Politecnico di Torino

Collegio ETF - ICM

Low-Power Electronic Systems
Lab 6

Marco Vacca, Mariagrazia Graziano, Fabrizio Riente,
William Baisi, Othman Laouibi, Valeria Piscopo, Marco Rausa

March 31, 2023

Unified Power Format (UPF)

In this chapter, you will experiment with the Unified Power Format (UPF) language. The purpose of UPF is the simulation of power elements such as power swith, voltage shifters, retention registers to design systems that can implement advanced powe reduction techniques, like power gating and dynamic voltage scaling, nowadays mandatory to obtain the lowest possible power consumption. To simulate such elements in your system you need 4 different elements:

- RTL description of your circuit (VHDL (.vhd) or Verilog file (.v));
- RTL description of a power management unit (PMU), a control unit dedicated to dynamically control power elements, to dynamically switch on and off different parts of the circuits or activate retention registers (VHDL (.vhd) or Verilog file (.v));
- Description of the power domains and power elements (UPF file (.upf)). This file is used both during the simulation (Modelsim) and the synthesis (Synopsys).
- Simulation testbench (System Verilog file (.sv)). System Verilog is required because it offers advanced functionalities to simulate UPF elements.

In the following an example circuit is provided to let you understand how to use UPF in your simulation. **Read everything carefully because in the next exercise you will have to design a circuit from scratch by yourself.** At the end of this first tutorial exercise you will be asked to do some simple analysis on the given circuit. The schematic of the test circuit is depicted in Fig. 6.2. As you may notice it is a simple ALU, made by a an 8 bit adder and an 8 bit multiplier. You can identify different elements in this circuit:

- In BLACK you may see the RTL elements, the circuit itself and the PMU. Registers are inserted at the beginning and the end of the circuit to implement a basic pipelined circuit. As you may notice, some control signals generated by the PMU are delayed of one clock cycle through a register to correctly synchronize them with the dataflow, otherwise you circuit will not work correctly.
- In different COLORS you may identify the power elements introduced by the UPF. Specifically the RED and GREEN color identify the sub-power domains of this circuit. One power domain is created for the multiplier, and one power domain is created for the adder. There are three aditional domains, 2 are highlighted in PURPLE and identify the retention controls on the registers, so that data are saved even when the two sub-domains are switched off, and one is the TOP (or global) domain of the entire circuit.
- Inside the two main sub-domains you can identify different power elements, the TRIANGLES are level shifter, that adapt the voltage level at the beginning and end of each sub-domain. The RECTANGLES identify isolation registers, that generate a fixed value at the output when the correspondent domain is switched off. The SWITCH identify the power transistors used to

implement the power gating to shut down a circuit. Usually two power transistors are used with CMOS logic, one on the Supply voltage pin and one on the Ground pin, but this is taken care automatically by the UPF for you.

Before moving on, understand carefully the schematic and the role of each element.

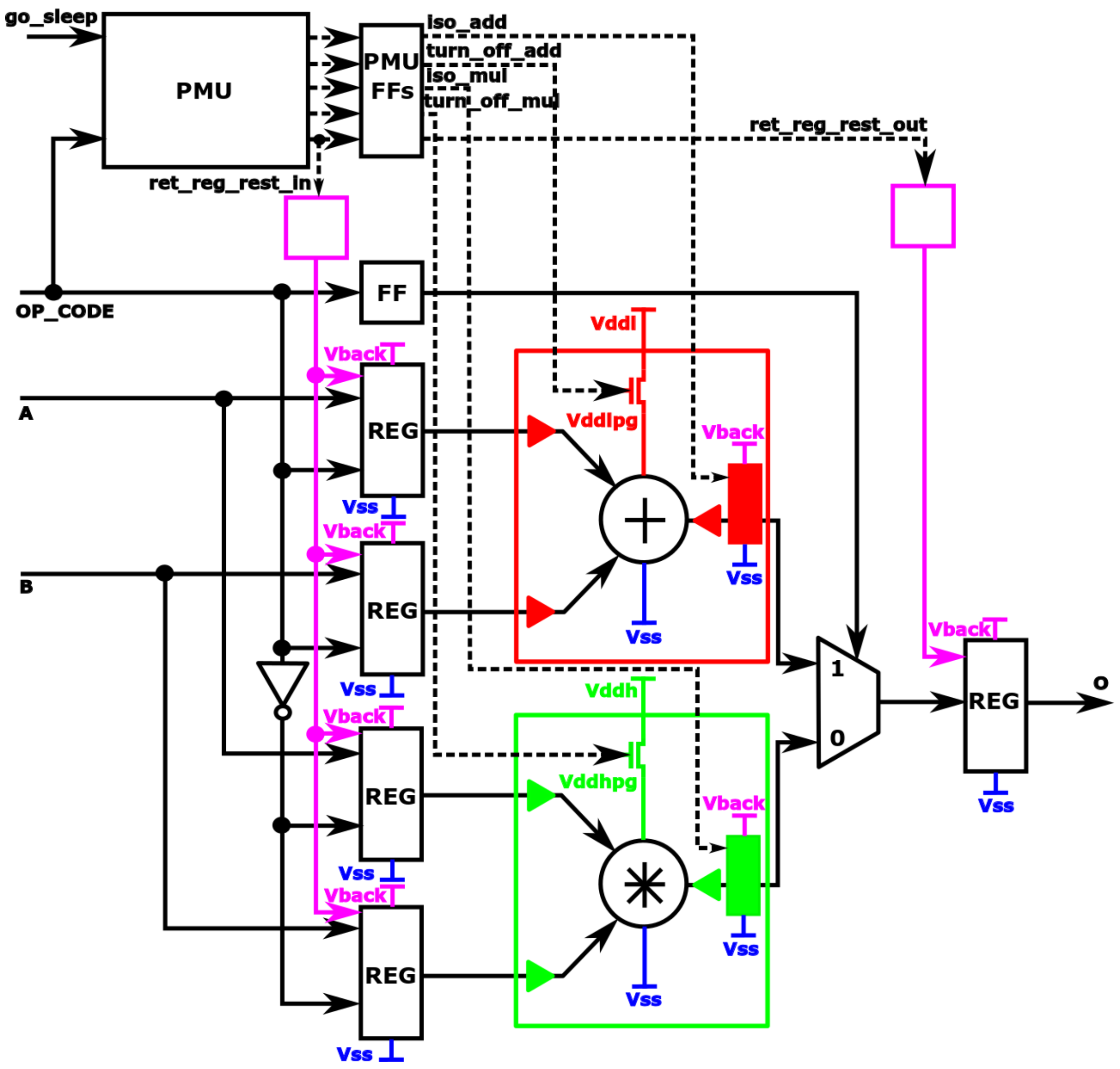

Figure 6.1:

IMPORTANT NOTE! Every single design of a circuit MUST start by hand drawing the block schematic of your circuit, yes, even in the age of videos that tells you everything to do, you MUST take pen&paper (or the virtual equivalent) and draw a schematic of what you want to create.

IMPORTANT NOTE 2! Read carefully the documentation, all the documentation. We know that is slow, but it is the only way to truly understand something.

6.1 Dummy ALU (TL & TP)

In this section of the Lab we will analyze the different parts that compose the test circuit. At the end you will be asked to do some simple analysis on the circuit to understand how voltage levels influence the power consumption.

You can download the source files with the following command

```
prompt> cp -r /home/repository/lowpower/cap6 .
```

6.1.1 RTL circuit descriptions

The VHDL code describing the circuit is in the folder *src*, take a look at it. There is not a lot to say regarding the RTL description of the test circuit, it is a simple circuit made by a mix of behavioral and structural components. For the sake of this laboratory behavioral description of the components (adder, multiplier, registers and multiplexer) would have been enough.

Regarding the Power Management Unit you may notice that is a simple behavioral description, that basically swith on and off the adder and the multiplier when they are not used, activating the corresponding isolation register when required. A *go_sleep* signal is used to switch off the entire circuit, activating the retention mode for the registers.

6.1.2 UPF file description

STEP1 - Initialization

The UPF file with the description of each component is in the *syn* folder, have a look at the *power.upf* file. In the following its part will be described. Let's start by initializing the file.

```
set upf_create_implicit_supply_sets false
set_scope /
```

The first command is used to avoid errors in the connection of the power domains, while the *set_scope* command defines on which part of the circuit the UPF is applied. In this case the UPF is applied to the whole circuit (in this case */final_tb_pa/UUT*, otherwise it is possible to use the name of a specific entity (by using its label defined in the VHDL file and the complete hierarchy) to apply it to a specific component, for example *comp_top_level/Multiplier_1* to apply the UPF only to the multiplier.

STEP2 - Power domains

The second step is the definition of the power domains.

```
create_power_domain TOP
create_power_domain PD1 -elements {comp_top_level/Adder_1}
create_power_domain PD2 -elements {comp_top_level/Multiplier_1}
create_power_domain REGIN -elements {comp_top_level/ff_sum_in_a
     comp_top_level/ff_sum_in_b comp_top_level/ff_mul_in_a comp_top_level/ff_mul_in_b}
create_power_domain REGOUT -elements {comp_top_level/ff_out}
```

In this case five domains are created, a global domain *TOP* that defines the highest level of the power domains hierarchy, two main sud domains for the adder and the multiplier, and two separate domains for the input registers and the output one. Two domains are required in this case since, due to the pipelined nature of the circuit, the control signals to activate the retention mode for input and output register must be different, since the output signal must be delayed by one clock cycle. The *-elements* option is used to identify which components are part of a domain, if not present the domains is applied to everything.

STEP3 - Power ports

The third step is the creation of the power ports, i.e. the voltage pins where the external voltages source must be connected to the circuit.

```
# POWER SUPPLY PORT
create_supply_port VDDL -domain TOP
create_supply_port VDDH -domain TOP
create_supply_port VBACK -domain TOP
create_supply_port VSS -domain TOP
```

They are equivalent at the *PORT* assignment in the *ENTITY* of the VHDL description. Four ports are created, *VDDL* is the power supply of the adder, *VDDH* is the power supply of the multiplier, *VBACK* is the backup power supply used for the registers in retention mode, *VSS* is the ground pin, all voltage domains will be connected to the same ground pin in this case. The *-domain* option is used to indicate at which power domain it refers.

STEP4 - Power nets

The fourth step is the creation of the supply nets for each domain, each domain requires the creation of dedicated voltage nets. A net represent the wire carrying around the voltage throughout the circuit.

```
#POWER SUPPLY NET (TOP)
create_supply_net VDDL -domain TOP
create_supply_net VDDH -domain TOP
create_supply_net VBACK -domain TOP
create_supply_net VSS -domain TOP

#POWER SUPPLY NET (PD1)
create_supply_net VDDL -domain PD1 -reuse
create_supply_net VSS -domain PD1 -reuse
create_supply_net VDDLPG -domain PD1

#POWER SUPPLY NET (PD2)
create_supply_net VDDH -domain PD2 -reuse
create_supply_net VSS -domain PD2 -reuse
create_supply_net VDDHPG -domain PD2

#POWER SUPPLY NET (REGIN)
create_supply_net VBACK -domain REGIN -reuse
create_supply_net VSS -domain REGIN -reuse

#POWER SUPPLY NET (REGOUT)
create_supply_net VBACK -domain REGOUT -reuse
create_supply_net VSS -domain REGOUT -reuse
```

The option *-reuse* indicates that the net must not be created, but it must be connected to an existing one, in this case the ones created in the *TOP* domain. *VDDLPG* and *VDDHPG* represent the voltage net connected after the power switch for the adder and the multiplier respectively.

STEP 5 - Ports/Nets connection

The next step is very simple, the supply nets must be connected to the supply ports, i.e. internal wires must be connected to the external pins.

```
connect_supply_net VDDL -ports {VDDL}
connect_supply_net VDDH -ports {VDDH}
connect_supply_net VBACK -ports {VBACK}
connect_supply_net VSS -ports {VSS}
```

Of course only the nets in the *TOP* domain must be connected, since all the others are derived from them.

STEP 6 - Primary supply nets

Finally, for each domain the primary supply nets must be defined.

```
set_domain_supply_net PD1 -primary_power_net VDDL -primary_ground_net VSS
set_domain_supply_net PD2 -primary_power_net VDDH -primary_ground_net VSS
set_domain_supply_net TOP -primary_power_net VBACK -primary_ground_net VSS
set_domain_supply_net REGIN -primary_power_net VBACK -primary_ground_net VSS
set_domain_supply_net REGOUT -primary_power_net VBACK -primary_ground_net VSS
```

STEP 7 - Power switches

The next step involves the creation of the power switch to switch on/off the domains.

```
create_power_switch PWRSW_PD1 -domain PD1 -input_supply_port {in VDDL}
     -output_supply_port {out VDDLPG} -control_port {sleep comp_top_level/turn_off_add}
     -on_state {ON in {!sleep}} -off_state {OFF {sleep}}
create_power_switch PWRSW_PD2 -domain PD2 -input_supply_port {in VDDH}
     -output_supply_port {out VDDHPG} -control_port {sleep comp_top_level/turn_off_mul}
     -on_state {ON in {!sleep}} -off_state {OFF {sleep}}
```

The *-domain* option allows to define the domain that the switch will refer to. The *-input_supply_port* and *-output_supply_port* options allows to define the input and output voltage nets for the switch. For each switch a *-control_port* must be defined. A label, *sleep*, for the control port must be defined, and it must be assigned to a signal in the VHDL code (*comp_top_level/turn_off_add* for example for the adder). Finally the rules that determines the on/off state of the switch must be defined. In this case the switch is ON when *sleep* and therefore (*comp_top_level/turn_off_add* for example is equal to logic 0 (*!sleep*) and OFF when it is equal to logic 1 (*sleep*).

STEP 8 - Level shifters

Now it is time to define the level shifter. Using different voltages implies that level shifter must be used to interface logic signals.

```
set_level_shifter LS1 -domain PD1 -applies_to inputs -rule high_to_low -location self
set_level_shifter LS2 -domain PD1 -applies_to outputs -rule low_to_high -location parent
set_level_shifter LS3 -domain PD2 -applies_to inputs -rule high_to_low -location self
set_level_shifter LS4 -domain PD2 -applies_to outputs -rule low_to_high -location parent
```

For each level shifter rules must be defined. The *-domain* defines the domain at which it refers. The *-applies_to* defines if it is applied to inputs or to outputs. The *-rule* option defines the scaling direction (from high to low voltage or the opposite). The *-location* defines in which domain the shifter is located, *self* means that is placed in the same domain of reference, *parent* means that is located in an higher domain in the hierarchy. Tipically, subdomains are at lower voltage with repsect to the parent domain, that it is why shifter are assigned like in this example, inputs shift from high to low voltage and outputs shift from low to high voltage, but, theoretically, it may be the opposite if required.

STEP 9 - Isolation cells

Isolation cells are typically used in conjunction with power switches. Their purpose is to keep the values at the output of a domain fixed when they are set, and they should be set when a domain is switched off.

```
set_isolation iso_cell_PD1 -domain PD1 -isolation_power_net VBACK -isolation_ground_net VSS
      -clamp_value 1 -applies_to outputs
set_isolation iso_cell_PD2 -domain PD2 -isolation_power_net VBACK -isolation_ground_net VSS
      -clamp_value 1 -applies_to outputs
set_isolation_control iso_cell_PD1 -domain PD1 -isolation_signal comp_top_level/iso_add
      -isolation_sense low -location parent
set_isolation_control iso_cell_PD2 -domain PD2 -isolation_signal comp_top_level/iso_mul
      -isolation_sense low -location parent
```

First the cell itself must be defined. As always the *-domain* option identify the domain of reference. The options *-isolation_power_net* and *-isolation_ground_net* defines the power supply nets of the isolation cells. In this case it is *VBACK* which is the net that is never switched off. The *-clamp_value* defines at which logic value the ouput must be fixed, and finally, the *-applies_to* rule defines if isolation cells must be applied to inputs or outputs. For each cell a control must be defined. The *-isolation_signal* option defines the name of the signal in the VHDL entity that controls the isolation cell. The *-isolation_sense* option identify the logic value that activates the cell, in this case it is *low*, so the cell is activated by a logic 0. Finally the *-parent* option defines the location of the cell in the domains hierarchy.

STEP 10 - Retention nets

Eventually retention rules can be defined for specific registers. Registers can be configured in such a way that, when the appropriate control signal is asserted, those registers keep the last value memorized and do not sample new data. They work in a similar way to isolation cells, but instead of clampling the output to 1 or 0, they clamp the output of the register at the last stored value.

```
set_retention ret_reg_in -domain REGIN -save_signal {comp_top_level/ret_reg_rest_in high}
   -restore_signal {comp_top_level/ret_reg_rest_in low} -retention_power_net VBACK
   -retention_ground_net VSS
map_retention_cell ret_reg_in -domain REGIN -lib_cells {RDFFARX1_HVT RDFFARX2_HVT}
set_retention ret_reg_out -domain REGOUT -save_signal {comp_top_level/ret_reg_rest_out high}
   -restore_signal {comp_top_level/ret_reg_rest_out low} -retention_power_net VBACK
   -retention_ground_net VSS
map_retention_cell ret_reg_out -domain REGOUT -lib_cells {RDFFARX1_HVT RDFFARX2_HVT}
```

Retention rules are applied to whole domains. The *-save_signal* option allows to define which signal set the output in transparent mode (registers operate normally) and which logic value must have the signal. Similarly the *-restore_signal* define when the register must be set in blocking mode, not accepting new data anymore. In this case the same signals, but with different values is used to control the retention network. For each retention network the proper *-retention_power_net* and *-retention_ground_net* must be defined, which in this case are *VBACK* and *VSS*. For each retention register it is possible to force the system to use a specific library cell with the command *map_retention_cell* and the option *-lib_cells*.

STEP 11 - Net states

The next step involves the definition of all possible states (voltage levels) for each power net.

```
add_port_state VDDL -state {HV 0.8} -state {LV 0.75} -state {OFF off}
add_port_state VDDH -state {HV 1.2} -state {LV 0.75} -state {OFF off}
add_port_state VBACK -state {HV 1.2}
add_port_state PWRSW_PD1/out -state {HV 0.8} -state {LV 0.75} -state {OFF off}
add_port_state PWRSW_PD2/out -state {HV 1.2} -state {LV 0.75} -state {OFF off}
add_port_state VSS -state {ON 0.0}
```

For each power net, a label is associated to a specific voltage to identify its state with the option *-state*. The possible voltage levels are defined in the technology library, for this case a SAED90nm technology, the voltage can go from 0.7V to 1,32V, with a nominal voltage of 1.2V. The keyword *off* identify that the supply net is swithed off

STEP 12 - Power state tables

Finally it is posible to define power states, essentially states that identify a specific condition of the circuit.

```
create_pst top_pst -supplies {VDDL VDDH VDDLPG VDDHPG VSS VBACK}
add_pst_state ADD_HS -pst top_pst -state {HV HV HV OFF ON HV}
add_pst_state ADD_LS -pst top_pst -state {LV HV LV OFF ON HV}
add_pst_state MUL_HS -pst top_pst -state {HV HV OFF HV ON HV}
add_pst_state MUL_LS -pst top_pst -state {HV LV OFF LV ON HV}
add_pst_state SLEEP -pst top_pst -state {OFF OFF OFF OFF ON HV}
```

The *create_pst* creates the power state table. The option *-supplies* identify the list of power nets that define a state. The *add_pst_state* command allow to define a specific state, by assigning it a label. The *-pst* option define which power table is used. The *-state* option contains the list of values (defined in the previous step) for each power net of the table. Power states are not used in our simple example, since voltage values are assigned directly in the testbench.

6.1.3 System Verilog Testbench

To test the circuit we need to use System Verilog instead of VHDL. Understanding System Verilog if you already know VHDL is quite straighforward, see it as a streamlined version of VHDL closer to C language. A testbench is already provided for this exercise (*/tb/final_tb_pa.sv*. Have a look at it, in the following you will find a brief explanation of the most important parts, so that you can easily use System Verilog for your testbenches in this lab.

```
‘timescale 1ns/1ps
module final_tb_pa;
import UPF::*;
```

This is the initialization part of the file, the first command defines the timescale of the simulation, the second one defined the name of the module and the start of it and the third one imports the library to use UPF files. This part is a mix between the library declaration and entity parts of a VHDL files.

```
localparam int unsigned NBITS = 8;
```

This command defines a local parameters, it has the same role of a generic constant of the VHDL in this case.

```
logic [NBITS-1 : 0] A_tb;
logic [NBITS-1 : 0] B_tb;
logic [2*NBITS-1 : 0] out_tb;
logic clk;
logic rst_n;
logic op_code_tb;
logic sleep;
bit status0,status1,status2,status3,status4,status5;
```

This section define local signals to be used inside the testbench, the first 3 are logic vectors, the last ones are of bit type. In VHDL this part is normally placed inside the *architecture* part.

```
dummyALU UUT(
    .clk(clk),
    .rst_n(rst_n),
    .O(out_tb),
    .B(B_tb),
    .A(A_tb),
    .go_sleep(sleep),
    .OP_CODE(op_code_tb)
);
```

This section is equivalent to the port map of the VHDL. The name of the component to be used *dummyALU* is followed by a label *UUT*. Then for every signal of the component (on the left) is assigned an internal signal (between brackets).

From now on it starts the part that in a VHDL file is usually located after the *Begin* of the *architecture*, the part where you decide how to assign values to the signals.

```
always@(posedge clk) begin
if(rst_n != 1'b0 && sleep != 1'b0) begin
    assign_inputs(A_tb, B_tb, op_code_tb);
        #10 compute_correct_result(A_tb, B_tb, op_code_tb, out_tb);
end
end
```

This section is equivalent to a sequential process of VHDL. The statement *always* means that this part is executed when always when a specific condition is met, in this case *(posedge clk)* means when a rising edge of the clock is detected. The combination of these two statements means that the execution of this part of the cose start when a rising edge is detected, then the code inside it is executed sequentially for how much time it requires to be completed. Inside this code an if is used to check the values of the *rst_n* and *sleep* signals. The notation *1'b0* means identify a 1 bit signal (*1'b*) of value 0. Inside the *if* statement the first subroutine (*assign_inputs* is executed. Then the system wait for 10 time steps (*#10*) and then executes the second subroutine *compute_correct_results*. Summarizing the code here written checks when a rising edge clock signals, then executes the code sequentially taking 10 time steps to conclude, then it wait for another clock rising edge to start again.

```
always #2 clk = ~clk;
```

This statement is executed *always* every 2 time steps, inverting the value of the clock.

```
initial begin
    rst_n = 0;
    clk = 1;
    A_tb = 8'b00000000;
    B_tb = 8'b00000000;
    op_code_tb = 1'b0;

    // PA Initializations
    status0=supply_on("UUT/VBACK", 1.2);
    status1=supply_on("UUT/VSS", 0.0);
    status2=supply_on("UUT/VDDH", 1.2);
    status3=supply_on("UUT/VDDL", 0.8);

    sleep = 1'b1;
    #10 rst_n = 1;
    #1000;
    status0=supply_on("UUT/VBACK", 1.2);
    status1=supply_on("UUT/VSS", 0.0);
    status2=supply_on("UUT/VDDH", 0.8);
    status3=supply_on("UUT/VDDL", 0.75);
    #1000
    sleep = 1'b0;
    #4
    status4=supply_off("UUT/VDDL");
    status5=supply_off("UUT/VDDH");
    #200;
    status2=supply_on("UUT/VDDH", 0.8);
    status3=supply_on("UUT/VDDL", 0.75);
    #4
    sleep = 1'b1;
end
```

This part should be quite easy to understand. At the beginning signals are initialized, supply voltages are switched on and a voltage is applied to them. After 10 time steps *#10* the reset is disabled so the circuit may start to operate. After other 1000 time steps (#1000) the value of the supply voltages is changed and so on. The character # is somehow equivalent to the *wait* or *after* statements in VHDL. Please notice that the power supply can be switched off (*supply_off*), this is an additional command that can be used to completely disable a power supply, in addition to activate the power switches. It is IMPORTANT that you first acrivate the isolation or retention register, and then, after some time, you deactivate the power supply. Similarly, first the power supplies must be reactivated and then, after some time, the circuit must go in the normal operation mode. Otherwise the circuit will not be able to work properly after it goes to sleep.

The rest of the code is based on functions, or tasks as they are called. To implement a function you just have to declare its name *assign_inputs()*, and then pass some signals to it as parameters *assign_inputs(A_tb, B_tb, op_code_tb).*

```
task assign_inputs;
    output [NBITS-1 : 0] A, B;
    output op_code;
    begin
    A = $urandom_range(0, 100);
    B = $urandom_range(0, 100);
    op_code = $urandom_range(0, 1);
    end
endtask
```

In the body of the task, temporary signals are created, in this case *A, B, op_code*. For each signals it is necessary to define is the task must generate a value and assign it to the parameters (*output*), or if the task must read the parameters (*output*). In this case parameters are passed to the task, the task calculate random integers number (from 0 to 100, and 0 or 1 in the last case), converts them to logic or logic vector values, and it assign them to the temporary signals that are then copied to the signal passed as parameters. That is way the signals are declared as *output*.

```
task compute_correct_result;
    input [NBITS-1 : 0] A, B;
    input op_code;
    input [2*NBITS-1 : 0] res;
    int correct_res;

    begin

        $write("[%0t] %d ", $time, A);

        if(op_code == 1) begin
            $write("+ %d = %d\n", B, res);
            correct_res = A + B;
        end else begin
            $write("* %d = %d\n", B, res);
            correct_res = A * B;
        end

    if(correct_res != res) begin
            $display("Incorrect result! Correct result is %d", correct_res);
    end
    end

endtask
```

This second function read the parameters *input*. The parameters read are the inputs generated by the previous task (*A, B, op_code*) and then it calculates the theoretical correct value by adding or multiplying the inputs, assigning the results to a temporary variable (*correct_res*). It also reads the output generated by the adder or multiplier and compares it with the theoretical expected result. If the two values match, the result is printed on the Modelsim terminal, otherwise an error message is printed instead, menaing that the circuit is not working properly. It is a very simple form of a very important part of the design process, called verification.

6.1.4 Simulation and Synthesis

Now that you should have understood what you are doing you can go on with the simulation and synthesis. Open a terminal inside the folder *tb* and start Modelsim. A simulation script has already been provided for this goal. Open the file *compile_PA.tcl* and try to understand it. You should be able to understand the commands, the only thing that you may notice are the presence of additional commands, *-pa* for the power analysis. Run, from Modelsim command line the following command:

do compile_PA.tcl

Does the circuit behave correctly? What it the effect of UPF commands? What happens when the circuit is forced in sleep mode?

You can now synthesize the circuit with synopsys. Open a terminal inside the *syn* folder and launch Synopsys. Also in this case a synthesys script has been already provided for you. Open the file *compile.src*, you should be able to understand what it does. The only differences are the commands used to set a voltage to a power port. While in Modelsim you can set a different voltage dynamically, in case of Synopsys you must set a fixed voltage to estimate the circuit performance. Run, from Synopsys command line the following command:

source compile.src

What is the power consumption of the circuit? What are the most power hungry components? Is the critical path respected?

Remark point

Power consumption, critical path, comments.

6.1.5 EXERCISE 1: Power supply voltage and power consumption

In this exercise we are going to analyze the influence of the power supply voltage on the power consumption of our circuit.

Modifiy the *compile.src* file and change the voltages, set the voltages **for both the adder and the multiplier to 0,8V**. Leave *VBACK* to 1,2V. When you synthesize the circuit you can tell to Synopsys to measure the power consumption in different operating conditions, so, **at the end** of the *compile.src* script change the power supply voltage by inserting again the commands:

```
set_voltage 0.8 -obj {VDDLPG}
set_voltage 0.8 -obj {VDDL}
set_voltage 1.2 -obj {VDDH}
set_voltage 1.2 -obj {VBACK}
set_voltage 1.2 -obj {VDDHPG}
set_voltage 0 -obj {VSS}
```

and then you can print new power reports inserting again the commands:

```
report_timing > ./results/timing.txt
report_area -hier > ./results/area.txt
report_power -hier > ./results/power.txt
```

Change the name of the files otherwise the old ones will be overwritten. Repeat this step three times so that you test the circuit in 4 different conditions:

- Vadd = 0,8V, Vmult = 0,8V (Initial condition)

- Vadd = 0,8V, Vmult = 1,2V
- Vadd = 1,2V, Vmult = 0,8V
- Vadd = 1,2V, Vmult = 1,2V

Run the synthesis script, read the reports generated and then write down the total power consumption, the data arrival time and the total circuit area. Is the critical path always respected?

Now, create a new synthesis script by copying/pasting the old one and repeat the same analysis **but change the initial conditions**, i.e. the values of voltage before the *compile* command, so that the voltages applied to the adder and the multiplier is in both cases 1,2V. Modify the rest of the script in such a way that the 4 conditions listed before are tested. As always leave the *VBACK* to 1,2V. Run again the synthesis script and write down again the total power consumption, the data arrival time and the total circuit area. Why the values are completely different? How power consumption, timing and area change?

Remark point

Synthesis scripts, power consumption, timing, area and comments.

6.2 EXERCISE 2: Circuit design (TL & TP)

In this exercise you will have to design a circuit, complete with power components, test and synthesize it. The circuit is reported in the following figure.

As you may see the circuit is a modified version of the previous ALU, where, in addition to the multiplier and adder, a programmable logic unit and a 3bit programmable Look Up Table (LUT) has been added. A LUT can be implemented in different ways, in this implementation it is just a register, containing the table of truth of the logic function that you want to implement, connected to a multiplexer, where the selection bits are used as logic inputs. Every airthmetic/logic unit must have its own power domain, complete with voltage shifters and isolator cells, so that the voltage can be independently regulated and the unit can be switched on/off. The isolator set the output to logic '0' when active. Each register must implement a retention policy. The circuit can be in 4 different power states:

- SLEEP: Everything is off except for the registers.
- LOW_POWER_MODE: Only the adder is active.
- NORMAL_MODE: The adder, the logic unit and the LUT are active.
- FULL_POWER_MODE: Everything is active.

What to do:

- Design the logic circuit at RTL level. You can write your own code or recycle part of the code from the previous exercise. Components can have a behavioral description, if it is written in a synthesizable form.
- Design the PMU unit, which must generate the power control signals. Logic control signals can be generated directly within the testbench. Pay attention to the timing of the control signals, given the pipelined nature of the circuit, otherwise it will not behave correctly.
- Create the UPF file describing the power components.
- Create the system verilog testbench, a simulation script and then test your circuit.

- Create the synthesis script and synthesize your circuit when all power supplies are equal to 0,8V and 1,2V.

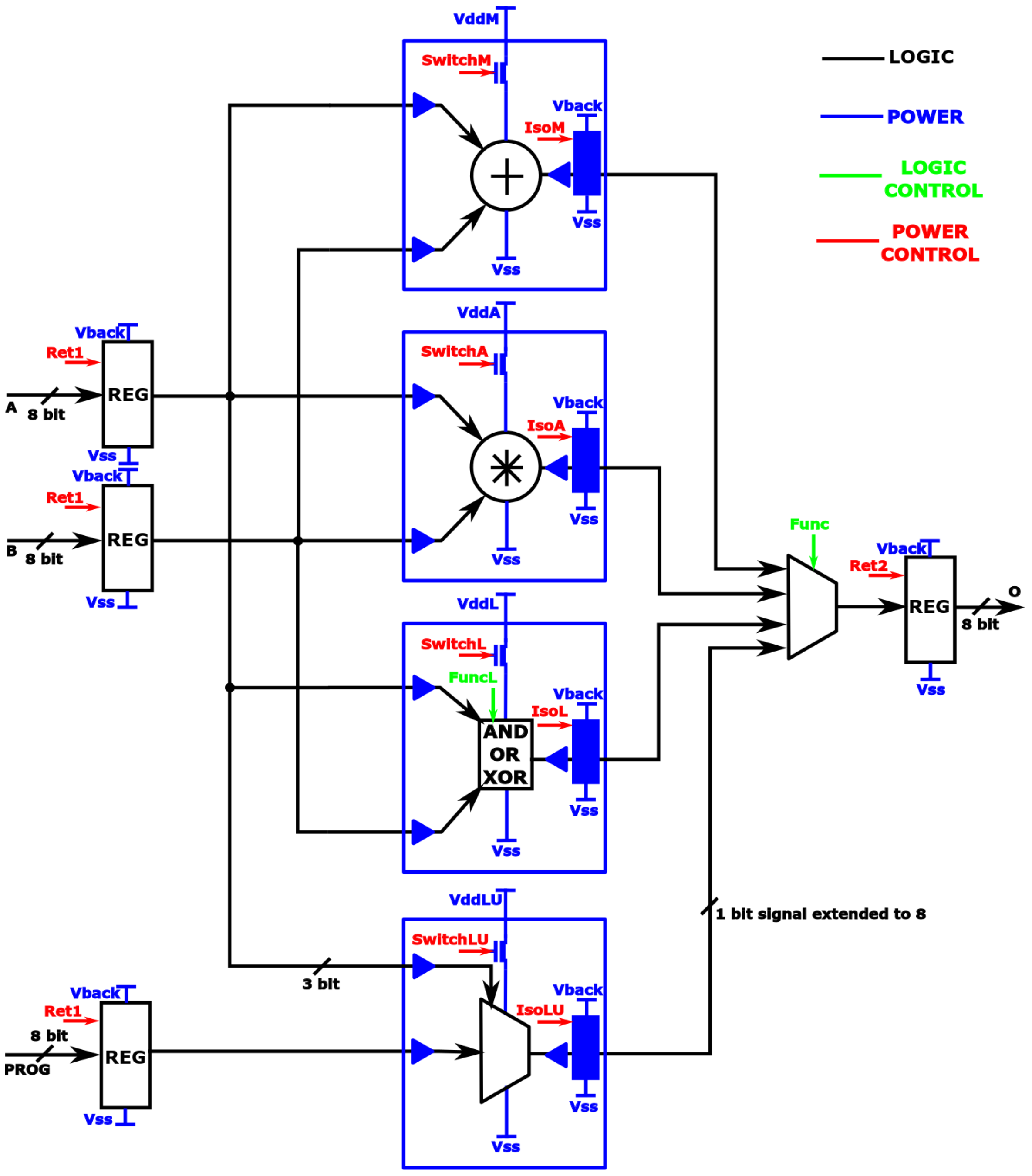

Figure 6.2:

Remark point

VHDL netlist of the circuit and the PMU, UPF file, simulation script, synthesis script, system verilog testbench, power, timing and area analysis.