements. The lab was divided into 2 tasks and another optional task that I didn't do:

-implementation of a parameterizable ripple carry adder with an adjustable size.

-implementation of a parameterizable multiplier using above described ripple carry adder and columnar addition logic.

Task1: Ripple carry adders are made of a static number of full adders cascaded together. Given two numbers A and B, every full adder of the set adds corresponding two bits from A and B to produce a sum and a carry output that is propagated to the next carry input of the next adder in the chain. The initial carry input can be set to zero. In order for the design to be reusable, I used *generic* to set the width of binary numbers A and B. And as required in the lab, I used *for generate statement* to generate N full adders instances (N = sizeOf(A)).




```
generic (size : natural := 4);

s_CARRY(0) <= '0';

SET_WIDTH : for i in 0 to size-1 generate
 full_adder_instance : full_adder
     port map (
       A  => A(i),
       B  => B(i),
       Cin => s_CARRY(i),
       S   => s_SUM(i),
       Cout => s_CARRY(i+1) );
end generate SET_WIDTH;

result_output <= s_CARRY(size) & s_SUM;
```

Figure1: Ripple Carry Adder RTL schematic Listing1: Ripple Carry Adder (VHDL) description

Task 2: Columnar addition method allows the use of *generate statement* to perform multiplication. To multiply a binary number A by another binary number B (A and B have the same size which is known and static), the number B is traversed from right to left and at each point, we multiply the corresponding bit of number B by the number A. N values where n is the size of A and B are then obtained and by adding those numbers (after a kind of right shift as in multiplication of decimal numbers) the result corresponding to the multiplication of A by B is obtained. To multiply a bit by a number, a logical AND is used, the two numbers are traversed bit by bit and if bits at the same position are '1', an output of 1 is produced else '0'.

The *for generate* statement in listing 2 is used to generated the ANDed values that will be added later on. Note that to store the result of the ANDed values, an array of *std_logic_vector* is created as *adder_signal.*

```
--for-generate  values to be added
 Traversing1rstNumber: for i in 0 to size-1 generate
    Traversing2ndNumber: for j in 0 to size-1 generate
        adder_signal(i)(j) <= B_mul(i) and A_mul(j);
    end generate;
end generate;
```

Listing 2: ANDed values' logic description

The next step consisted of adding the ANDed values using another *for generate statement* by instantiating the ripple carry adder description. The *kind of shift* logic is performed using iteration index.

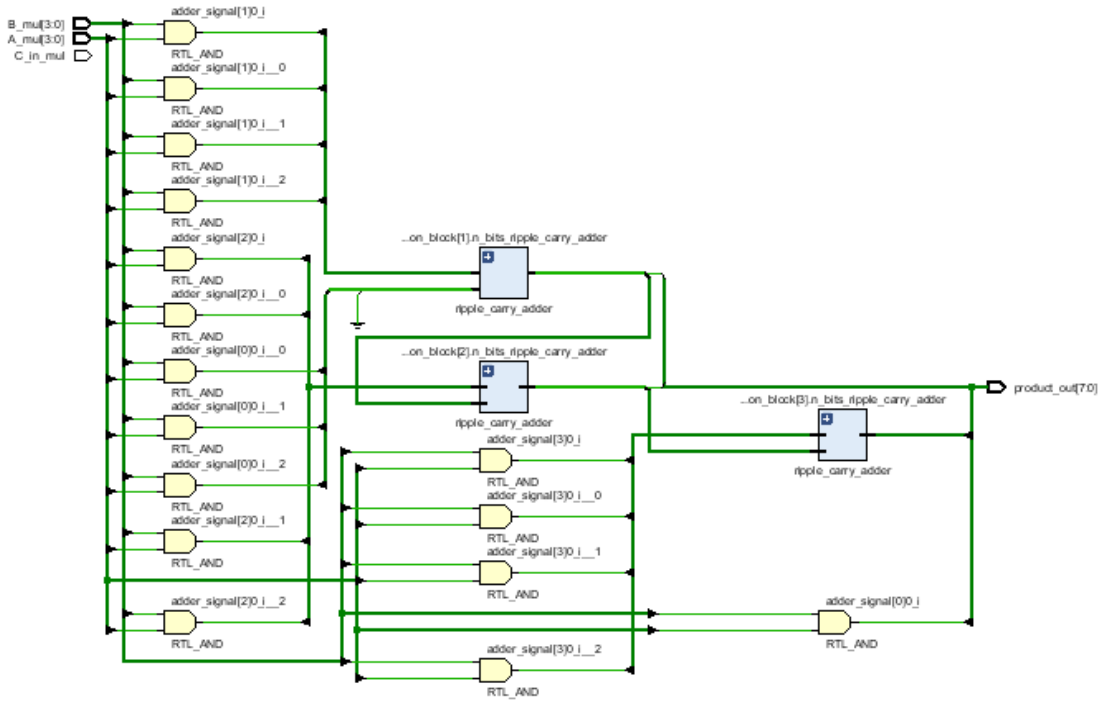
```
product_signal(0) <= '0' & adder_signal(0);--initialize to first number of adder list if not will encounter null
multiplication_block: for k in 1 to size-1 generate
      n_bits_ripple_carry_adder: component ripple_carry_adder port map(A => adder_signal(k),
                                                                       B => product_signal(k-1)((size) downto 1),
                                                                       result_output => product_signal(k) );


end generate;
```

Listing 3: Addition of ANDed values with ripple carry adder.

The result of the multiplication is obtained by including the shift logic in columnar addition as shown is listing 4.

```
finally:for k in 0 to size-2 generate
    product_out(k)<= product_signal(k)(0);
end generate;
product_out((2*size-1) downto size-1) <= product_signal(size-1);
```

The RTL schematic corresponding to the multiplier description is as follow:



The *for generate statement* have generated $size^2$ AND gates and $size - 1$ cascaded ripple carry adder blocks. Each ripple carry adder has its own *size* number of full adders. The output is obtained by combination between ANDed values and outputs of ripple carry adders

Results and Discussion

Both ripple carry adder and multiplier description were tested using testbench. In both cases, the unit under test was instantiated with binary numbers of 4 bits. *For loops* in the range of 0 to 15 (since we are dealing with 4 bits) are used to assign input to binary numbers A and B. In the first case of ripple carry adder since the result was so easy to see in the waveform there was no need to use assert statements. But in the case of the multiplier, assert statements were used to report any errors that may be happen. The outputs from the unit under test were compared to the direct multiplication result. Below are screenshots of both tests.

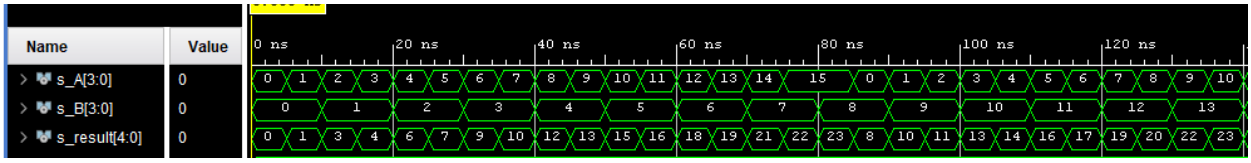


Figure 2 Ripple Carry Adder Testbench Waveform

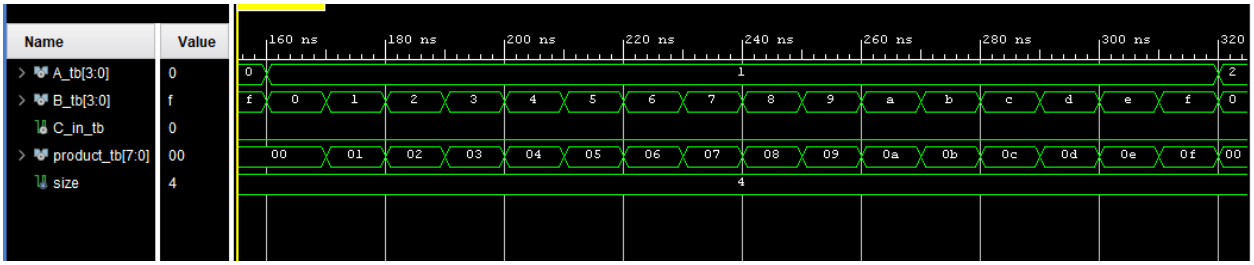


Figure 3 Multiplier Testbench Waveform

Conclusion

To summarize in this lab, I have learnt how to use generate statements (with for and if) to repeat component instantiation or turn on/off some blocks and make descriptions reusable. The lab was completed successfully.