Microelectronic Systems

Lab 3

Behavioral and structural sequential blocks

April 22, 2023

Introduction

In this laboratory experience you will learn how to generate and synthesize behavioral and sequential structures.

The files you need are available from:

/home/repository/ms/cap3/

Generate a new folder **cap3** 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/cap3/* .**

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.

During this laboratory session you will learn how to constrain your synthesis in terms of *frequency* and *power*, so please, perform carefully the synthesis phase.

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.

3.1 Behavioral Register File

The aim of this exercise is to describe two behavioral Register Files, one with and one without windowing, and to synthesize these architectures. The information you need to know are detailed in the following subsections.

3.1.1 RF design and simulation

The register file you have to design must fulfill the specification reported below:

- Integrate 32 registers
- Bit width equal to 64 bits
- 1 write port
- 2 read ports

- synchronous R/W on the rising edge of the clock signal, if the R1/R2/W signal is active *(high)*
- synchronous reset
- enable signal, active high
- simultaneous Read and Write capabilities

An example of the RF behavior is depicted in Figure 3.1.

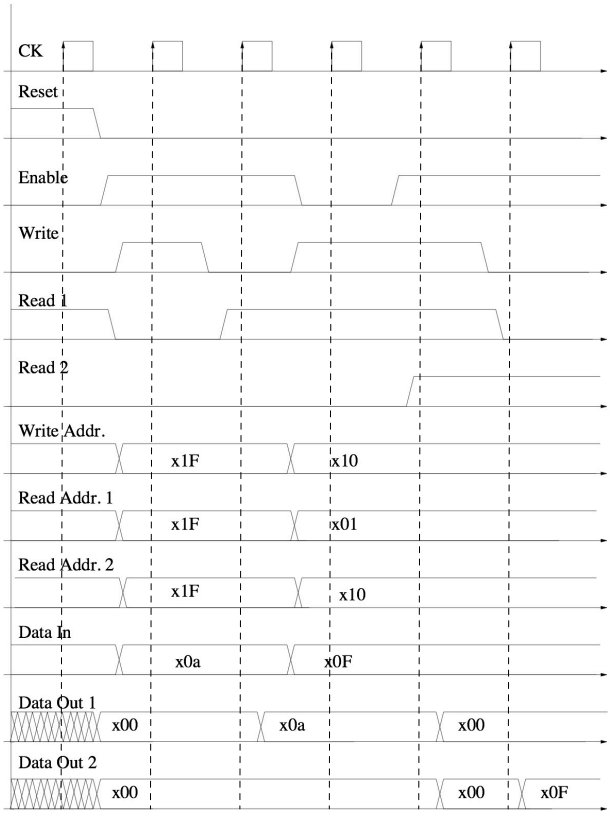

Figure 3.1: Behavior of the Register File that has to be designed.

In order to define your Register Files, follow the subsequent instructions:

1. Describe in VHDL the behavioral architecture of a Register File that respects the given specifications. Use **type** and **subtype** definitions and the **integer conversion**, as already suggested inside the file *registerfile.vhd* that you've already copied form the repository.
2. Simulate it using the given test-bench *(tb_registerfile.vhd)*.
3. Transform the Register File in a parametric form in terms of parallelism *(data and address)*.

Summary of what is requested

RF VHDL netlist with Parametric parallelism, meaningful simulation wave forms.

3.1.2 Synthesis: how to constraint a sequential block

In real life, Register Files are a particular kind of memory structures which cannot be synthesized on a standard cell library. However, as our architecture is not complex, we can synthesize it and analyze the results *(registers will be used as basic memory elements)*. To perform the operation, proceed as follows:

1. Synthesize a 32 bits RF considering a script *(please, remember the suggestions you received during Lab 2)* and analyze the results.

2. Generate the timing and area reports without constraints.

3. Now, we want to constrain the synthesis process. As the RF is a sequential block, we must define a **clock signal**. So, type the following command:

create_clock -name "CLK" -period 2 CLK

This forces a "CLK" generator at the input of your physical pin CLK *(identified by the second occurrence in the command)*.

Note that there must be exact correspondence between the name you declare for the CLK pin and that of the physical port in your VHDL architecture. Instead, the generator name can be any.

The default time unit is defined in the library and normally corresponds to *nanoseconds*.
If you want to check if the clock signal has been properly created, just type on the command window:

report_clock

4. Finally, compile the design:

compile

Question 1
Analyze the real timing report. Did something change? Do you understand the meaning of the reported values?

Note that thanks to the **create_clock** command you have only forced a *timing constraint* on the combinational path between two consecutive registers. As for example, in Figure 3.2 such a path is represented by *Block 4*.

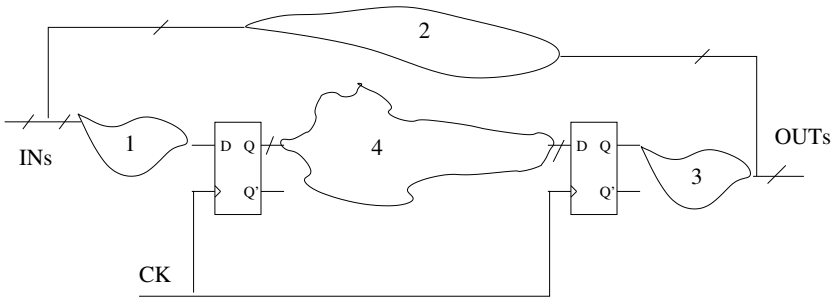

Figure 3.2: Combinational paths inside a sequential circuit.

Question 2
Consider again Figure 3.2, what happens to the combinational paths between the inputs INs and the first stage of registers *(paths in Block 1)*, or between the output registers and the outputs OUTs *(paths in block 3)*?

5. Now, you have also to limit the maximum length of these paths by typing the same command you have considered to constrain combinational circuits:

set_max_delay 2 -from [all_inputs] -to [all_outputs]

6. Compile again the whole architecture and analyze the new timing report. Try to understand from the obtained results where the critical path of your architecture is.

Suggestion:
Include all the useful commands in a script and try if you are able to obtain the same achievements without using the mouse. Real designers always use scripts and trace their work by commenting them.

7. At the end, generate the synthesized netlist.

Summary of what is requested

Final synthesis script, Non-Optimized area and timing report, Optimized area and timing reports, Optimized Post-synthesis netlist.

3.1.3 Windowed register file

In this section, you have to transform the simple Register File that you have previously designed allowing the operation of **context switching**, when a subroutine is called. In order to do that, use *three generics*:

- M for the number of **global registers**
- N for the number of **registers** in each IN, OUT or LOCAL window *(fixed window)*
- F for the number of **windows**

The structure is similar to the one depicted in Figure 3.3: on the right there is the structure of the physical Register File, while on the left you can observe the details of the part included in the active window.
Please, note that:

- Four registers might be necessary to transform the virtual Register File in the physical version and to manage the moment in which the Register File must SPILL/FILL to/from the memory without the need of excessive hardware. These registers are named **SWP**, **CWP**, **CANSAVE**, **CANRESTORE** and are used only internally.
- At least four further signals are necessary as I/O for the Register File: **CALL** and **RETURN** for the subroutine management, and **FILL** and **SPILL** when data are to be moved *from* and *to* the memory, respectively. Therefore, a BUS connecting the memory is needed and you have to define a proper signal inside your entity.

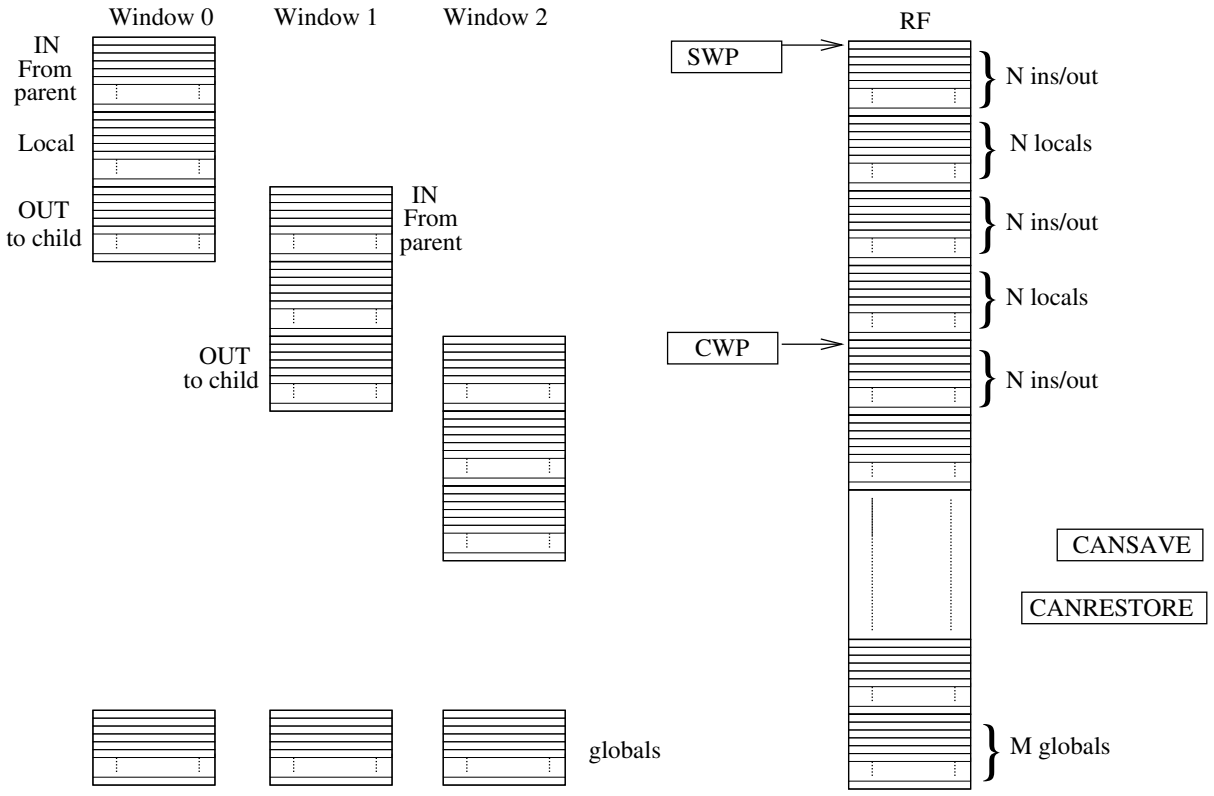

Figure 3.3: Register File with windowing

The expected behavior of this version of the Register File is described in the following.

Within a SUB For each subroutine the external blocks see only the active window. The whole active window is composed of the ensemble of the global registers, of one IN, one LOCAL and one OUT block of N registers. The fact that the data will be written in a IN or LOCAL, does not depend on the Register File control: it only receives an address related to that active window and it has to transform it in the actual address to the physical register file. Notice that from the external point of view the Register File is always *N*3+M* wide.

In order to transform the external address into a physical one, you can use a few registers in which saving the values of the pointers. The *Current Window Pointer (CWP)* holds the pointer to the IN block of the current window. During normal operations within a subroutine, the **CWP** is used to transform the external address to the physical address.

SUB call Let's now suppose that a subroutine is called. A signal **CALL** is risen, for example by the Control Unit or the decode stage: this could represent one input signal for your Register File. When a new subroutine is called *(CALL active)* the **CWP** is shifted by $2 \times N$ positions, so that the OUT of the current window becomes a IN for the next window. This can be done provided that new windows are available in the physical register. This information *(new windows are still available, if not see the SPILL paragraph below)* is stored in the **CANSAVE** register. You can decide how to use this register: the suggestion is to consider it as an aid to avoid the implementation of more complex hardware *(e.g. comparisons like CWP>3*N+M or similar are not exactly economic choices)*.

SUB return When the current subroutine gives a **RETURN**, then the **CWP** is shifted back of $2 \times N$ positions, provided that the window correspondent to the parent sub-directory is present in the Register File *(otherwise see FILL paragraph below)*. This information, i.e. if the parent is in the Register File, is stored in the **CANRESTORE** register.

SPILL In case no more registers are available, then a **SPILL** in memory must be performed for the oldest IN-LOCAL blocks. This operation means that the Register File rises the **SPILL** signal to an external block *(e.g. the Memory Management Unit)* and puts on the bus the data form the window that has to be spilled. Notice that this operation cannot be executed in one clock cycle: you must suppose to spill one register at each clock cycle. To do this you can use the **CWP** that must be correctly updated *(remember that this is used as a circular buffer)*. At the end of this operation the *SAVED WINDOW POINTER (SWP)* should hold the pointer to the Register File address *(or block number)* which is no more present in the Register File.

FILL In case there had previously been one or more SPILLs, followed by subsequent returns from subroutine, it happens sooner or later that the values spilled must be fed again in the Register File from the memory. This occurs when the **CWP**, that is being decremented as during a return phase, equalizes the **SWP**: this means that a further decrease of CWP must be preceded by a FILL. For this reason, the Register File rises the **FILL** command. Here you are free of deciding what will happen: either assume that the MMU always sends the sequence of data immediately, or that the MMU reacts and gives the data and an ack-signal, you should add to the I/O list, after a while. Once the FILL is concluded, also the CWP and the SWP should be updated.

Your job is to describe this organization, **considering only the operations that are strictly related to the Register File and its local control.** For example, in case a SUB or a RETURN instruction is to be executed, for the Register File not the whole execution of the instruction is important but only the related control signals generated by the Control Unit, which manages all the Register File operations. These control signals will activate a few management actions, local to the Register File and that you should provide.

At the end, write a test bench for the designed architecture, starting from the ones that you have previously considered in order to show its correct behavior in the event that a **SUB** or a **RETURN** instruction is received.

You should take into account the SPILL and FILL occurrences, but the operations of **writing and reading to/from memory is not your concern**: just define a way to give the Register File the proper inputs. Finally, you are not required to design any controller: define the few control signals you may need.

Summary of what is requested

Fully commented VHDL netlist of the Windowed Register File, test bench, meaningful wave forms.

3.1.4 Synthesis

In order to complete the characterization of the designed architecture, you have to proceed with the synthesis process. Hence, synthesize your Windowed Register File and check the derived results.

Question 3

How does the timing report change with respect to the one obtained from the previous structure?

Summary of what is requested

Post-synthesis netlist, timing report.

3.1.5 Synthesis of a SI-PI-SO-ALU

In this section you can *relax* a little bit: just look, execute and especially **understand** the way you are using the synthesizer. The aim is to have a look to a particular structure and to use it as a benchmark to learn how to force a **low-power synthesis**. Consider the sketch in Figure 3.4.

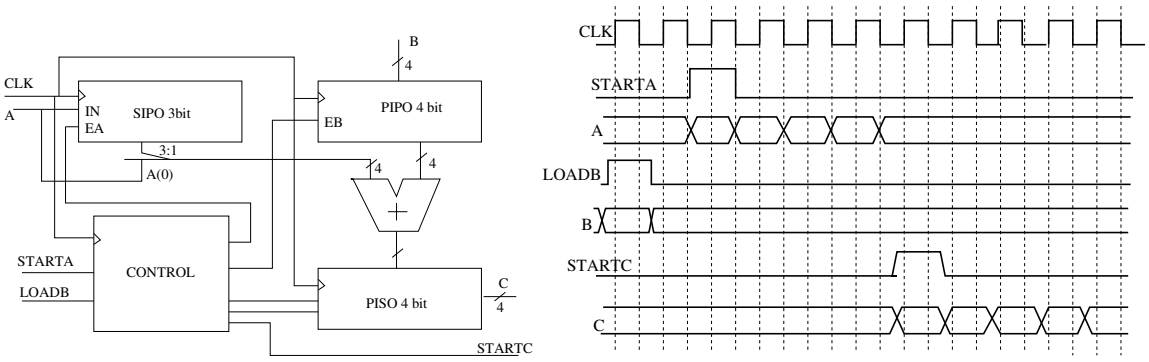

Figure 3.4: *On the left:* Structure of the Serial-in/Parallel-in Parallel-out Arithmetic and Logic Unit; *on the right:* Expected wave forms at the output of the ALU

Question 4
What do you think it implements in detail?

When you are ready to start, consider the following operations:

1. Copy the VHDL code **sipisoalu.vhd**, which describes both the registers and the ALU in a behavioral style.
2. Put yourself in *"reverse engineering mode"* and analyze the code. You should be able to understand how the behavioral *parallel to serial* and *serial to parallel* implementation is described. This is useful for now and the next laboratory experiences.

3.1.6 Synthesis: how to analyze and constraint the power consumption

In the following, you can find the instructions to proceed with a synthesis process with the awareness of the power consumption of the designed circuit:

1. As you always do, copy the **sipisoalu.vhd** into the *syn* directory.
2. Synthesize the sipisoalu architecture *without constraints* and check how your FSM has been implemented.
3. Perform the timing analysis. Let's call MAX_PATH the maximum delay found.
4. Let's take a look at the power dissipation by executing the command:

   report_power

   You can also perform the same operation via the main menu: **Design→Report Power** and leaving the default configurations *(save report power file)*.

Question 5
The report that you have here is general for the whole SIPISOALU and you can distinguish three contributions: do you understand the source of these power consumption?

5. You can obtain more detailed information by selecting in the displayed window from **Design→Report Power**, not the term *"summary only"*, but the *"cells only"* one: now, you get the contribution of each cell in the circuit. This action corresponds to the command:

   report_power -cell

   Note the difference among the *internal switching power*, the *driven net switching power* and the *cell leakage power*. You can get an additional information about the power consumption, the *toggle count of each net*. In order to obtain the related report, just type:

   report_power -net

   Again, the same operation can be performed via the Graphical Interface and, in particular, via the **Design→Report Power** window where you have to select *"nets only"*. The toggle activity of each net will be displayed. Note that if you don't change the default values, a 0.5 toggle probability is assumed for the inputs. The *switching activity* of the internal nodes are evaluated by Design Compiler accordingly[1].

Notice that the power is a function of frequency, that is of period:

$$P \approx V_{dd}^2 C_{load} f \alpha$$

where α is the switching activity of the nodes (toggle count).

Question 6

Which is the period used for the power computation here, given that we did not defined any clock signal? The period assumed by default is defined inside the technology library. We can change it and see what happens to the power report. To which value? We can use for example exactly the worst critical path as a limit clock (MAX_PATH).

6. Let's create the clock signal: as the name *"CLK"* has been considered for the identification of the clock signal in the VHDL file[2], you must now define the clock waveform *(as you did before)* referring to it. This can be easily accomplished by the command window, in which you have to type:

   create_clock -name "CLK" -period MAX_PATH CLK

Notice that in case you are working on a combinational circuit, this clock signal is only virtual and the second CLK occurrence in the command line shouldn't be present. In any case, such a virtual clock must be defined for the computation of power consumption, otherwise a default frequency would be used.

[1] This aspect will be clarified during the next lessons. In particular, different methods will be discussed for importing the *real switching activity* of a circuit through the process of back-annotation.

[2] Also if you don't have an actual clock signal in your design, the same procedure can be used to define a virtual one

7. Now, you should note that the power is a little bit different than before, by executing the following command and observing the results:

report_power

8. Finally, consider the given simple part of a script, **sipisoalu_pw.scr**. Take a look at the general use of the variable *"Period"*, defined at the beginning. You can execute it row after row, copying and pasting the associated commands, to see what happens. Pay attention to the use of the *power constraint* and substitute the value of 100 uW with a lower one, taking into account the value you have obtained in the previous analysis:

set_max_dynamic_power 100 uW

Question 7

Compile again and analyze both the new power and timing reports: you should expect that if you force a lower power limit, the maximum delay within the circuit will increase... do you know why?

9. As a final step, play with the constraints to find the limits of the compiler and save the timing and power results, as suggested in the given script.

Summary of what is requested

Final synthesis script, Post-synthesis netlist, initial and final power and timing reports.