CHAPTER 2

Hierarchical arithmetical blocks

In this lab 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.

The files for this lab are availabe from: **/home/repository/ms/cap2/** Generate a new directory **cap2** in your home. Create in it other two directories: **vhdlsim** and **syn** Copy all the files in directory vhdlsim: remember that the syntax is

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

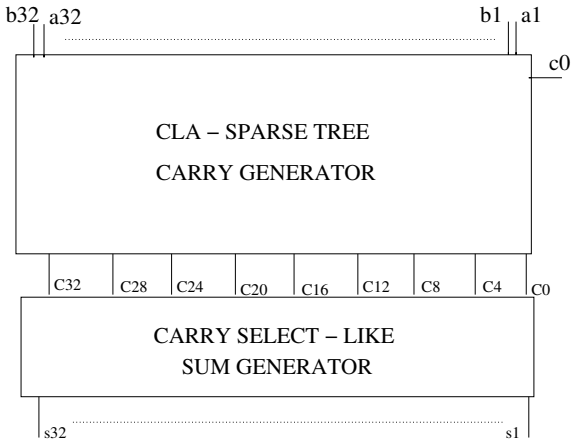
The lab is organized so that for each block you first simulate it and then you synthesize it to check the result of your code. When you switch from a simulation phase to a synthesis, copy the vhdl entity files from the **vhdlsim** to the **syn** directory as well.

It is suggested to use two different work spaces 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 (see cap1 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 (in another work space), set the synopsys environment variables (setsynopsys) and create the work directory (mkdir work) in the **syn** directory. Copy also the file .synopsys_dc.setup in there.

2.1 Pentium 4 adder

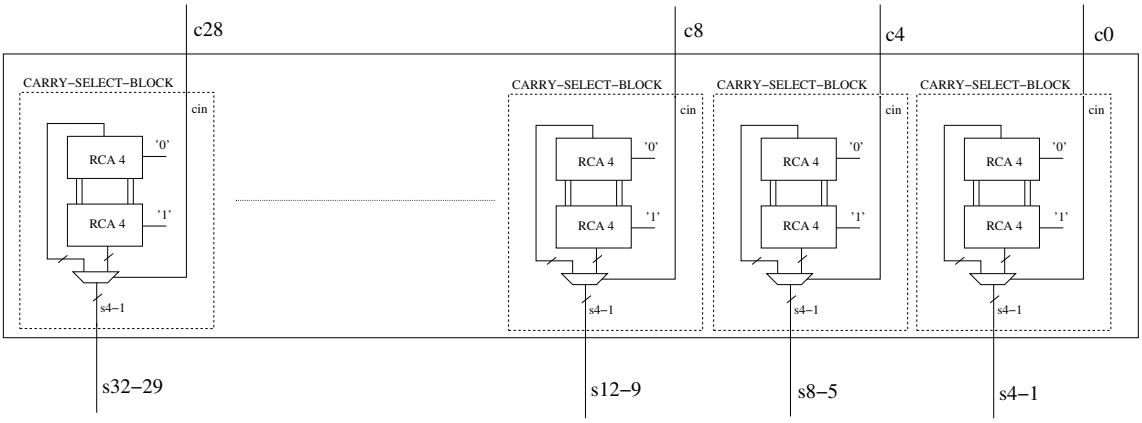
As you know the P4 adder is based on two substructures as shown in figure 2.1: a carry generator and a sum generator. In the following, then, you will build it block by block.



2.1.1 A starting point: a given RCA

You will start with the sum generator, which is based on the carry select principle, even though simpler (without carry propagation). In figure 2.1.1 you have the sketch of the whole structrure.

Your starting point is the Ripple Carry Adder you simulated in the lab 1. So copy files **ha.vhd**, **rca.vhd** (and for simulation **lfsr.vhd** and **tb_rca.vhd**) from lab1 directory.



2.1.2 Firt step: carry select

Using the give RCA adder build a higher level block which realizes a parametric CARRY SELECT BLOCK architecture as shown in figure 2.1.1. Use the RCA test bench for simulating the correct behavior supposing to have the real carry in.

Summary of what is requested

Netlist of the carry select structure based on the RCA structure.

2.1.3 Second step: sum generator

Now you can put the carry select block together in the sum generator of figure 2.1.1. Use a generic organization in terms of number of bit and number of blocks. Use the RCA test bench for simulating the correct behavior supposing to have the real carry in array.

Summary of what is requested

Netlist of the sum generator based on the carry select block.

2.1.4 Third step: the carry generator

Now describe a parametric sparse tree lookahead adder based on the P4 structure we studied in previous classes. The structure is reported in figure 2.1.4. Use a structural view for the tree structure, and a behavioral for the elementary blocks.

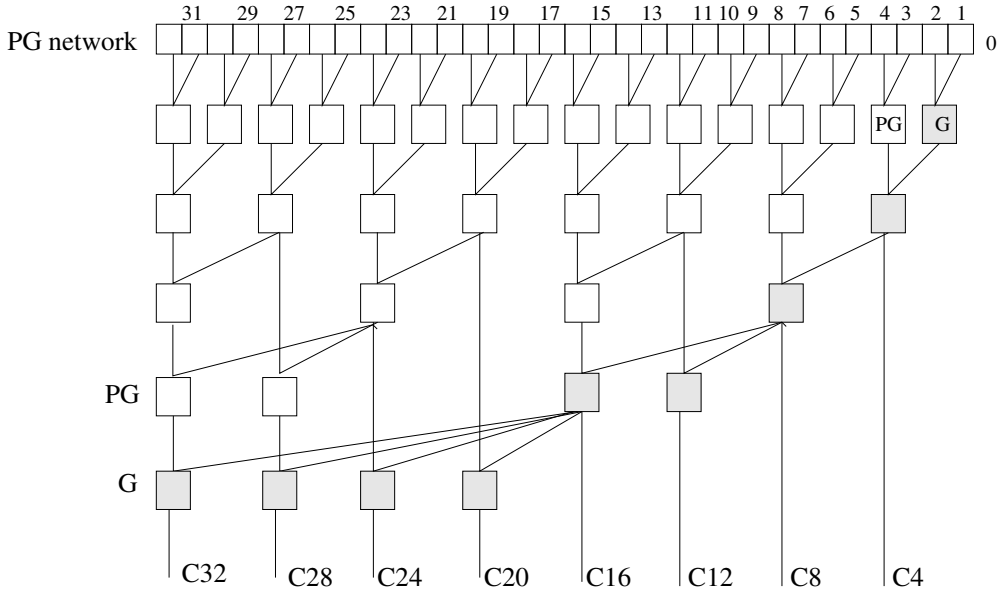
Remember that:

- 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 (shadowed box) superblocks generates outputs as

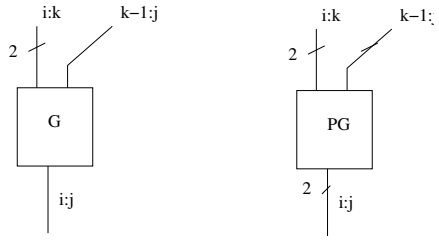
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \qquad P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$



as in figure 2.1.4 The PG blocks (white box) generates both $G_{i:j}$ and $P_{i:j}$, while the the G block (shadowed box) generates only $G_{i:j}$.

- Particular cases are:

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



SUGGESTION 1: 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);
```

so that a signal can be declared of type *SignalVector* and then used:

```
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
```

SUGGESTION 2: A further useful point is that 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.

2.1.5 Fourth step: the complete P4 adder

Connecting it all together: connect the sum generator and the carry tree generator.

Now you should simulate the two macro-blocks using the test bench used for the RCA and for the carry select before, at least for a few bits (four).
Finally you should checnk and simulate each sub-block and the whole architecture in case of 32 bits.

Summary of what is requested

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

2.1.6 Synthesis

The aim now is to synthesize your adder. Anyway in this case we want to start using the synthesizer in a more clever way.

Basic synthesis Copy all the adder files in the syn directory. Analyze them from bottom up (in terms of hierarchy level), elaborate and compile your "P4ADD" architecture.

Check and save Explore the structure and check if it is similar to what you expected.
Generate the extracted VHDL neltist and report both the timing performance. Notice where the critical path is placed using **Highlights→Critical Path** from the main menu.

Which is the critical path? Is it where you expected it?

Go up to the top level (up arrow). It is possible to see in order also the others paths; for example, the 10 worst C.P. can be obtained with the command:

report_timing -nworst 10

or select from the main many **Timing→Report Timing paths** and choose in the "Worst path per endpoints" 10.

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

Constraint the synthesis Suppose now that in previous timing analysis you got MAX_PATH ns as max delay. Now we want to run again the synthesis requiring a lower timing value. Let's sppose we want REQUIRED_TIME ns, that is, for example, a time 20% lower than MAX_PATH. Type in the command window:

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

This is used to force a combinational maximum path delay (in the next labs we will learn how to constraint a clocked block).
Now run again the compilation:

compile - map_effort high

Now the synthesizer is optimizing the compilation. Please notice that as your structure is not behavioral the synthesizer has not many degrees of freedom to optimize it (in the next exercise it will be better....)
When finished analyze the new timing performance:

report_timing

and save them:

report_timing > p4add-timing-opt.txt

Look at the differences: did something change? Display the critical path and compare it with previous results. In case the result is better, which is the cause of such improvement?

Now use this interesting graphical feature: from the main menu select **Timing→Endopoint Slack**. Leave the default settings and see the results: a path distribution is being displayed. Click on the first histogram rectangle. The five worst slacks are shown. Click on the second: what's being displayed?

Using timing scripts: If you are sure you saved all what you need, read carefully the script file *P4ADD_t.scr* you have in your directory with the comments. Fill the analyze row with your your file names and use the timing constraint you used before.

Then execute it in the command window typing:

source "P4ADD_t.scr"

and see what's happening.....! Which are the differences between the results obtained before? Now read the timing reports ADD_timeopt_1t.rpt and ADD_timeopt_2t.rpt obtained before and after the second contrained compilation step. Note that in the first report the path is said to be *unconstrained*, as we didn't use any maximum delay. In the second, the critical path delay is OPTIMIZED_TIME ns, the requested delay was REQUIRED_TIME ns, so the margin (slack) is REQUIRED_TIME - OPTIMIZED_TIME ns. If the slack is negatinve, then the constraint is not met. How has the synthesized managed to obtain a lower delay? Surf inside the design and see what changed. Look at the saved vhdl netlist as well and see the used components.

VERY IMPORTANT Hereinafter you are invited to use script files for rapidly synthesizing and forcing constraint on synthesis. Use the history window for help (you can also save the history in a file, that you can edit and change after).

MOST IMPORTANT Hereainafter it is expected that you use the commmand line help, e.g. if you write on the command window :: "man report_timing" the manual page will be displayed (note that the command line has the completion..).
Furthermore from an external terminal you can use the command "SOLD" to get the whole synopsys manual documentation. For synthesis you can use all the Design Vision and Design Compile manuals.

Summary of what is requested

Adder VHDL netlist, adder post synthesis VHDL netlist, area and timing report. Analisys of where the critical path is (comparison between the worst case delay of the carry generation and of the sum generation). Use a txt file to describe your analysis.

2.2 Parallel multiplier based on BOOTH's algorithm

2.2.1 Simulating multiplier based on BOOTH's algorithm

Describe a N bit parallel multiplier using a mixed structural and beavioral architecture based on BOOTH's algorithm (see figure 2.1). Call the multiplier entity "BOOTHMUL". Use the components available by previous sections if useful. If you use a single architecture do not use the configuration feature. Choose a 32 bit implementation.
A test bench is given in the file **tb_multiplier.vhd** for an exhaustive check. You only have to declare and instance your component in it.

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

Summary of what is requested

Multiplier vhdl netlist, a meaningful waveform.

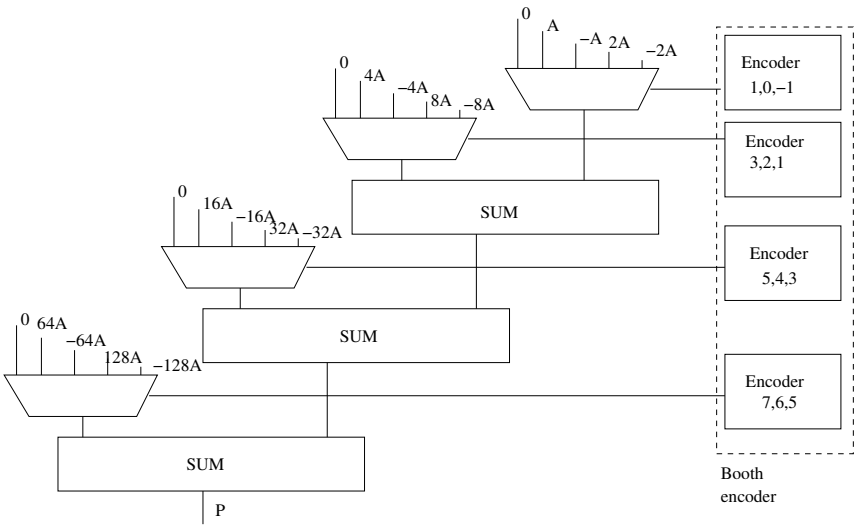


Figure 2.1:

2.2.2 Synthesis

The aim now is to synthesize your multiplier following the instructions you had for the ADDER (here repeated just for your convenience).

Basic synthesis Copy all the multiplier files in the syn directory. Analyze them from bottom up (in terms of hierarchy level), elaborate and compile your "BOOTHMUL" architecture. For a first check use a 8 bit implementation only.

Check and save Explore the structure and check if it is similar to what you expected.
Generate the extracted VHDL neltist (8 bit) and report both the 8 bit timing performance. Notice where the critical path is placed using **Highlights→Critical Path** from the main menu.

Go up to the top level (up arrow). It is possible to see in order also the others paths; for example, the 10 worst C.P. can be obtained with the command:

report_timing -nworst 10

or select from the main many **Timing→Report Timing paths** and choose in the "Worst path per endpoints" 10.

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

Higher number of bit and timing Now change the number of bits from 8 to 32 in the constant file. Analyze, elaborate and compile the block again without contraints (the results obtained here will be our starting point for further optimization). Report the timing performance and save on file "mul-timing-no-opt.txt":

report_timing > mul-timing-no-opt.txt

Do the same with the area report (reprot_area > mul-area-no-opt.txt).

Constraint the synthesis Suppose now that in previous timing analysis you got MAX_PATH ns as max delay. Now we want to run again the synthesis requiring a lower timing value. Let's sppose we want REQUIRED_TIME ns, that is, for example, a time 20% lower than MAX_PATH. Type in the command window:

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

This is used to force a combinational maximum path delay (in the next labs we will learn how to constraint a clocked block).
Now run again the compilation:

compile - map_effort high

Now the synthesizer is optimizing the compilation. When finished analyze the new timing performance:

report_timing

and save them:

report_timing > mul-timing-no-opt.txt

Look at the differences: did something change? Display the critical path and compare it with previous results. In case the result is better, which is the cause of such improvement?

Now use this interesting graphical feature: from the main menu select **Timing→Endopoint Slack**. Leave the default settings and see the results: a path distribution is being displayed. Click on the first histogram rectangle. The five worst slacks are shown. Click on the second: what's being displayed?

Now play with the time constraint and find the synthesizer limit...

Using timing scripts: If you are sure you saved all what you need, read carefully the script file *MUL_t.scr* you have in your directory with the comments. Fill the analyze row with your your file names and use the timing constraint you used before.

Then execute it in the command window typing:

source "MUL_t.scr"

and see what's happening.....! Which are the differences between the results obtained before? Now read the timing reports rca_timeopt_1t.rpt and rca_timeopt_2t.rpt obtained before and after the second contrained compilation step. Note that in the first report the path is said to be *unconstrained*, as we didn't use any maximum delay. In the second, the critical path delay is OPTIMIZED_TIME ns, the requested delay was REQUIRED_TIME ns, so the margin (slack) is REQUIRED_TIME - OPTIMIZED_TIME ns. If the slack is negatinve, then the constraint is not met. How has the synthesized managed to obtain a lower delay? Surf inside the design and see what changed. Look at the saved vhdl netlist as well and see the used components.

Summary of what is requested

Synthesized netlist (8 bit), timing report 8 bit, timing report 32 bit, optimized and not. Area report 32 bit, optimized and not. Completed synthesis script.