CHAPTER 3

Behavioral and structural sequential blocks

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

The files for this lab are availabe from: **/home/repository/ms/cap3/** Generate a new directory **cap3** 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/cap3/* .**

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. **Important: in this lab you will learn how to constraint you synthesis in terms of frequency and power, so please carefully perform the synthesis phase.** When you switch from a simulation phase to a synthesis, copy the vhdl entity files from the **vhdlsim** to the **syn** directory.

Again 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 the syn directory (not in the work!).

3.1 Register File Behavioral

The aim is to generate the description of a behavioral RF, both with and without windowing and to synthesize it.

3.1.1 RF simulation

You have a behavioral VHDL netlist describing a Register File entity (netlist *registerfile.vhd* and test bench tb_registerfile.vhd) with the following specifications:

- 32 registers
- bitwidth = 64 bit
- 1 write port
- 2 read ports
- synchronous R/W on the clock rising edge if R1/R2/W signal active (high)

- synchronous reset
- enable signal active high
- simultaneous Read and Write capabilities

An example of the RF behavior is in figure 3.3.

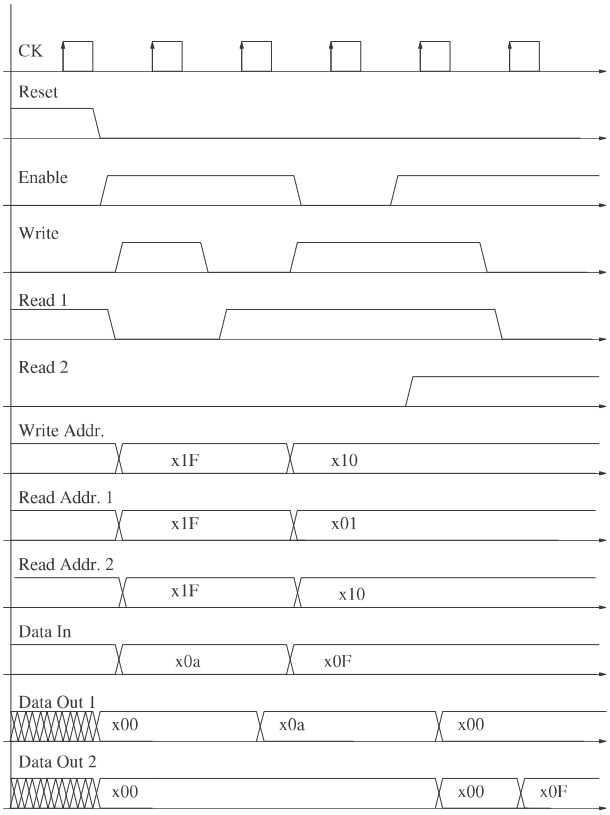

Figure 3.1:

Write the VHDL architecture using a behavioral description following the given specifications and the use of type and subtype definitions and of integer conversion already suggested in the architecture. Simulate it using the given test bench.

Now transform the RF in a parametric RF in terms of bit number (data and address).

Summary of what is requested

RF VHDL netlist with parametric bit number, meaningful imulation waveforms.

3.1.2 Synthesis: how to contraint a sequential block

In real life, RF are particular kind of memory structures and cannot be synthesized on a standard cell library. However, as our one is not complex, we can synthesize it and analyze the results (registers will be used!).
Synthesize thus a 32 entry RF using a script (remember the suggestions you received during lab 2) and analyze the result. Generate the timing and area report without constraints.

As a second step we want to contraint the synthesis. As this is a sequential block, we must define a clock signal. Type the following commands:

create_clock -name "CLK" -period 2 CLK

which says that you are forcing a "CLK" generator to your physical pin CLK (the second occurence in the the command; note this one must be exactly your physical port in VHDL, while the generator name can be any). The default time unit is defined in the library and normally is nanosecond. If you want to check if the clock has been created just type on the command window bottom bar:

report_clock

Now compile the design:

compile

and analyze the real timing report! Did something changed? Do you understand the meaning of the reported values?

Note that thanks to the create_clock command we only forced a timing constraint on the path between two register, as for example in figure 3.2, paths present in block 4. What happens to combinational paths between general inputs and registers (paths in block 1) or between output registers and general outputs (paths in block 3)?

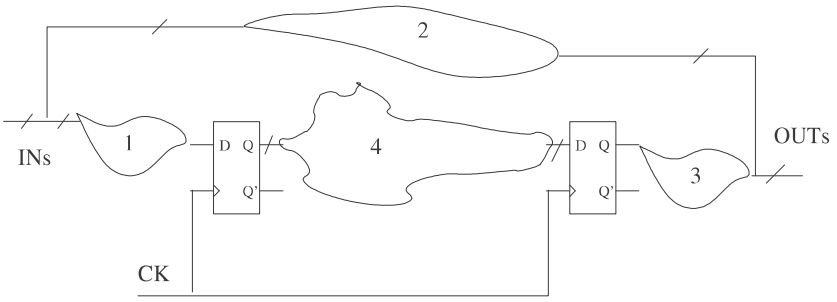

Figure 3.2:

We must also force these paths typing the same command we used for the combinational circuits:

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

Compile again the structure and analyze the new timing report. Search the critical path (the action is: highlight critical path). Include all these commands in a script and try if it works without using the mouse! (real designers always use scripts and trace their work by commenting them).
At the end, generate the synthesized netlist.

Summary of what is requested

Final synthesis script, timing report, synthesized VHDL netlist.

3.1.3 Windowed register file

Now you should transform your smple RF in a structure that allows context switching when a subroutine is called. Use three generics:

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

The structure is similar to the one in figure 3.3: on the right is the structure of the physical RF and on the left the part of the physical RF that are included in the active window.

Four registers might be necessary to transform the virtual in the physical RF and to manage the moment in which the RF must SPILL/FILL to/from memory without the need of eccessive HW. These registers are SWP, CWP, CANSAVE, CANRESTORE and are used only internally.

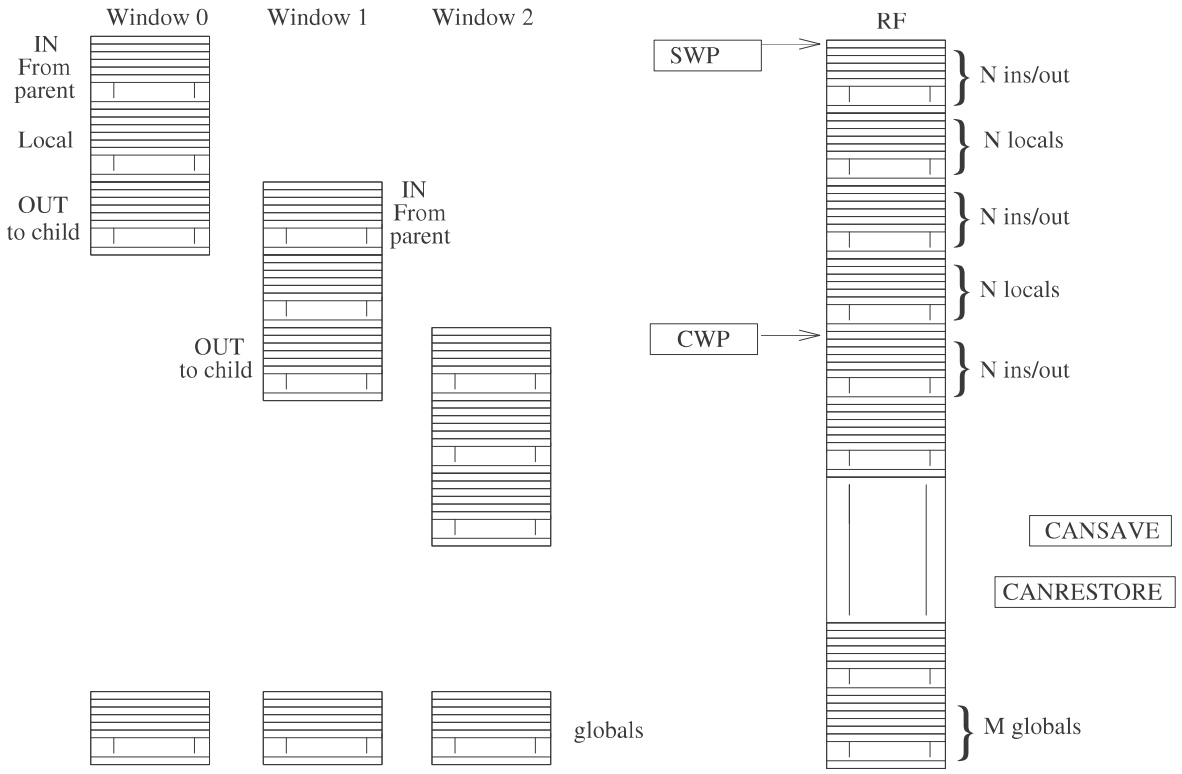

Figure 3.3:

Four further signals at least are necessary as I/O of you RF: CALL and RETURN for subroutine management, and FILL and SPILL when data are to be moved from and to the memory respectively. This also means that a BUS to/from memory is needed as a signal in your entity.

The expected behavior is in the following.

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

In order to transform the external address to the physical address you can use a few registers which save the values of 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 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 decode stage: you can suppose this is one input signal for your RF. When a new subroutine is called (CALL active) the CWP is shifted by $2 \times N$ positions so that OUT of current window becomes a IN of 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 SPILL below) is stored in the CANSAVE register. You can decide how to use this register: the suggestion is that it helps you avoiding to use complex HW (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 subdirectory is present in the

register file (otherwise see FILL below). This information (parent is in RF) 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 RF rises the SPILL signal to an external block (e.g. the MMU) the reads the bus where the RF puts the data in the window 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 RF address (or block number) which is no more in the RF.

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

Jour job is to describe this organization, considering only the operations strictly related to the RF and its local control. For example in case a SUB or a RETURN instruction are to be executed, from the RF point of view not the whole instruction execution is important but only the correspondent control signals from the control unit which manages the RF operations. Such control signals will activate a few management actions local to the RF tha you should provide.

Write a test bench starting from the previously used ones to show the correct behavior in case a SUB or a RETURN instructions. You should also take into account SPILL and FILL occurrences, but writing and reading to/from memory is not your concern: just define a way to give to RF output or recevie from RF input a sequence of registers. Finally, you are not writing any controller here: just suppose to have as input or give as output a few control signals you may need.

Summary of what is requested

Windowed register file VHDL netlist fully commented (!!!!) and test bench, meaningful waveforms.

3.1.4 Synthesis

Synthesize your WRF and check the correct implementation. How does the timing report change with respect to the previous structure?

Summary of what is requested

Synthesized VHDL netlist, timing report.

3.1.5 Sythesis of a SI-PI-SO-ALU

Here you can relax a little bit: just look, execute and especially understand how you are using the synthesizer. Consider the sketch in figure 7.11. The aim is to have alokk to a particular structure and to use it as a benchmark to learn how to force a lowpower synthesis.

Consider the sketch in figure 7.11.

What do you think it performs (in details)? Copy the vhdl code *sipisoalu.vhd* which describes the registers and the alu in a behavioral style. Configure yourself in "reverse engineering mode" and analyze the code, in particular you should be able to understand how the behavioral parallel to serial and serial to parallel implementation is described (it could be useful now and then).

Figure 7.11 on the right shows the expected waveforms at the ALU output.

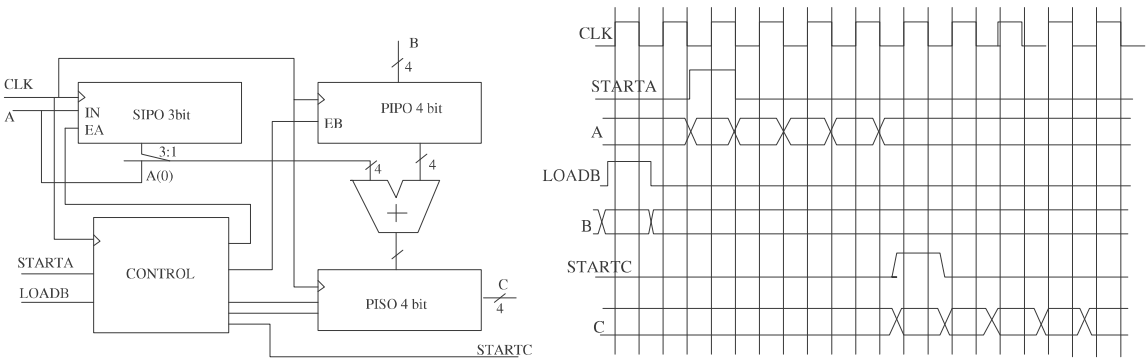

Figure 3.4: Structure of the Serial-in/Parallel-in Parallel-out Aritmetic and Logic Unit

3.1.6 Synthesis: how to analyze and constraint the power consumption

As usual copy the sipisoalu.vhd in the syn directory. Synthesize the sipisoalu for 32 bits without constraints and check how your FSM has been implemented.
Report timing and let's call MAX_PATH the maximum delay found.

Let's look at the power dissipation:

report_power

or by the main menu: **Design→Report Power**, leaving the default configuration (save report power file). The report you have here is general for the whole SIPISOALU, and separated in three contributions: do you understand the source of these power dissipation? You can have more detailed informations by selecting in the **Design→Report Power** window not summary only, but cells only: you have now the contribution on 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. A further information you can get on power is the toggle count of each net. Just type:

report_power -net

... for the power report regarding the circuit nets or from the **Design→Report Power** window selects "nets only". It displays the toggle activity of each net in the FA you are displaying. Note that if you don't change the values a default 0.5 toggle probability is assumed for the inputs, and the consequent probability is derived for the internal nodes. (This point will be clarified during the next lessons). Methods are available for importing the real switching activity after a backannotated simulation has been performed.

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

Let's create clock: supposing you used the name "CLK" in your VHDL for describing the clock signal, or that you don't have a clock but want to define a virtual one, you must now define the clock waveform (as before). This is more easily done usign the command window, in which you can type:

create_clock -name "CLK" -period MAX_PATH CLK

Note that in case you are working on a combinational circuit, this clock is only virtual and the second CLK occurence in the command line shouldn't be present. In any case such a virtual clock must be

defined for power computation, otherwise a defaould frequency would be used for reckoning the power.

Now you will note that the power is a little bit different:

report_power

A simple part of a script is given: **sipisoalu_pw.scr**. Note the general use of the variable "Period" (defined at the beginning). You can execute in single row steps (copy and past each row with mouse) to see what happens. Note the use of the power constraint (substitue 100 uW with a value lower than the one you obtained before):

set_max_dynamic_power 100 uW

Now compile again and analyze both the new power and timing results: You shoud expect that if you force a lower power limit the delay will increase... dow you know why?

Play with the constraints to find the compiler limit and save the timing and power results as suggested in the given script.

Summary of what is requested

Final synthesis script, initial and final power and timing reports.