Exercise 5

a) 4-bit Multiplier

In this exercise, you must design a 4-bit multiplier.

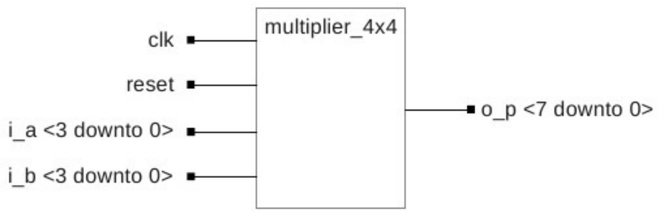

Figure 1: multiplier_4x4

Port declaration:

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic_vector (3 downto 0)
- *i_b* : in std_logic_vector (3 downto 0)
- *o_p* : out std_logic_vector (7 downto 0)

To design this module, you will use a structural description. This means that you describe the structure of the hardware instead of describing its behavior. With a behavioral description, we can multiply two signals in a very simple way like it is shown in the example code.

```
A <= B * C;
```

The synthesis tool will use the dedicated multipliers on the FPGA or, if no multipliers are available, synthesize the code in the usual way. Your task is to describe a 4-bit multiplier using a structural description.

To understand the structure of a binary multiplier, you should first have a look at the scheme of the binary multiplication shown in Table 1.

$b_3\ b_2\ b_1\ b_0\ \ x\ \ a_3\ a_2\ a_1\ a_0$

					b_3 x a_0	b_2 x a_0	b_1 x a_0	b_0 x a_0
			+	b_3 x a_1	b_2 x a_1	b_1 x a_1	b_0 x a_1	
		+	b_3 x a_2	b_2 x a_2	b_1 x a_2	b_0 x a_2		
	+	b_3 x a_3	b_2 x a_3	b_1 x a_3	b_0 x a_3			
SUM:	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Table 1: Binary multiplication scheme

Binary multiplication is very similar to the multiplication of decimal values and is achieved by adding a list of shifted multiplicands according to the digits of the multiplier. You can see an example multiplication in table 2.

1 1 0 1 x 1 0 0 1

					1	1	0	1
			+	0	0	0	0	
		+	0	0	0	0		
	+	1	1	0	1			
	0	**1**	**1**	**1**	**0**	**1**	**0**	**1**

Table 2: Example multiplication

In hardware, we can realize a binary multiplier with an array of multiplier units, as shown in Figure 1. These units multiply two input bits and feed the result to a full adder. With this unit, we are able to multiply two single bits, e.g. b_1 x a_0 , and add a shifted multiplicand, e.g. b_0 x a_1. Since the inputs a_{in} and b_{in} must also be connected to the next multiplier unit, they are also fed directly through as unmodified signals.

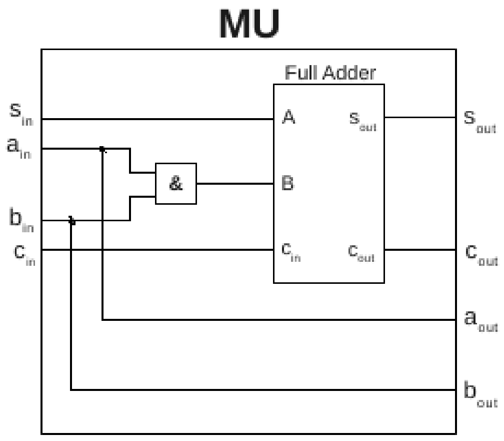

Figure 2: Multiplier unit

For a 4-bit multiplier, 16 MUs are needed and must be connected as shown in Figure 2. Comparing this array of MUs to the scheme shown in Table 1, you can see that both have exactly the same structure.

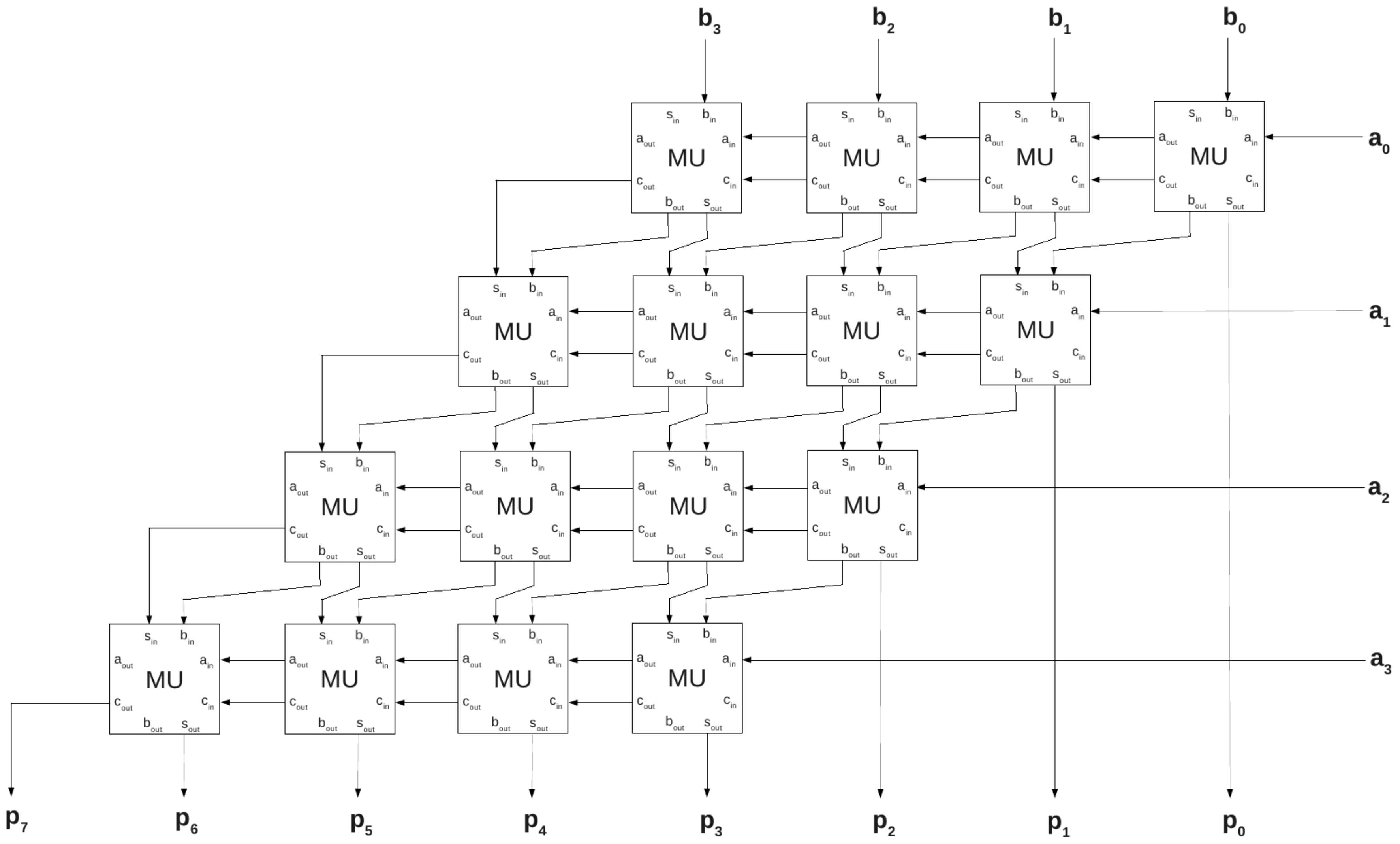

Figure 3: 4-bit multiplier schematic

Instead of instantiating and connecting each of these components separately, you should use the generate statement. This statement is particularly important for generating regular structures such as memories. Figure 4 shows a simple 4-bit shift register that is realized with the generate statement.

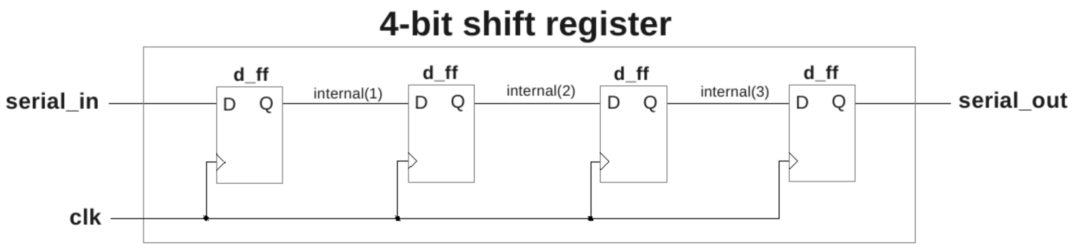

Figure 4: 4-bit shift register

Now have a look at the code we need to generate this shift register. It consists of 3 **if** (condition) **generate** statements within a **for** (loop variable) **generate** statement. The component d_ff is instantiated inside of the if statement and connected with signals depending on the loop variable.

```
shift_reg: for i in 4 downto 1 generate

-- connection of first flip-flop
reg_begin: if (i = 1) generate
               d_ff_begin: d_ff
                 port map (  d => serial_in,
                             clk => clk,
                             q => internal(i));
           end generate;

-- connection of middle flip-flops
reg_middle: if (i > 1) and (i < 4) generate
               d_ff_middle: d_ff
                  port map ( d => internal(i-1)
                             clk => clk,
                             q => internal(i));
            end generate;

-- connection of last flip-flop
reg_end: if (i = 4) generate
              d_ff_end: d_ff
                  port map ( d => internal(i-1),
                             clk => clk,
                             q => serial_out);
         end generate;

end generate;
```

The two main challenges when designing the 4 bit multiplier will be the differentiation between the various cases and the numbering of the interconnections required. Therefore, it is helpful to create a scheme of all interconnections before writing any code. Since the multiplier has a 2-dimensional structure, you must use a nested loop with different loop variables for the rows and columns.
Try to reduce the number of cases you have to consider. Nevertheless, you will have to use at least 6 different **if** statements inside the nested loop.

Furthermore, you cannot leave any inputs unconnected, so they must be set to '0'. Unconnected outputs only produce warnings during synthesis. To suppress these warnings, you can use the keyword **open** in the component instantiation.
The component mu is provided (you can find mu.vhd in the folder of this exercise), so you only have to declare and instantiate it.

When you have a look at the multiplier structure, you can see that it consists of a large amount of combinational logic. As you can see in Figure 5, the connection of different combinational modules may result in a very long critical path, which lowers the maximum usable frequency.

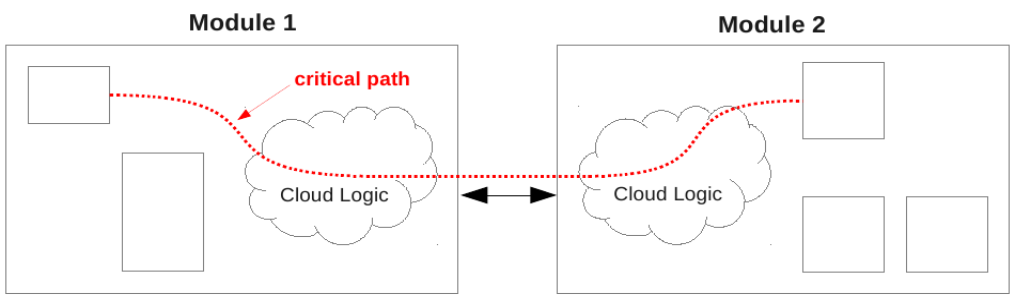

Figure 5: Critical path without I/O registers

To prevent this, it is good design practice to use registers for the inputs and outputs. This results in shorter critical paths, but you have to put up with additional cycles of latency (1 for every register inserted).

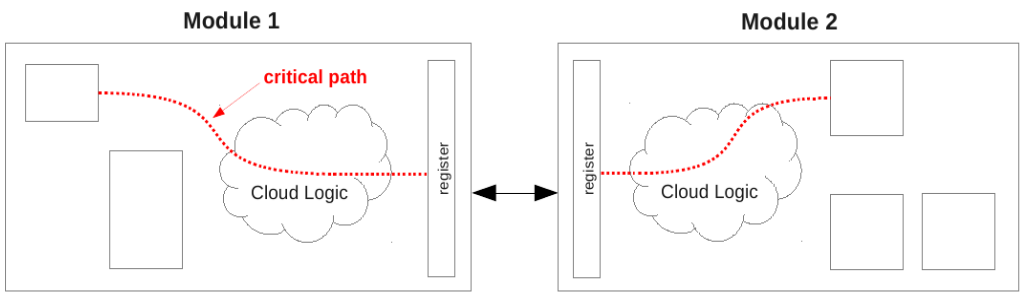

Figure 6: Critical path with I/O registers

Therefore, you should also use registers for the inputs and outputs of the multiplier.

The stimuli of the provided testbench tests all possible input combinations. The testbench is self-checking, which means that it recognizes each incorrect output value of your module. It expresses a warning in the ModelSim prompt every time you have an incorrect output value. This is done with an assertion and a severity warning in the VHDL code. Examine the testbench to see how that works.