Microelectronic Systems

Lab 2

Hierarchical arithmetical blocks

March 26, 2023

Introduction

During this laboratory experience you will learn how to generate and synthesize some more complex structures. Pay particular attention to the **synthesis scripts** and learn how to use them.

All the needed files are available from:

/home/repository/ms/cap2/

Generate a new folder, named **cap2**, in your home directory. Create inside it two more folders, **vhdlsim** and **syn**, and copy all the files from the repository into **vhdlsim**. Remember that the syntax to perform the operation is:

```
prompt> cp /home/repository/ms/cap2/* .
```

The lab is organized so that, for each defined block, you first simulate it and then you synthesize it to check the results of your design.

Suggestions:

* When you switch from a simulation phase to a synthesis one, copy only the VHDL entity files from **vhdlsim** directory to **syn**.
* Use two different workspaces in your system *(click on another square in the bottom right rectangle of your display)*, one for managing the simulations and the other for the synthesis.
* Before simulating, remember from Lab1 instructions, to set the simulator environment variables ***(setmentor)*** and to create a work library ***(vlib work)*** in the **vhdlsim** directory.
* In the same way, IN ANOTHER TERMINAL and in ANOTHER WORK SPACE, set the Synopsys' environment variables ***(setsynopsys)*** and create the work directory ***(mkdir work)*** in the **syn** folder. Copy also the file **.synopsys_dc.setup** in there.

2.1 Pentium 4 adder

As you know, the P4 adder is based on two substructures as shown in Figure 2.1: a **Carry Generator** and a **Sum Generator**. In the following, you are requested to build it block by block.

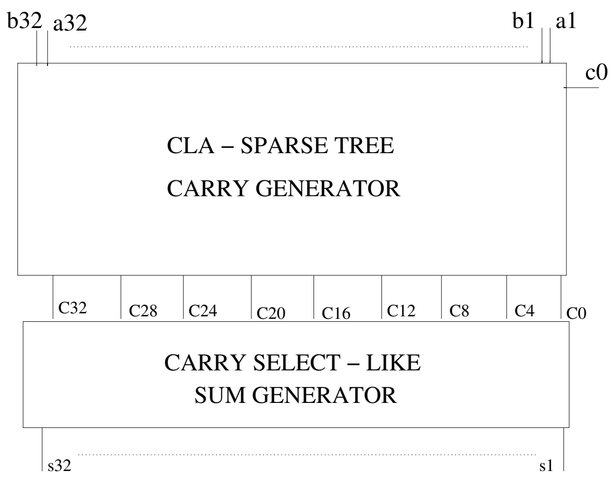

Figure 2.1: P4 Adder Complete Structure

A starting point: a given RCA

You will start with the design of the **Sum Generator block**, which is based on the *carry select principle*, made it easier by the absence of carry propagation. In Figure 2.2 you can observe a sketch of the whole structure.

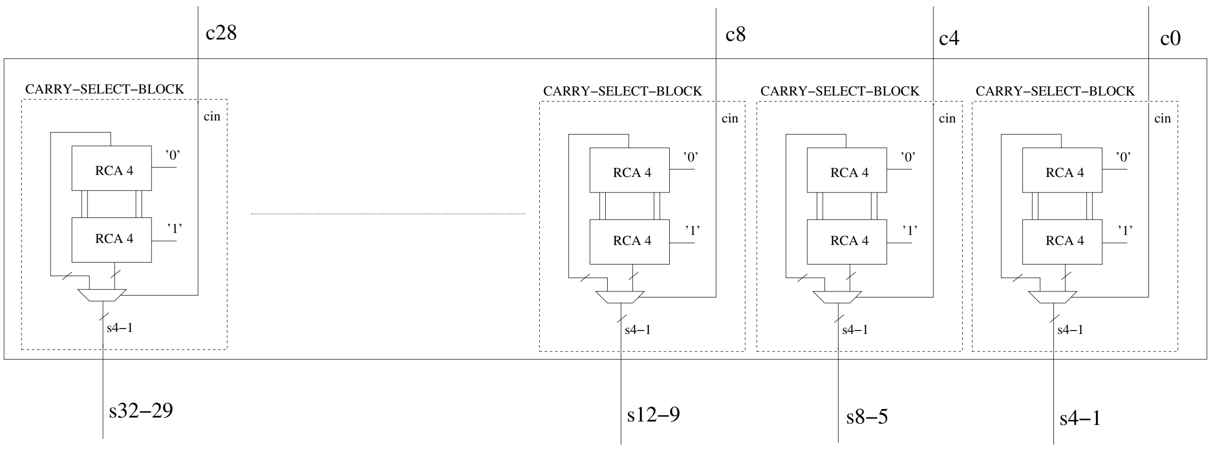

Figure 2.2: Carry Select Like structure

Your starting point is the Ripple Carry Adder you simulated in the Lab 1, therefore copy the files **fa.vhd**, **rca.vhd** from lab1 directory, together with **lfsr.vhd** and **tb_rca.vhd** for simulation purposes.

2.1.1 First step: Carry Select block

In order to define the Carry Select block, you have to proceed through the following steps:

1. Using the given Ripple Carry Adder, build a higher level block which implements a parametric CARRY SELECT BLOCK architecture (shown in Figure 2.2).
2. Use the RCA test bench as a starting point for simulating the correct behavior of the CARRY SELECT BLOCK, supposing to have a real **carry in**.

Summary of what is requested

Netlist of the Carry Select block based on the RCA structure.

2.1.2 Second step: Sum Generator

Starting from your Carry Select block, you can describe the *carry select-like* Sum Generator shown in Figure 2.2. It is suggested to:

- Use a generic organization in terms of number of bit and number of blocks.
- Use the **tb_sum_generator.vhd** testbench as a starting point for simulating the correct behaviour of the circuit, supposing to have the real carry in array.

Summary of what is requested

Netlist of the Sum Generator based on the Carry Select block.

2.1.3 Third step: Carry Generator

It is now the turn of the definition of the Carry Generator and, in particular, you have to refer to the structure reported in Figure 2.3.

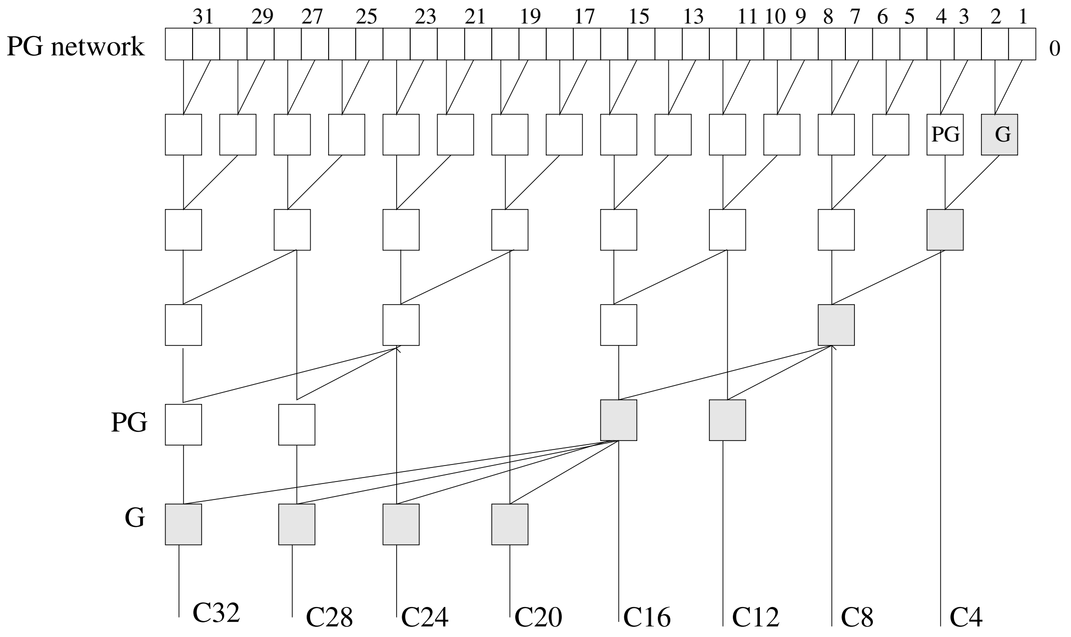

Figure 2.3: PG Network

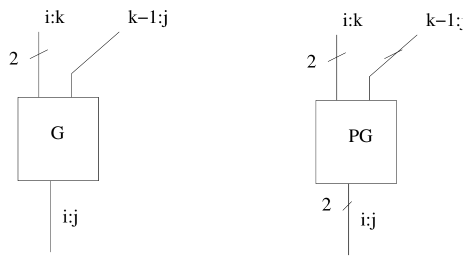

Figure 2.4: G and PG blocks

To perform this task, follow *carefully* the instructions below:

1. Describe a **parametric sparse tree look-ahead adder** based on the P4 structure we studied. Use a structural description for the tree structure, and a behavioral one for the elementary blocks.

2. Please, remember that:

   - The PG network generates the **propagate** and **generate** terms defined as:

   $$p_i = a_i \oplus b_i \qquad g_i = a_i \cdot b_i$$

   - The general **propagate** *(white box)* and general **generate** *(dark box)* super blocks, shown in details inside Figure 2.4, produce outputs as:

   $$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \qquad P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

   The PG blocks *(white box)* generates both $G_{i:j}$ and $P_{i:j}$, while the the G block *(dark box)* generates only $G_{i:j}$.

- Particular cases inside the array are represented by:

$$G_{x:x} = g_x \qquad P_{x:x} = p_x \qquad p_0 = 0 \qquad g_0 = C_{IN}$$

Suggestions:

- Remember that VHDL allows the definition of the **ARRAY type**. For example:

```
type SignalVector is array (N-1 downto 0) of std_logic_vector(N-1 downto 0);
```

In this way, a signal can be declared of type **SignalVector** and then used as follows:

```
signal tmpsign1: SignalVector;
signal tmpsign2: SignalVector;
signal tmpsign3: std_logic_vector (5 downto 0);

---- examples
tmpsign2 <= tmpsign1; --- assign a matrix to a matrix
tmpsign2(0) <= tmpsign1(5); --- assign a std_logic_vector to a  std_logic_vector
tmpsign2(0)(8) <= tmpsign1(4)(7); -- assign a bit to a bit
cicle:  for J in 0 to 3 generate
        tmpsign2(0)(J)<=tmpsign3(J+2);
end generate;  -- generic assignement with translation
```

- The **generate** command can be conditioned by **if** statements. For example:

```
righe: for riga in 0 to N-1 generate
     riga0:      if riga = 0 generate
                arrayand0: for colonna in 0 to N-1 generate
                        rigaand0: ARRAYAND2
                                generic map (0.1 ns)
                                port map (A(colonna), B(riga), OutFirstAnd(colonna));
                       and generate;
                end generate riga0;
and generate righe;
```

Summary of what is requested

Netlist of the sparse tree Carry Generator.

2.1.4 Fourth step: Complete P4 Adder

The two parts designed before are now ready to the assembly phase:

1. Connect the Sum Generator and the Carry Tree Generator.
2. Simulate the final circuit considering the test bench used for the RCA and for the Carry Select block, in a progressive way *(first for few bits and then for the whole extension of the input data)*.

Summary of what is requested

Netlist of the whole structure. Proof of the correct behavior with a waveform showing the correct carry generation in a critical case *(e.g 1111...111 + 0000...001)*.

2.1.5 Synthesis of the P4 Adder

Now, the aim is to synthesize your adder. Anyway, in this case, we want to start using the synthesizer in a **more clever way**. The steps you have to follow are detailed below.

Basic synthesis

1. Copy all the files useful for the complete description of the adder into the **syn** directory.
2. Analyze them from bottom up *(consider the hierarchy)*. Then, elaborate and compile the whole architecture of your P4 Adder.

Check and save the design

3. Explore the synthesized structure and check if it is consistent with what you expected.
4. Generate the related VHDL netlist and report both timing and area.
5. Go up to the top level *(use the up arrow)*. From this perspective, it is possible to analyze also the other paths, sorted by length. For example, the ten worst critical paths of the circuit can be obtained with the command:

   report_timing -nworst 10

   The same operation can be performed through the Graphical Interface: select from **Timing→Report Timing paths** and choose 10 in the “Worst path per endpoints”.

Question 1
Analyze the report: which are the differences among the paths?

Constrain the synthesis

6. Suppose that in the previous timing analysis you got *<MAX_PATH> ns* as maximum delay. Now, we want to run again the synthesis imposing a **lower timing value**. Let’s suppose we want it to achieve *<REQUIRED_TIME> ns*, that is, for example, a time 20% lower than *MAX_PATH*. In order to constrain with an upper bound the length of the critical path, type on Design Compiler’s command window:

   set_max_delay *<REQUIRED_TIME>* -from [all_inputs] -to [all_outputs]

   This procedure is used to force a *combinational maximum path delay*. During the next laboratory experiences, you will learn how to constrain a *clocked block*.
7. Now, run again the compilation:

   compile -map_effort high

   Here the synthesizer optimizes the compilation. Please notice that as your structure is not behavioral, the synthesizer has not many degrees of freedom to perform the operation. In the next exercise, the results will be better.
8. Upon completion, analyze the new timing performance:

   report_timing

   Then, save them:

   report_timing > p4add-timing-opt.txt

Question 2
Look at the differences: did something change? Display the critical path and compare it with the previous results. If there is any improvement, explain why.

9. At this point, use an interesting graphical feature: from the main menu select **Timing→Path Slack**. Leave the default settings and look at the results: a **path distribution** is being displayed. Click on the first histogram rectangle. The five worst slacks values are shown. Click on the second one: what's being displayed?

Using timing scripts

10. If you are sure you have saved everything you need, read carefully the script file **ADD_t.scr**, that you have inside your directory, and all the comments inside. Fill the row of the analyze command with the names of your files and impose the same timing constraint you have considered before.
11. Then, execute it in the command window typing:

    source "ADD_t.scr"

    See what's happening.

Question 3
Which are the differences with respect to the results obtained before?

12. Now, read the timing reports **ADD_timeopt_1t.rpt** and **ADD_timeopt_2t.rpt**, respectively generated before and after the second constrained compilation step. Note that inside the first report the critical path is said to be *unconstrained*, as you didn't use any maximum delay. Inside the second one, the critical path delay is *OPTIMIZED_TIME* ns, while the requested delay was *REQUIRED_TIME* ns. Therefore, the margin *(slack time)* is obtained as *REQUIRED_TIME* - *OPTIMIZED_TIME* ns. Please, note that if the slack is negative, then the constraint is not met.

Question 4
How was the synthesized managed to obtain a lower delay? Browse the design and see what has changed. Look at the saved VHDL netlist as well and see the used components.

⚠

Hereinafter, you are invited to use script files for rapidly synthesizing and forcing constraints on synthesis. Use the history window for observing the executed commands as a support *(you can also save the whole history inside a file and then you can modify it as needed, from time to time).*

⚠

You are also expected to consult the manual of the command terminal. For example, if you write on the command window *"man report_timing"* the manual page of the report_timing command will be displayed. Note that the command line has the auto completion, accessible via the TAB key.
Furthermore, from an external terminal you can use the command **"SOLD"** to get the whole Synopsys' documentation. Lastly, for more information about the synthesis process you can use both Design Vision and Design Compiler's manuals.

Summary of what is requested

Adder VHDL netlist, adder post synthesis VHDL netlist, area and timing report. Analysis of where the critical path is. Use a text file to describe your analyses.

2.2 Parallel multiplier based on Booth's algorithm

In this section, you will design, test and characterize a parallel multiplier implementing Booth's algorithm. The steps to follow are described in details below.

2.2.1 Design and simulate the multiplier

As a starting point, you have to:

1. Describe a N bit parallel multiplier using a *mixed structural and behavioral* architecture based on Booth's algorithm (see Figure 2.5).
2. Name the multiplier entity **"BOOTHMUL"**. If useful, re-use the components you have designed in the previous sections. If you decide to define a single architecture DO NOT CONSIDER the configuration feature. Choose a 32 bit implementation.
3. Simulate exhaustively your design through the provided test bench **tb_multiplier.vhd**. You have only to declare and instantiate your component in it.

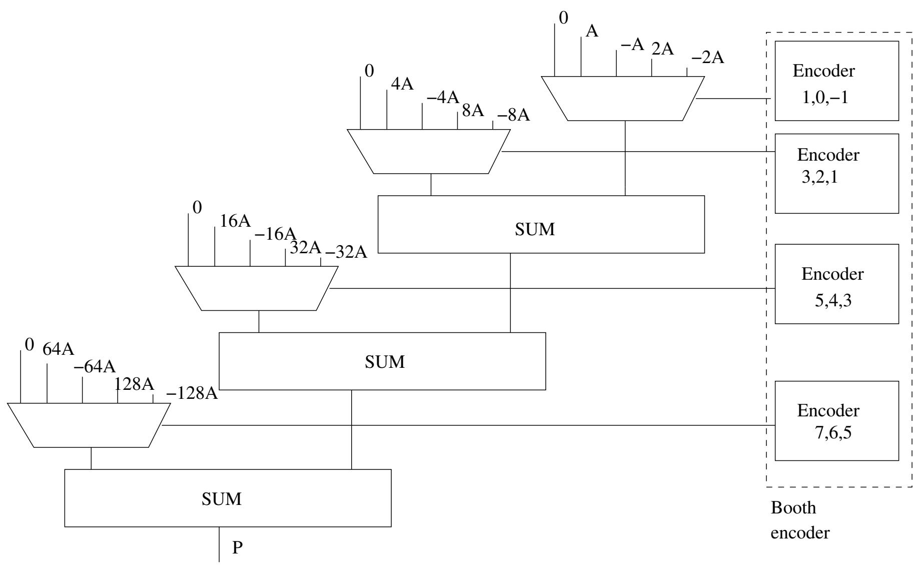

Figure 2.5: Structure of a 32 bit multiplier based on Booth's algorithm

⚠

Try to write a clean, generic and commented VHDL code.

Summary of what is requested

VHDL netlist of the Booth's multiplier and report of a meaningful waveform.

2.2.2 Synthesis

At this moment, you will synthesize the multiplier following the same steps you considered for the synthesis of the P4 Adder. Those instructions are here reported again, just for your convenience.

Basic synthesis

1. Copy all the files describing your multiplier into the **syn** directory. Analyze them from bottom up *(consider the hierarchy)*. Then, elaborate and compile your **"BOOTHMUL"** architecture and for a first check use a 8 bits implementation only.

Check and save

2. Explore the synthesized structure and check if it is consistent with what you expected.

3. Generate the associated VHDL netlist and report both timing and performance.

4. Go up to the top level *(use the up arrow)*. From here, report the ten worst critical paths, obtained through the following command:

   report_timing -nworst 10

Question 5
Analyze the report: which are the differences among the paths?

Higher number of bit and timing

5. Now, change the number of bits of the input data of the multiplier from 8 to 32. Analyze, elaborate and compile the whole design again, without imposing any constraint *(the results obtained here represent the starting point for further optimization)*.

6. Report the timing performance and save it on the text file **"mul-timing-no-opt.txt"**:

   report_timing > mul-timing-no-opt.txt

7. Do the same for the generation of the area report, considering a text file named **"mul-area-no-opt.txt"**:

   report_area > mul-area-no-opt.txt

Constrain the synthesis

8. Then, suppose that in the previous timing analysis you got *<MAX_PATH> ns* as maximum delay. You have to run again the synthesis process imposing a lower critical path. Suppose we consider *<REQUIRED_TIME> ns*, that is, for example, a time 20% lower than MAX_PATH. So, type in Design Vision's command window:

   set_max_delay REQUIRED_TIME -for [all_inputs] -to [all_outputs]

9. Run again the compilation process:

   compile -map_effort high

10. Once the optimization is complete, analyze the new timing report:

    report_timing

    Then, save it:

    report_timing > mul-timing-no-opt.txt

Question 6
Look at the differences: did something change? Display the critical path and compare it with the previous results. If there is any improvement, explain why.

11. Now, from the main menu select **Timing→Endopoint Slack**. Leave the default settings and see the results. Click on the first histogram rectangle and the five worst slacks are shown. Click on the second one: what's being displayed?
12. At the end, play with the time constraint and find the synthesizer limits

Using timing scripts

13. If you are sure you have saved everything you need, read carefully the script file *MUL_t.scr* that you have inside your directory, and all the comments inside. Fill the row of the analyze command with the names of your files and impose the same timing constraint you have considered before.
14. Then execute it in the command window typing:

    source "MUL_t.scr"

    See what's happening.

Question 7
Which are the differences with respect to the results obtained before?

15. Read the timing reports **rca_timeopt_1t.rpt** and **rca_timeopt_2t.rpt** obtained before and after the second constrained compilation step. Note again that inside the first report the critical path is said to be *unconstrained*, as you didn't use any maximum delay. Inside the second one, the critical path delay is *OPTIMIZED_TIME* ns, while the requested delay was *REQUIRED_TIME* ns. Therefore, the margin *(slack time)* is obtained as *REQUIRED_TIME* - *OPTIMIZED_TIME* ns. Please, note again that if the slack is negative, then the constraint is not met.

Question 8
How was the synthesized managed to obtain a lower delay? Browse the design and see what changed. Look at the saved VHDL netlist as well and see the used components.

Summary of what is requested

Synthesis script. Post-synthesis netlist of the 8 bits multiplier and related timing report. Timing and area reports of the 32 bits version, optimized and not.