CHAPTER 6

Functional Verification

In this chapter you will work on functional verification of digital designs. It is a very important part of the design of complex digital circuits that can take up to 70% of time, while the design phase normally requires only 30% of the total design time. Functional verification can be seen as a more systematic way of testing digital circuits. From directory /home/repository/lowpower/ese6/ copy all files:

```
cp /home/repository/lowpower/ese6/* .
```

6.1 VHDL testing

In this section you will understabd how to do a simple functional verification of your circuit by using VHDL language.

6.1.1 A given RCA

Your first task will be the analysis of a given Ripple Carry Adder, a circuit that you already know, modified in order to introduce an error in the netlist. VHDL files are located in */ese6/1/*. Look at the testbench file (*rca_tb.vhd*) and note the three main differences with a standard testbench, key elements of the functional verification.

- **RANDOM INPUT VALUES.** From line 64 to 71 several instructions are used to generate two random standard logic vectors, they are fed to the inputs of the ripple carry adder. First random integer values are generated and then converted to standard logic vectors. In your opinion, why are random numbers used as inputs for the device under test (DUT)?
- **REFERENCE ARCHITECTURE.** The goal of functional verification is to verify that the circuit behaves as expected. To reach this goal its output must be compared with a reference architecture. This architecture must have the same expected behavior of the DUT and must not contains errors in the netlist. In this case the reference architecture is generated on line 75 of the testbench. A simple behavioral adder is used as reference.
- **ASSERT INSTRUCTIONS.** The *assert* instruction allows in VHDL language (line 78 of the testbench) to print a message when a determined condition is reached. In this case it is used to print an error message whenever the output of the DUT is different from the output of the reference architecture.

At industrial level there are of course dedicated tools to functional verification, but structuring a testbench with these three simple elements allows to setup a basic functional verification without the need of using such tools.

NOTE ON SCRIPTING. In a professional environment scripting will be an essential part of your job. It is necessary for the simple reason that allows you to save an enormous amount of time. In the context of functional verification scripting assumes a far bigger importance since it is normally required to run a huge number of simulations. The file *rca_tb.do* is an example of script for Modelsim that you can use for this exercise. So, before doing anything, look at the given script and understand it. Use therefore a script to run all the subsequent simulations (modify the file paths if they are on different folders).

Now that you have understood how to write a proper testbench, simulate the RCA using the given testbench. Does the simulation highlights the error in the RCA? If not extend the simulation by increasing the length of the *for loop* in the testbench. Try to simulate again the circuit, does it takes the same amount of iterations to find the error? If there is a difference, what do you think is the reason behind it?

Look at the RCA netlist (*rca.vhd*) and try to find the error in the code. Given that the RCA has a structural description you can use the simulation results looking at the output of each individual component to identify what part of the circuit does not behaves correctly. Now modify the netlist and the testbench increasing the number of bit of the RCA to 64 and then 128. Does it takes the same amount of time to find the error or do you have to extend the length of the testbench to find it?

Remark point

Concept of functional verification, scripting.

6.1.2 A more complex case

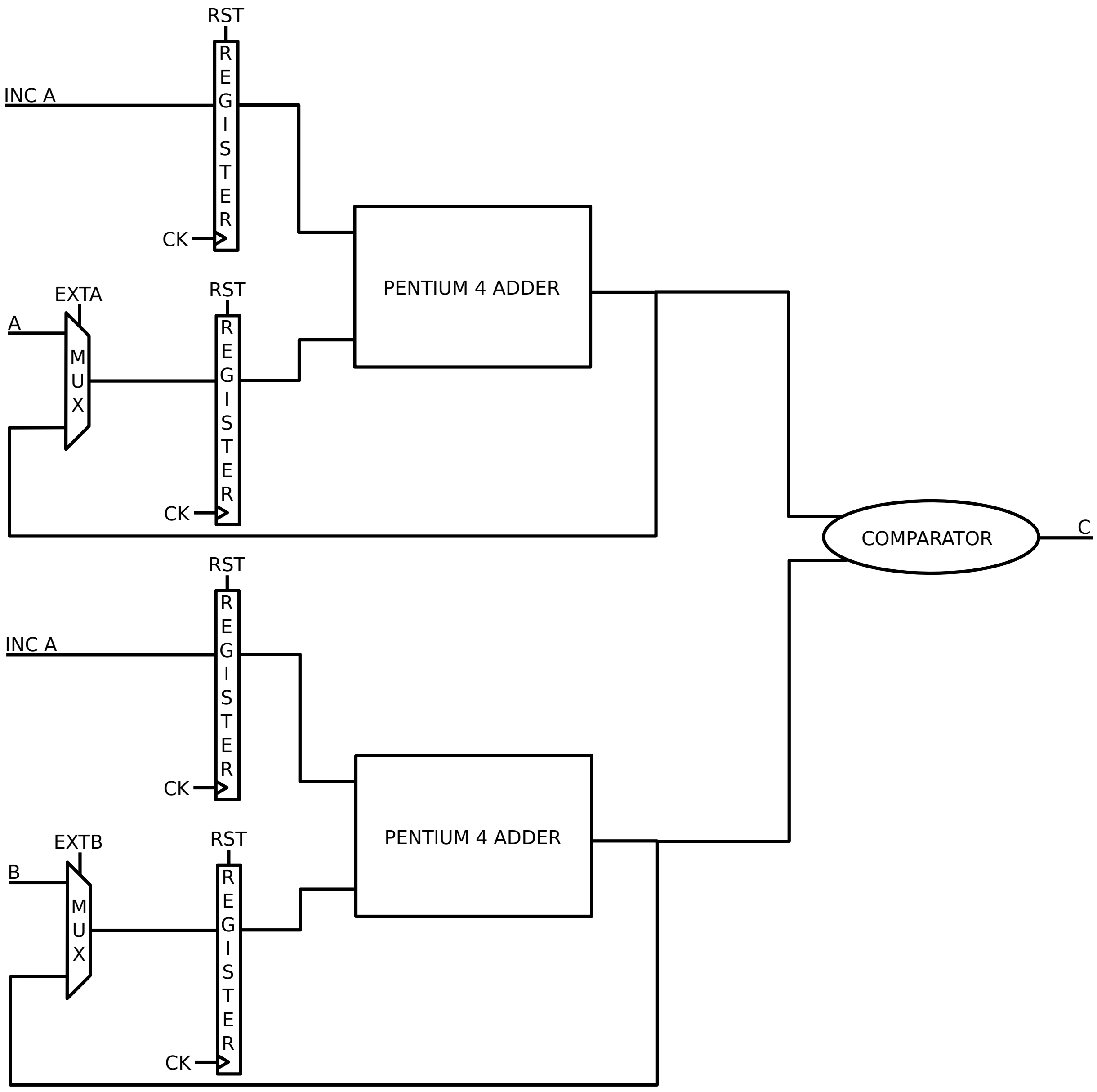
In the previous part you have analyzed a given RCA. A RCA is a very simple circuit and more importantly it is a combinational circuit. As a consequence it does not allow to appreciate why functional verification is so important in modern IC design. In this part you will analyze a more complex partially sequential circuit. It is a modified version of the incrementer&comparator circuit that you have seen in the *Clock Gating laboratory.* Its schematic is reported in figure 6.1.2.

The differences with the circuit that you have seen are two. First the incrementer part is implemented with a structural description using a particular type of adder, the Pentium 4 Adder. The P4 adder is a type of sparse tree adder used in the old P4 microprocessors. It represents a realistic and complex circuit that you can find inside modern high performance chips. Details on the internal structure of the P4 adder are provided in Section 6.3. The second difference is that two multiplexers are inserted on the adder inputs to allow the initialization of the counter to any desired value. This modification is required to properly test the circuit.

In the folder */ese6/2/* you will find 4 different versions of the circuit, from v1 to v4. Each of them contains a folder named *src* with the VHDL source files. The file *inccomp.vhd* contains the main entity, while the folders *P4_1* and *P4_2* contain the P4 adder files, please refer to Section 6.3 for the VHDL file structure of the P4 adder. The folder *utils* contains basic components of the P4 adder alongside with the multiplexers, registers and the comparator used in the main file *inccomp.vhd.*

You are required to:

- **CREATE A TESTBENCH** to simulate and to functionally verify the circuit. Follow the example of the previous exercise on how to structure the testbench using random numbers,



assert and a reference circuit. A bug free circuit that you can use as a reference is given inside the folder *vREF*.

- **CREATE A SIMULATION SCRIPT** to launch the simulation automatically. You can reuse the simulation script for all the four different versions.
- **IDENTIFY THE VERSIONS OF CIRCUIT WITH A BUG**. Only one of the four architectures is bug free.
- **IDENTIFY THE BUG** inside each version not working correctly. Navigate through the hierarchy inside the simulation window and identify the component not working properly. Then identify inside the netlist the error. Please refer to Section 6.3 for details on the P4 adder structure.

PLEASE NOTICE THAT this is not a challenge in terms of how good you are at finding the bug

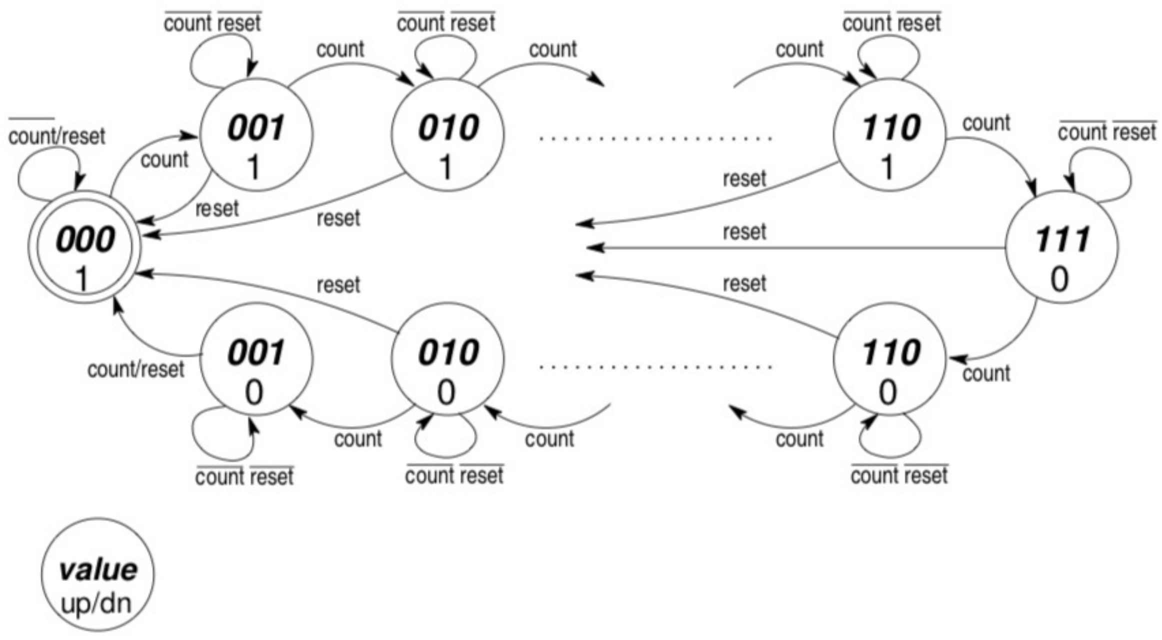
HERE in this context; this is an exarcise on what it might happen in a real life work situation where you are given a HW description that you don't know written by somebody else, you don't even have a correct version, you barely know what is the function, and you have to arrange an efficient way to find analyze the code and find the bug. So any method that is outide the set of strategies that you could use in that real-life situation is not a solution here.
Please refer to this notice when working and when documenting the steps you have adopted to reach the solution.

Remark point

How to write a testbench for functional verification, how to identify faulty components by analyzing the code and executing it in a clever way, how to read a code written by someone else.

6.1.3 Finite State Machine

The previous exercise was about a complex circuit mostly combinational. Finding a bug in a pure sequential circuit is more complex. In this part of the lab you will analyze a counter implemented as a finite state machine.



In the folder */ese6/3/* you will find 4 different versions of the counter. Similarly to the previous exercise, you are required to:

- **CREATE A TESTBENCH** to simulate and to functionally verify the circuit. Follow the example of the previous exercises on how to structure the testbench using random numbers, assert and a reference circuit. A bug free circuit that you can use as a reference is given inside the folder *vREF*.
- **CREATE A SIMULATION SCRIPT** to launch the simulation automatically. You can reuse the simulation script for all the four different versions.

- **IDENTIFY THE VERSIONS OF CIRCUIT WITH A BUG**. Only one of the four architectures is bug free. Does it take more or less iterations to find the bug compared to the circuit in the previous exercises?
- **IDENTIFY THE BUG** inside each version not working correctly. Navigate through the hierarchy inside the simulation window and identify the component not working properly. Then identify inside the netlist the error.

REPETITA IUVANT: PLEASE NOTICE THAT this is not a challenge in terms of how good you are at finding the bug HERE in this context; this is an exarcise on what it might happen in a real life work situation where you are given a HW description that you don't know written by somebody else, you don't even have a correct version, you barely know what is the function, and you have to arrange an efficient way to find analyze the code and find the bug. So any method that is outide the set of strategies that you could use in that real-life situation is not a solution here.
Please refer to this notice when working and when documenting the steps you have adopted to reach the solution.

Remark point

Differences between verifying a sequential circuit and a combinational one.

6.2 Scripting and Python: OPTIONAL

Since now we have seen how to properly test and verify circuits using VHDL language. It is a more detailed procedure than the simple creation of a testbench. The next step is to use a scripting language to automatically test and verify circuits. A common language used at industrial level for this purpose is Python.

PLEASE NOTE THAT this exercise is left OPTIONAL because of the different backgrounds of the students in this course. Of coursee if you are a student that has already attended a course where python is taught, it would be a pity not to use your skills at this point and not to experiment an exercise that you have been already working. So if you are in this condition you are strongly encouraged to work on it, and the results reported in the final report will be evaluated very well.

6.2.1 How to automatically create a VHDL testbench

A design rule that must be followed when complex circuits are designed is to test every single component as soon as it is created. However, creating a testbench for each component can be time consuming. A way to speed up the test phase is to use a script that automatically creates a testbench for you. Open with a text editor the file */ese6/4/TB_generator.py.*

It is a simple script that automatically creates a testbench for a circuit described in VHDL. By modifying the input parameters it is possible to automatically create a testbench for any circuit (WARNING!! The script is not perfect, it may not work for any circuit, particularly SignalsMode and SignalsType must be respectively in capital letters and in small letters. There may be other unknown issues...).

Your first task will be to read the script and to understand it. Even if you do not know Python, you should be able to understand it, the script is heavily commented. Try to modify it to test your ripple carry adder (you just have to change the input parameters at the beginning). Then run the script, create the testbench and simulate the ripple carry adder with Modelsim. To run the script simply open a terminal and type *python filename.py.*

Remark point

Basic understanding of Python language. Automatic generation of a testbench.

6.2.2 Reading inputs from file, writing outputs on a file

The next important step is the reading/writing to an external file. When testing a complex circuit you cannot write every input combination manually inside the testbench and look at the waveforms with your own eyes. Instead a more useful option is to read input patterns from an external file and to write output values on an external file, so that they can be easily post processed with dedicated and powerful tools.

Your task will be to create a modified version of the *TB_generator.py* script, so that the generated testbench is able to read data from a text file and to write the results on another file. By reading the *TB_generator.py* file, and by using your previous programming skills, you should have now understood how to do that in python.

!!IMPORTANT!!

Before starting to modify the python code try to understand how reading/writing to an external file in VHDL works. Take a look at the following code.

```
component ripple_carry_adder is
   .........
end component ripple_carry_adder;

signal .....

-- Definition of input/output file
file file_INPUTS: text;
file file_OUTPUTS: text;

begin

RCA: ripple_carry_adder
  generic map (
    .....
    );


process

  -- definition of variables used to store an entire line
  variable var_InputLINE: line;
  variable var_OutputLINE: line;

  -- definition of variables used to store input data
  variable var_TEMP1: std_logic_vector(N-1 downto 0);
  variable var_TEMP2: std_logic_vector(N-1 downto 0);
  variable var_TEMP3: std_logic_vector(N-1 downto 0);
```

```
    -- definition of variables used to store the separation character
    variable var_SEP: character;

  begin

    -- Opening input and output files in read/write modes
    file_open(file_INPUTS, "input_data.txt",  read_mode);
    file_open(file_OUTPUTS, "output_data.txt", write_mode);

    -- Read input stimuli from file
    -- Loop until the end of the file
    while not endfile(file_INPUTS) loop

      -- read a line from the file
      readline(file_INPUTS, var_InputLINE);
      -- get first value from the line
      read(var_InputLINE, var_TEMP1);
      -- read the separation character
      read(var_InputLINE, var_SEP);
      -- get the second value from the line
      read(var_InputLINE, var_TEMP2);
      -- repeat this last two instructions for every other data in the file

      -- Pass the variables to a signal to allow the circuit to use it
      (input signal of the circuit) <= var_TEMP1;
      (input signal of the circuit) <= var_TEMP2;
      -- Repeat for all the input signals

      -- Insert here a wait instruction (wait for a proper time)

      -- Write output result to file
      -- write single line
      write(var_OutputLINE, var_TEMP3, right, N);
      -- write line to file
      writeline(file_OUTPUTS, var_OutputLINE);
    end loop;

    -- Closin In/Out files
    file_close(file_INPUTS);
    file_close(file_OUTPUTS);
```

Now that you have understood how it is possible to interact with external files in VHDL, it is time to modify the script in order to create the testbench. You have to modify three sections of the script:

- **Libraries definition.** You have to add the following library: *use ieee.std_logic_textio.all; use STD.textio.all.*
- **Signals definition.** You have to add the definition of the files.
- **Generation of input signals.** You have to create the process described above.

The key function that you have to use in Python is the function that makes possible to write a string to a file with a specific format. Take a look at the following command for example.

TestbenchFile.write(" " + SignalsList[i] + " => " + SignalsList[i] + "_i,\n")

- TestbenchFile is the pointer to the file.
- Between the bracets there are the fixed strings that are chained to create the string that will be ultimately written to the file.
- Fixed strings are identified by " ".
- SignalList[i] identify a string that is part of a list.
- RandomInputBinary is a string created starting from a binary number.
- \n is the line termination character.
- + is used to chain strings.

Your task will be to create a modified version of the script that generates a testbench, (**ONLY**) for the ripple carry adder case, based on reading/writing data from files.
(**OPTIONAL.**) Create a version of the script that is able to generate a testbench compatible with all kind of circuits.

The format of the input/output file is the following:

- Every row represent a time step.
- For each row input signals are written in binary form separated by a space.

The script */ese6/5/generate_inputs.py* can be used to generate a custom input file to test your code. Run the script and see the format of the generated data input file.

Remark point

How to read/write from file in VHDL. How to write external files with Python.

6.2.3 Automatic verification through scripting

Now you have seen the basic structure of a python script used to automatize part of the design process, the creation of the testbench. You have also understood how to read inputs from a file and how to write the results on a file. This is an essential step for the testing phase. In this exercise your task will be to create a python script that will automatically launch the Modelsim simulation, collect the results from file and verify the correct behaviour of the circuit.

Have a look at the files in the folder */ese6/6*, specifically at the file *testing_script.py.* This is the template of the script that you have to create. It is divided in 5 different parts.

- **Part 1.** Generation of the input .txt file, containing the values that must be loaded at the input of your circuit under test.
- **Part 2.** Launch the Modelsim simulation. This part of the script is already provided to you, basically it sets up the Modelsim environment and it launches a .do script to compile the files. The compile.do script is also provided. Keep in mind that, while these scripts are given, they may require some adjustments if they are used with any generic circuit.

- **Part 3.** Calculation of the output values using python. The aim is to verify the output of the Modelsim simulation. In this part, supposing that you are testing an adder for example, you have to load the input values, sum them and generate the outputs, all this by using python.
- **Part 4.** Load the results of the Modelsim simulation from file.
- **Part 5.** Compare the value that you have calculated with python with the ones obtained from Modelsim, and verify that all of them are correct.

Complete the script, considering as testing circuit the RCA, then run it and verify the circuit under test. [**OPTIONAL**] Complete the script so that it works with any generic circuit.

To help you in this task two python scripts are already provided to you, *generate_inputs.py* that generates the input data file and *read_file.py* that reads values from a file. Copy and paste them into your code or, if you want, transform them into a function. The only part that are entire on your shoulders are **Part 3** and **Part 5**, which are the core part of the verification process.

Remark point

How to automatically verify a circuit using a script.

6.2.4 Further generalization

By doing these exercises you may have understood that, with few tweaks, you can generalize the script so that it works for any generic circuit, so, every time you create a new VHDL component you can verify it in one click. Now you should understand why scripting is so important for the verification process and how powerful it is.

There is however an important piece of the puzzle missing. The script that creates the VHDL testbench requires that you manually insert the input parameters. But it is possible to create a script that reads a generic VHDL code and parses the information automatically from the entity. So, if you want, try to create such a script. If you write it correctly you will never have to write again a testbench in your life (probably :-).

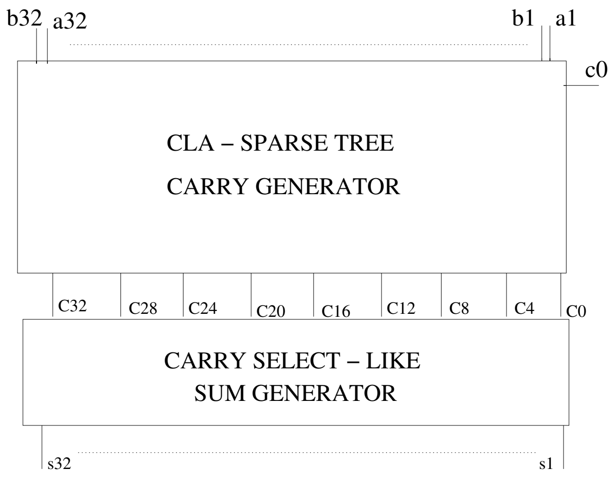
P.S. There are already libraries in python that are able to parse a VHDL file. You can use them in the future, but of course the goal of this exercise is that you create your own simple parser... otherwise you will learn nothing...

Remark point

How to parse a VHDL file/entity.

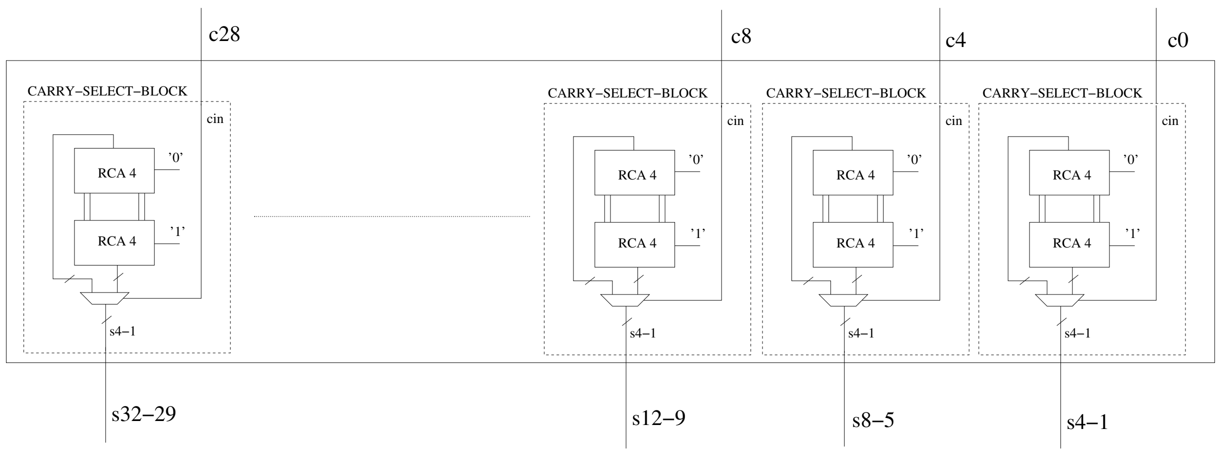
6.3 Appendix: Pentium 4 Adder

The P4 adder is based on two substructures as shown in figure 6.3: a carry generator and a sum generator.



6.3.1 Sum Generator

The sum generator is based on the carry select principle, even though simpler (without carry propagation). In figure 6.3.1 you have the sketch of the whole structrure. The sum generator is composed by several groups of carry select adders. Each of them is composed by two ripple carry adders and a multiplexer. The selection bit of the multiplexer is connected to the carry signals generated by the carry generator. While the structure can be completely generic in terms of bit number, in the Pentium 4 each carry select is indeed mapped on 4 bit.



6.3.2 Carry generator

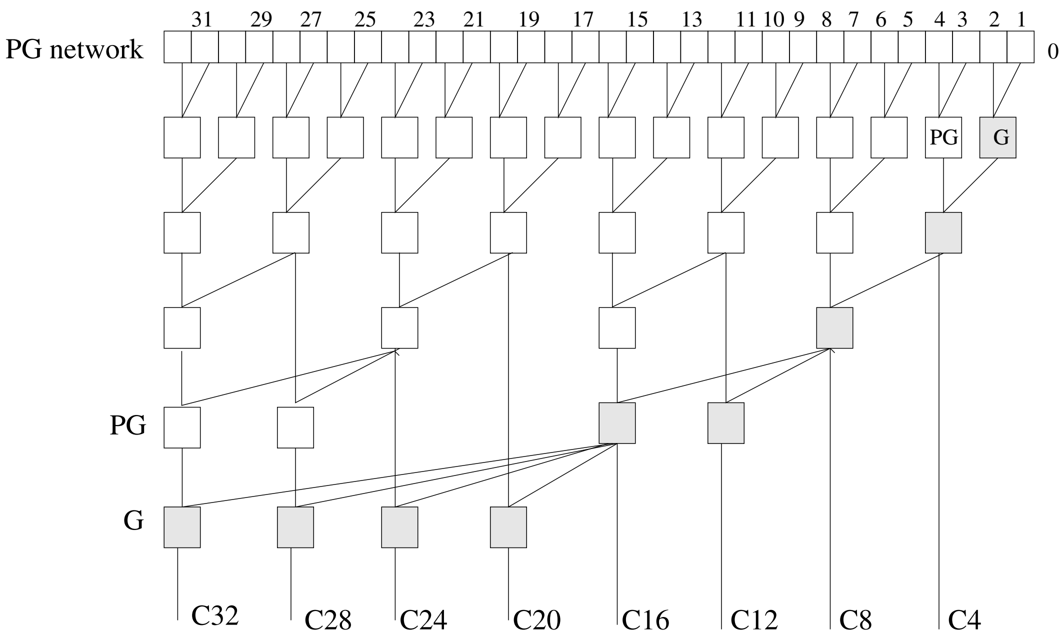
The carry generation network is a parametric sparse tree lookahead adder, a simplified and optimized version of the carry lookahead adder that generates only some of the carry signals. It represents a trade-off between occupied area and performance. The structure is reported in figure 6.3.2.

With:

- 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



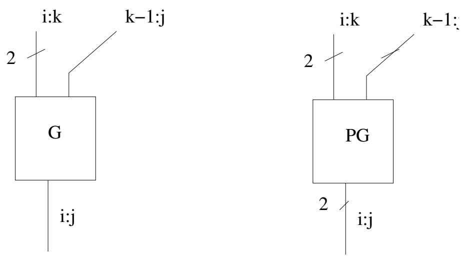
outputs as

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

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

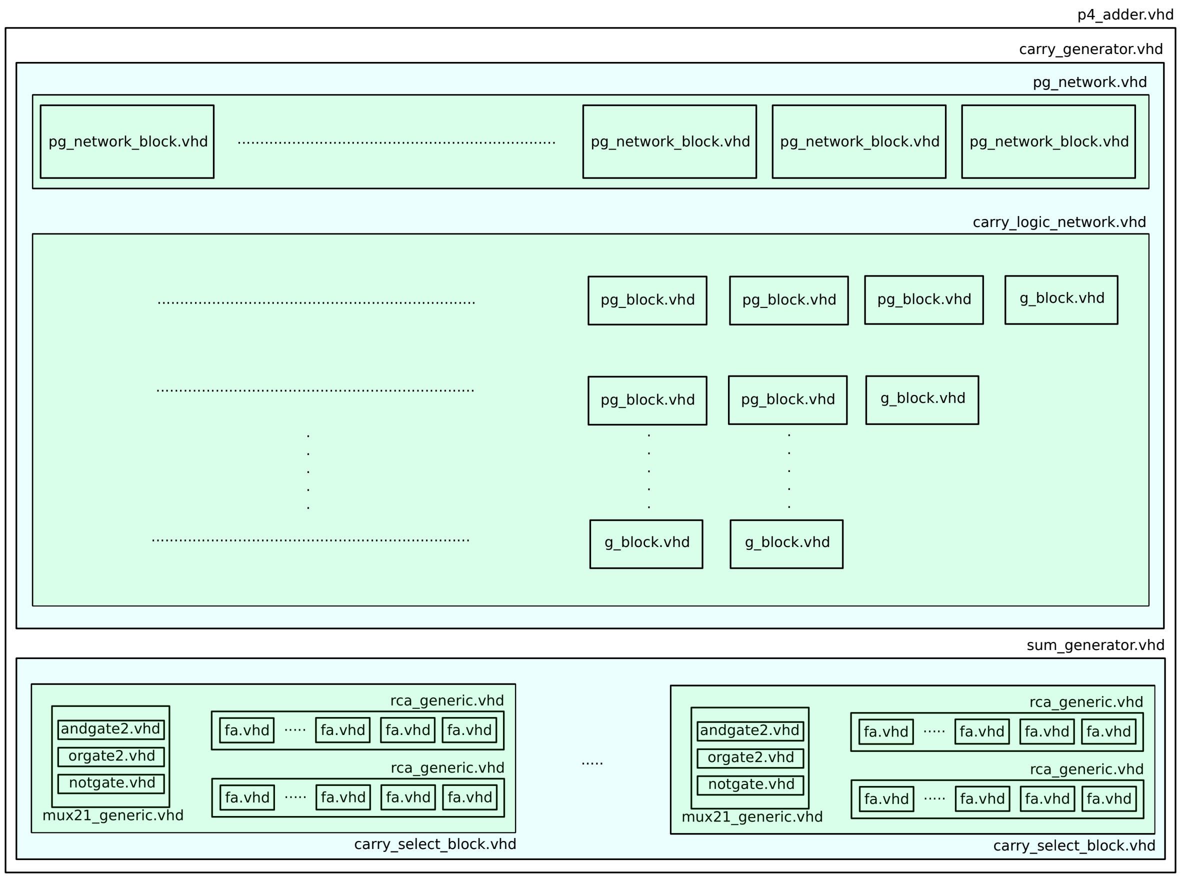
- Particular cases are:

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



6.3.3 VHDL file structure

The VHDL of the completely generic Pentium 4 adder is given. Figure 6.3.3 highlights the files structure.


p4_adder.vhd
carry_generator.vhd
pg_network.vhd
pg_network_block.vhd
carry_logic_network.vhd
pg_block.vhd
g_block.vhd
sum_generator.vhd
rca_generic.vhd
fa.vhd
andgate2.vhd
orgate2.vhd
notgate.vhd
mux21_generic.vhd
carry_select_block.vhd