Laboratory 6

Parameterizable Multiplier

Digital Systems Design with VHDL
IAS0600

Marcel Cases Freixenet

Tallinn,
November 7, 2018

Tallinna Tehnikaülikool
TalTech

Contents

1 Introduction

1.1 Aim

This project consists in the design, simulation and implementation in VHDL of a parameterizable multiplier using *generate* and *for* statements and structural design. Given a number *data_width*, the synthesizer will reconfigure itself in order to perform a multiplication using *data_width* switches by *data_width* switches. The result will be shown on four 7-segment displays in hexadecimal.

1.2 Background

A digital multiplication can be performed in several ways. Two possible methods, as recommended in this laboratory, are:

- ❒ Ripple Carry Adders [1]
- ❒ Carry Lookahead

For this project I have chosen Ripple Carry Adders method.

A Ripple Carry Adder consists of multiple Full Adders [2], as many as bits we are multiplying. They are connected to each other through their *carry* signals.

Following: a 4 bit Ripple Carry Adder configuration for *data_width = 4*.

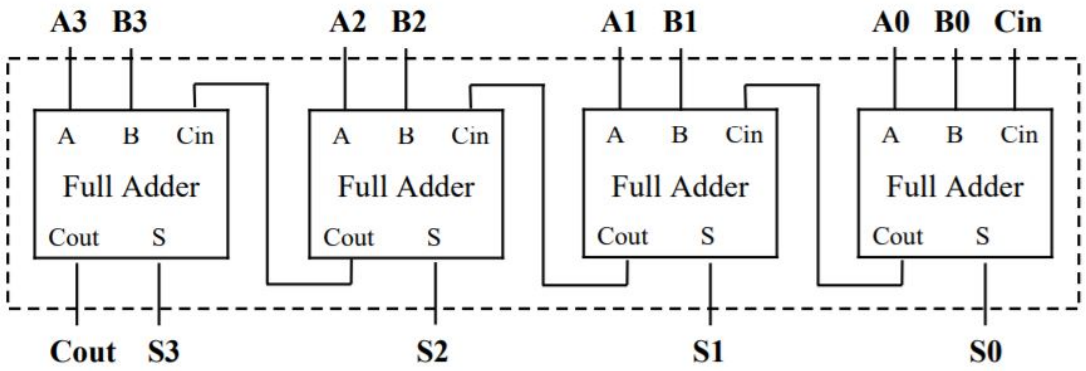

Figure 1.1 *A 4-bit Ripple Carry Adder*

Using **parameterizable Ripple Carry Adders** as components, they can be instantiated in a higher lever to perform a columnar multiplication. Then, in order to perform an *n*-bit multiplication, it is required to instantiate *n-1* adders of size *n*. This structure is rather regular, so using for/generate in this case is

quite appropriate.

An example of columnar multiplication is found below for *data_width* = *4* and using 13 *base 10* as the first element and 10 *base 10* as the second element.

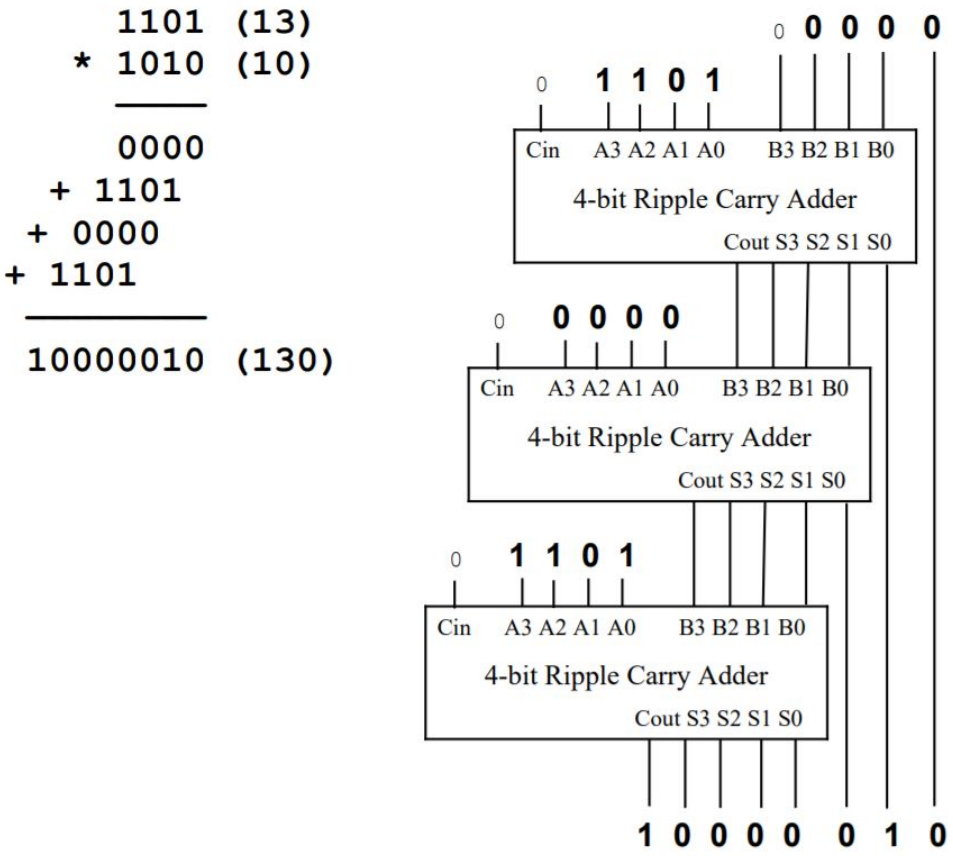

Figure 1.2 *A 4-bit Columnar Multiplication*

2 Workflow

2.1 Top level file

2.1.1 Libraries

Two libraries have been used for this project:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
```

2.1.2 Entity

A clock and reset signals **are not necessary** for this Lab because all of the calculations are concurrent, but as an extra, to show the result of the multiplication on four of the 7-segment displays, a clock and its divided signal (1kHz) are necessary to refresh and multiplex the values on them.

A **generic** value is used in the entity of the top level file to indicate the size of the multiplier. As long as it is located in the top level file, it will have priority over the rest of the statements and components.

```
1  entity parameterizable_multiplier is
2          generic (   data_width : integer := 4
3                  );
4          port (  -- Input (16 switches, base 2) and
5                  -- Output (4 7SEG-display, base 16)
6                  clk : in std_logic ; -- Internal clock (100MHz)
7                  reset : in std_logic ; -- Upper button on Nexys 4 board
8                  sw : in std_logic_vector (15 downto 0);
9                  cat : out std_logic_vector (7 downto 0);
10                 an : out std_logic_vector (3 downto 0);
11                 result_to_testbench : out std_logic_vector (15 downto 0)
12                 );
13 end parameterizable_multiplier;;
```

The signal *result_to_testbench* has the only purpose to communicate to the Testbench the current value of the result so an assertion can be performed.

2.1.3 Architecture

A *Ripple Carry Adder* is used as a component in this project. This is a good way to use it and to instantiate the component an undetermined number of times that will depend on a parameter, *data_width*.

A 7-segment decoder and a clock divider are used as components in order to show the result of the multiplication on the 7-segments display in hexadecimal.

Regarding the internal signals used in the Parameterizable Multiplier, a new type is created, *ripple_carry_adder_bus_type*. This is an unconstrained type that will allow us to create internal signals a multiple number of times, and will give us the ability to use them as a matrix (for example, *ripple_carry_adder_input(1)(2)* refers to the 3rd element *(position 2 in matrix)* of the second bus of inputs *(position 1 in matrix)*).

```
architecture Behavioral of parameterizable_multiplier is

        -- Component *RIPPLE CARRY ADDER*
        component ripple_carry_adder is
                generic (   data_width : integer := data_width
                        );
                port (  a, b: in std_logic_vector (data_width-1 downto 0);
                        cin: in std_logic;
                        s: out std_logic_vector (data_width-1 downto 0);
                        cout: out std_logic
                        );
        end component;

        -- Component *7SEG HEXA DECODER*
        component decoder is
                port (  value : in std_logic_vector (3 downto 0);
                        cat : out std_logic_vector (7 downto 0)
                        );
        end component;

        -- Component *CLOCK DIVIDER*
        component clock_divider is
                generic (   eoc: integer := 99999
                        ); -- 1kHz
                port (  clk : in STD_LOGIC;
                        reset : in STD_LOGIC;
                        clk_div : out STD_LOGIC
                        );
        end component;

type ripple_carry_adder_bus_type is array (natural range <>) of
↪  std_logic_vector(data_width-1 downto 0);
        signal ripple_carry_adder_input :
        ↪  ripple_carry_adder_bus_type(data_width-1 downto 0); -- Inputs 'A'
        ↪  for each signal ripple_carry_adder component
        signal ripple_carry_adder_s : ripple_carry_adder_bus_type(data_width-1
        ↪  downto 0);  -- Output sum from component ripple_carry_adder
signal ripple_carry_adder_carryout : std_logic_vector (data_width-2 downto 0)
↪  := (others => '0') ; -- Output from component ripple_carry_adder
signal result : std_logic_vector (15 downto 0) := X"0000";
signal result_mux : std_logic_vector (3 downto 0); -- Result signal previous
↪  to decoder
signal current_display : integer range 0 to 3;
signal clk_1kHz : std_logic ; --1kHz
```

The following code is using *for . . . generate* statement according to the parameter *data_width* described at the top of the entity.

```
begin

gen_rca_component : for i in 0 to data_width-2 generate -- "0 to data_width - 2" because for 'n' inputs it uses 'n-1' RCAs of size 'n'
        rca_component_number_0 : if i = 0 generate
                inst_ripple_carry_adder : ripple_carry_adder
                        generic map (   data_width => data_width
                                )
                        port map (  a => ripple_carry_adder_input(1),
                                b(data_width-1) => '0',
                                b(data_width-2 downto 0) => ripple_carry_adder_input(0)(data_width-1 downto 1),
                                cin => '0',
                                s => ripple_carry_adder_s(0),
                                cout => ripple_carry_adder_carryout(0)
                                );
        end generate;
        rca_component_number_greater_than_0 : if i > 0 generate
                inst_ripple_carry_adder : ripple_carry_adder
                        generic map (   data_width => data_width
                                )
                        port map (  a => ripple_carry_adder_input(i+1),
                                b(data_width-1) => ripple_carry_adder_carryout(i-1),
                                b(data_width-2 downto 0) => ripple_carry_adder_s(i-1)(data_width-1 downto 1),
                                cin => '0',
                                s => ripple_carry_adder_s(i),
                                cout => ripple_carry_adder_carryout(i)
                                );
        end generate;
end generate;

gen_rca_input : for i in 0 to data_width - 1 generate
        ripple_carry_adder_input(i) <= sw(2*data_width-1 downto data_width) when sw(i) = '1' else (others => '0');
end generate;

gen_result : for i in 0 to data_width-2 generate
        gen_result_0 : if i = 0 generate
                result(0) <= ripple_carry_adder_input(0)(0);
        end generate;
        gen_result_greater_than_0 : if i > 0 generate
                result(i) <= ripple_carry_adder_s(i-1)(0);
        end generate;
        gen_result_msbits : if i = data_width-2 generate
                result(i+data_width downto i+1) <= ripple_carry_adder_s(i);
                result(i+data_width+1) <=
                 ripple_carry_adder_carryout(data_width-2); --carryout
        end generate ;
end generate ;

result_to_testbench <= result; -- Function: sends the internal signal 'result' to a physical port in order to perform an 'assert' on Testbench

-- RESULT TO DISPLAYS
-- Values to show on dispays (mux)-- Values to show on dispays (mux)
```

```
51  with current_display select
52          result_mux <=
53                  result(3 downto 0)      when 0,
54                  result(7 downto 4)      when 1,
55                  result(11 downto 8)     when 2,
56                  result(15 downto 12)    when 3,
57                  "0000"              when others;
58
59  inst_decoder_display_i : decoder
60          port map (  value => result_mux,
61                  cat => cat
62                  );
63
64  -- Clock divider to 1kHz (Display mux shift frequency)
65  inst_clock_divider : clock_divider
66          port map (  clk => clk,
67                  reset => reset,
68                  clk_div => clk_1kHz
69                  );
70
71  -- Display mux
72  process (clk, reset) begin
73          if rising_edge(clk) then
74                  if reset = '1' then
75                          current_display <= 0;
76                  elsif clk_1kHz = '1' then
77                          current_display <= current_display + 1;
78                  end if;
79          end if;
80  end process;
81
82  -- Anode mux
83  gen_an: for i in 0 to 3 generate
84          an(i) <= '0' when current_display = i and i <= 3 and reset /= '1' else
            ↪  '1';
85  end generate gen_an;
86
87  end Behavioral;
```

2.2 Components

2.2.1 Ripple Carry Adder

A Ripple Carry Adder, as seen in ➪ Figure 1.1, requires n Full Adders for n bits. They are instantiated according to a *generate* process (*gen_full_adder*). Inside this generate statement, there are two more generate statements nested. One is for generating the first Full Adder, and the other one generates the rest of Full Adders.

Finally, a concurrent statement routes the most significant bit of the internal signal *carry_int* to the carry output *cout*'s port.

```
1  entity ripple_carry_adder is
2          generic (   data_width : integer := 4
3                  );
```

```
4         port (  a, b: in std_logic_vector (data_width-1 downto 0);
5                 cin: in std_logic;
6                 s: out std_logic_vector (data_width-1 downto 0);
7                 cout: out std_logic
8                 );
9 end ripple_carry_adder;
10
11
12 architecture Behavioral of ripple_carry_adder is
13
14         -- Component *FULL ADDER*
15         component full_adder is
16                 port (  a_fa : in std_logic;
17                         b_fa : in std_logic;
18                         cin_fa : in std_logic ;
19                         sum_fa : out std_logic;
20                         cout_fa : out std_logic
21                         );
22         end component;
23
24         signal carry_int: std_logic_vector (data_width-1 downto 0);
25
26 begin
27
28 gen_full_adder : for i in data_width-1 downto 0 generate
29         least_significant_bit : if i = 0 generate
30                 inst_fa : full_adder
31                         port map (  a_fa => a(0),
32                                 b_fa => b(0),
33                                 cin_fa => cin,
34                                 sum_fa  => s(0),
35                                 cout_fa  => carry_int(0)
36                                 );
37         end generate ;
38         other_bits : if i > 0 generate
39                 inst_fa : full_adder
40                         port map (  a_fa => a(i),
41                                 b_fa => b(i),
42                                 cin_fa => carry_int(i-1),
43                                 sum_fa  => s(i),
44                                 cout_fa  => carry_int(i)
45                                 );
46         end generate ;
47 end generate gen_full_adder;
48
49 most_significant_bit : cout <= carry_int(data_width-1);
50
51 end Behavioral;
```

2.2.2 Clock Divider and Decoder

They are used to show values on four of the 7-segment displays. They are explained in *Laboratory 5. Creeping Line* [3].

2.2.3 Full Adder from three Half Adders

This component is used in this project for the hardware configuration of the Ripple Carry Adder. It is designed and tested in *Laboratory 3. Adder* [4].

2.3 Testbench

This testbench has been designed to be parameterizable, as it is the multiplier that has to be tested. For this purpose, a *generic* value has been declared inside of the entity and has been set to 4. This value will be spread to any parameterizable hardware concerning this project: the testbench, the parameterizable multiplier top level file, the ripple carry adder component and the size of the signals and buses inside them.

In the *stimulus process*, depending on the current value of *data_width*, the corresponding values of the input switches will be calculated, and for each combination an assertion will be performed in order to check the result. If the multiplication is not correct, the test will fail and thus will stop.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity parameterizable_multiplier_tb is
7          generic (   data_width : integer := 8
8          );
9  end parameterizable_multiplier_tb;
10
11 architecture bench of parameterizable_multiplier_tb is
12         component  parameterizable_multiplier is
13                 generic (   data_width : integer := data_width
14                         );
15                         port (  clk : in std_logic ;
16                         reset : in std_logic ;
17                         sw : in std_logic_vector (15 downto 0) := X"0000";
18                         cat : out std_logic_vector (7 downto 0);
19                         an : out std_logic_vector (7 downto 0);
20                         result_to_testbench : out std_logic_vector (15 downto
                           ↪  0)
21                         );
22         end component ;
23
24         signal clk_tb : std_logic := '0';
25         signal reset_tb : std_logic := '1';
26         signal sw_tb : std_logic_vector (15 downto 0);
27         signal cat_tb : std_logic_vector (7 downto 0);
28         signal an_tb : std_logic_vector (7 downto 0);
29         signal result_to_testbench_tb : std_logic_vector (15 downto 0);
30
31 begin
32
33 clk_tb <= not clk_tb after 5ns; --half_period
34 reset_tb <= '0' after 10ns;
35
```

```
36 uut: parameterizable_multiplier
37         generic map(   data_width => data_width
38                 )
39         port map (  clk => clk_tb,
40                 reset => reset_tb,
41                 sw => sw_tb,
42                 cat => cat_tb,
43                 an => an_tb,
44                 result_to_testbench => result_to_testbench_tb
45                 );
46
47 stimulus: process begin
48         generic_loop : loop -- Generates multiplication and asserts
           ↪  multiplication for any value of data_width from 2 to 8
49                 sw_tb <=
                   ↪  std_logic_vector(to_unsigned(to_integer(unsigned(sw_tb) +
                   ↪  1), 2*data_width)); wait for 100 ns;
                   ↪  assert(unsigned(sw_tb(2*data_width-1 downto
                   ↪  data_width))*unsigned(sw_tb(data_width-1 downto 0)) =
                   ↪  unsigned(result_to_testbench_tb)) report "Multiplication
                   ↪  error" severity failure;
50                 exit generic_loop when sw_tb(2*data_width-1 downto 0) =
                   ↪  (2*data_width-1 downto 0 => '1');
51         end loop generic_loop;
52         wait;
53 end process;
54
55 end bench;
```

3 Results and discussion

This is a discussion of the results obtained through the simulation of both the component **Ripple Carry Adder** (which sums two vectors) and the **Parameterizable Multiplier** (using Ripple Carry Adders as components and performing columnar multiplication).

3.1 Simulation

According to the testbench developed in ➪ Section 2.3, the simulation for the **Ripple Carry Adder** component for *data_width* = *4* is the following:

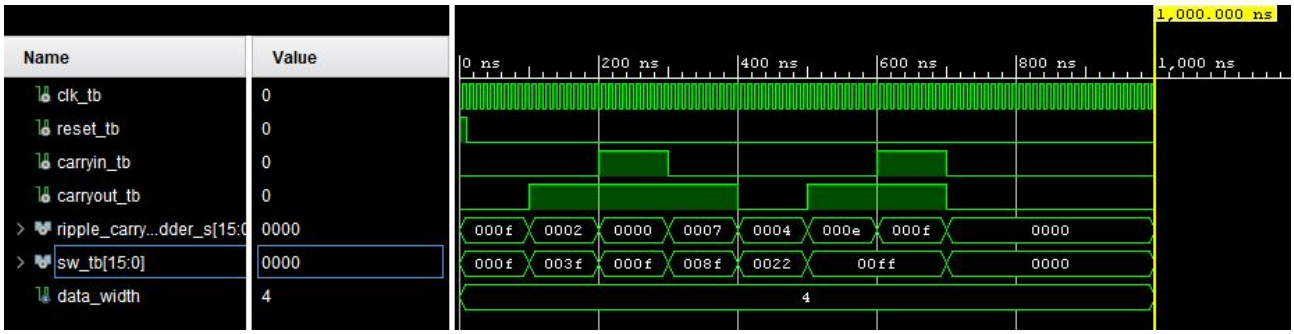

Figure 3.1 *Ripple Carry Adder simulation*

Then the top level file that contains the HDL description of the Parameterizable Multiplier, for *data_width* = *4*, has the following behavior:

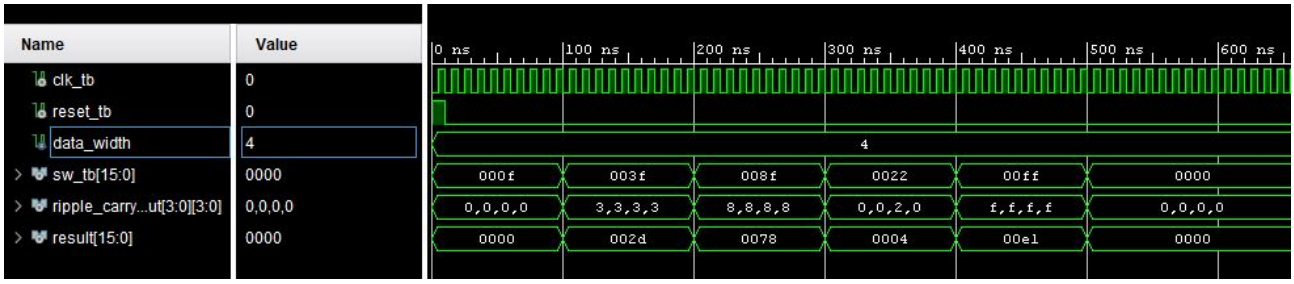

Figure 3.2 *Parameterizable Multiplier simulation*

4 Conclusion

This Lab has shown the ability of the VHDL language to **reconfigure itself** depending on the value of a parameter than can be modified previous to the synthesis.

This is a powerful tool for this hardware description language, which is the **generate** statement. Given a defined integer number, this statement will generate as many hardware elements as the given number requires. It has been used to generate multiple instantiations of **Ripple Carry Adder** components.

We have also experimented with **generic** values, which give the possibility to change the behavior of a component giving a number on the instantiation.

For this Lab, some parts of code have been reused from previous work, like a 7-segment decoder, a clock divider or a full adder made of half adders. They are instantiated in the top level as components, thus this Lab has been developed using **structural design** patterns.

References

[1] University of Victoria. Ripple carry and carry lookahead adders. http://www.ece.uvic.ca/~fayez/courses/ceng465/lab_465/project1/adders.pdf, 2018. 1

[2] Wikipedia. Full adder. https://en.wikipedia.org/wiki/Adder_(electronics), 2018. 1

[3] Marcel Cases Freixenet. A creeping line in vhdl. https://www.marcelcases.com/projects/a-creeping-line-in-VHDL/, 2018. 7

[4] Marcel Cases Freixenet. A two bit adder in vhdl. https://www.marcelcases.com/projects/a-two-bit-adder-in-VHDL/, 2018. 8