Parameterizable Multiplier

Task:

Implement a parameterizable multiplier using **for/generate** statement to describe repeating components/assignments. The size of the multiplier should be adjustable using parameter ***data width*** (possible values are ***2***, ***3***, ***4***, ... , ***8***). Following is the list of steps for the parameterizable multiplier design:

- describe a parameterizable ripple-carry adder. It should be possible to adjust its size using parameter ***data width*** (possible values are ***2***, ***3***, ***4***, ... , ***8***).
- ***optionally***, add a carry-lookahead adder description to the previous design. In this case the adder design should have another adjustable parameter ***adder type*** (possible types are ***ripple carry***, ***carry-lookahead***).
- use parameterizable adder as component to describe a parameterizable multiplier that implements multiplication using columnar addition. It should be possible to adjust its size using parameter ***data width*** (possible values are ***2***, ***3***, ***4***, ... , ***8***).

Simulate and implement each design step on FPGA development board.

Design using For/Generate Statement and Generics

When designing a digital circuit it is quite often required to repeat (reuse) a section of VHDL code to create several instances of the same component/assignment. For example, in order to describe a 4-bit ripple-carry adder (Figure 1), it is needed to instantiate a full adder four times. Each full adder adds two bits of the corresponding order from inputs A and B and produces a sum, while its carry output is propagated to the carry input on the next adder in the chain. In a similar way it is possible to build a ripple-carry adder of size ***n*** bits by instantiating ***n*** full adders. This can be easily done using **for/generate** statement. Note that because support for **loop** statement varies greatly among synthesis tools, it is ***not recommended*** to use it in synthesizable design descriptions.

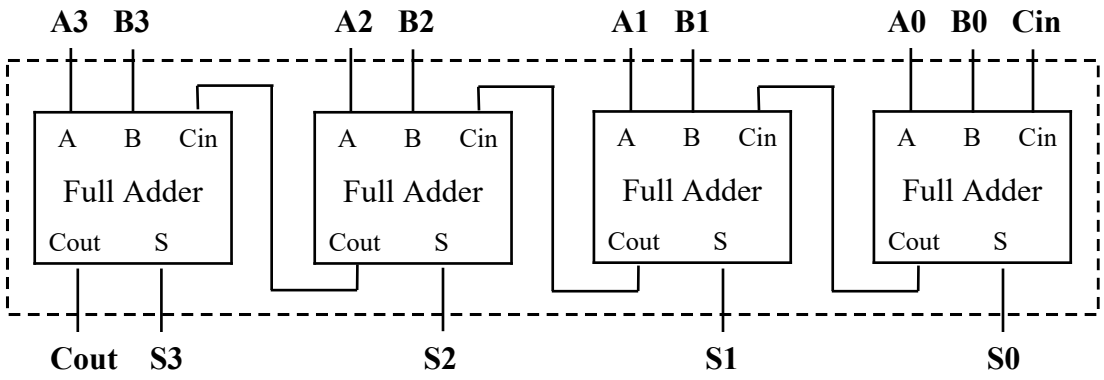

Figure 1: 4-bit Ripple-Carry Adder

One important remark concerning **for/generate** statement is that its range must be static (number of iterations should be known before the synthesis). The number of iterations can be specified using **constant** or **generic**. **Constant** can be declared and initialized in the **architecture** body, thus its value is only visible within that architecture. **Generic** is declared in the **entity** and its value is visible in the **entity** itself as well as in the **architecture** body. An example of the entity for the ***n***-bit ripple-carry adder is shown in Listing 1. Note that the value of ***n*** is actually used in the **entity** to define the size of inputs/outputs for the adder. This is accomplished with declaration of the inputs/outputs as an **std_logic_vector** type, for which the upper bound actually depends on the parameter ***n***. The default value of ***n*** can be specified either in the **generic** statement itself (as shown in Listing 1), or in the instantiation of the RCA component (when it is used at the higher level of hierarchy) using generic mapping.

Listing 1: Entity of the n-bit Ripple-Carry Adder

```
entity RCA is
   generic (n: integer:=4);
      port (A, B: in std_logic_vector (n-1 downto 0);
            Cin: in std_logic;
            S: out std_logic_vector (n-1 downto 0);
            Cout: out std_logic);
end RCA;
```

Binary multiplication can be substituted with addition. In order to multiply two 4-bit values, e.g. 1101 (decimal 13) by 1010 (decimal 10), three additions can be performed

instead as shown in Figure 2 on the left. In order to do the same in hardware, three 4-bit adders can be instantiated and connected as shown in Figure 2 on the right. So 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.

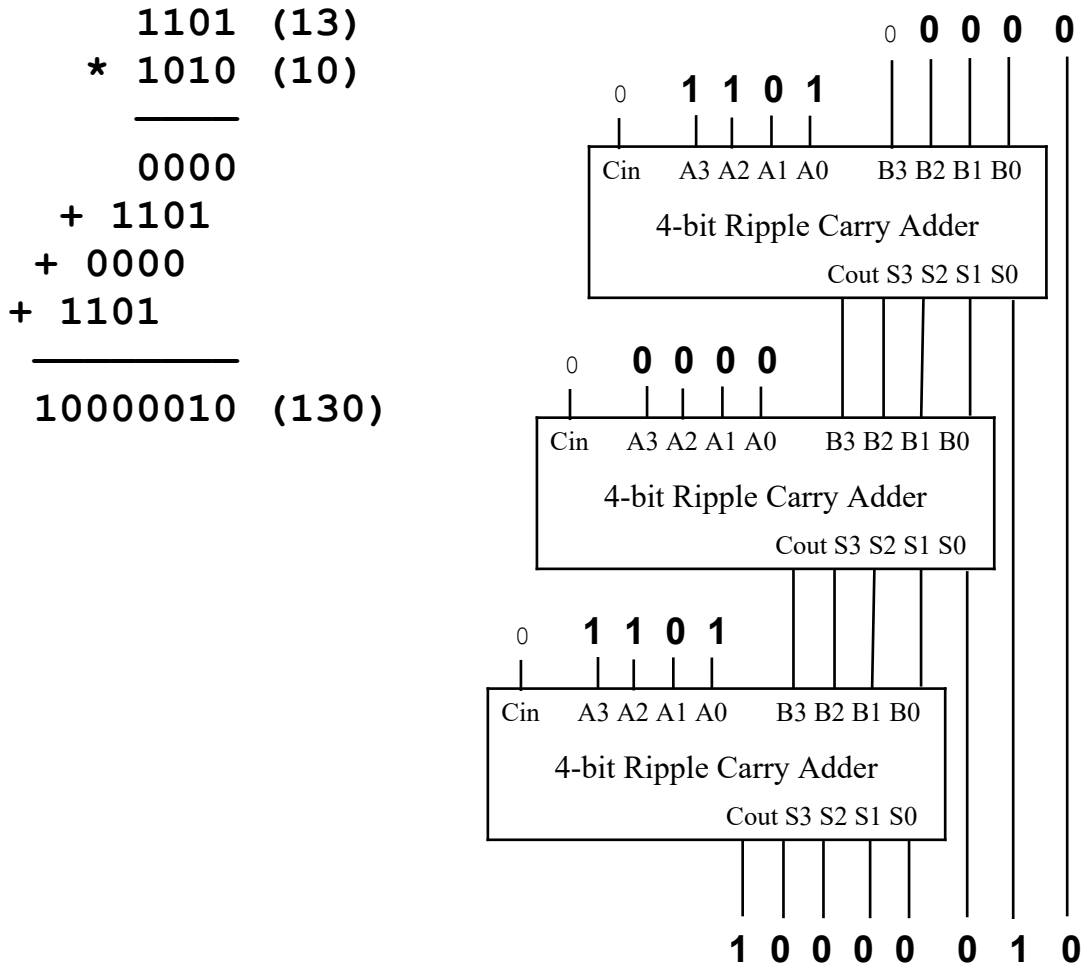

Figure 2: Example of Multiplication Using Columnar Addition with Ripple-Carry Adders

Since ripple-carry adders in Figure 2 have inputs and outputs as buses, it may be more convenient to employ vector arrays in the **for/generate** description of the multiplier. This requires the use of a new type, that declares array of **std_logic_vector**. The array can be constrained (first two line in Listing 2) or unconstrained (last two line in Listing 2). In case of unconstrained array type, the range must be specified in the signal declaration. Note, that in the example of unconstrained array type in Listing 2 it is specified that the array range

values should belong to the **natural** type. Also, the array range may depend on the value of **constant** or **generic** (e.g. in Listing 2 the value of ***n*** is used to constrain the size).

Listing 2: Type Declaration for Array of std_logic_vector

```
type constr_array is array (0 to n-1) of std_logic_vector(n-1 downto 0);
signal constr_array_signal : constr_array;

type unconstr_array is array (natural range <>) of std_logic_vector(n-1 downto 0);
signal unconstr_array_signal : unconstr_array(0 to n-1);
```