**Name: Abdelkarim benjilali.**

**My microprocessor.**

**Introduction:**

Following the VHDL course that I followed as part of my engineering training, I wanted to go a step further in the discovery and use of this hardware logic development, the principles of which I would explain in the rest of this report. To make it an interesting challenge for myself, I chose to design the architecture and then describe the behavior in VHDL of my own microprocessor hierarchically from basic components like registers etc.., starting from scratch.

The interest of this way of proceeding is twofold: at the end of the project and thanks to the successive errors, I have a much better understanding of the VHDL language and its mechanisms but above all of the way of considering the design of a complex system in digital electronics (and more generally).

This article therefore traces all the creation, from the definition of the architecture to functional tests of my simplistic processor.

**What is VHDL?**

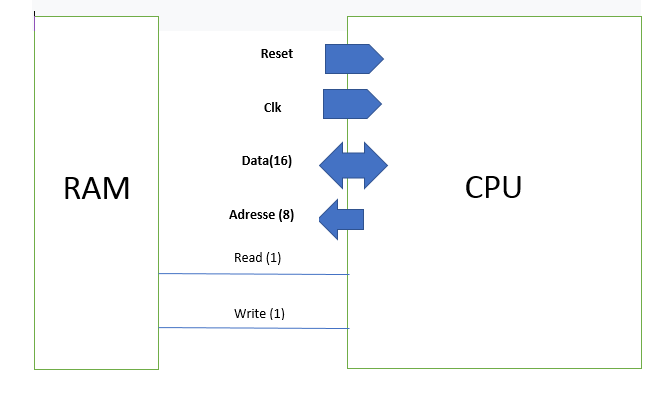
VHDL (VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

For example a line a<=b and c will be implemented physically with an and gate along with taking a, b and c as 3 input signals.

**General architecture:**

Before I begin explaining the architecture I had some schemas that I wrote in paper and that I could’ve uploaded here but I prefer showing the architecture proposed by the software “Quartus”.

So the final model doesn’t contain only the micro processor but it’s a kind of like a computer that includes the microprocessor, RAM.

The ram that I added is just for testing how my processor works.

**Let's define each element of the CPU:**

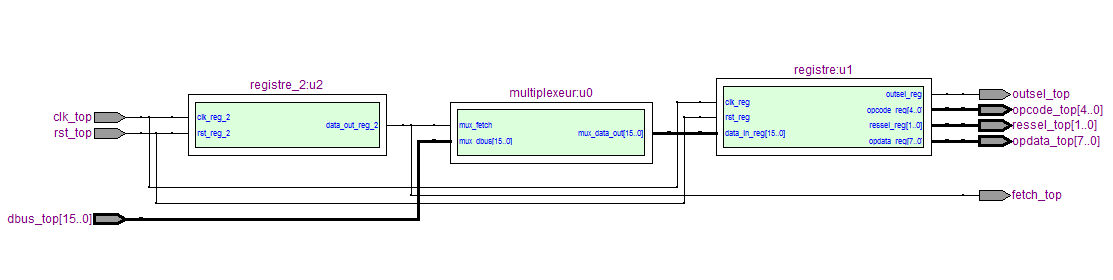
* **The Fetch & Opcode:**

-The Fetch & Opcode Manager is responsible for retrieving the instructions in memory and passing them to the ALU and the rest of the system.

Each instruction is executed within two cycles:

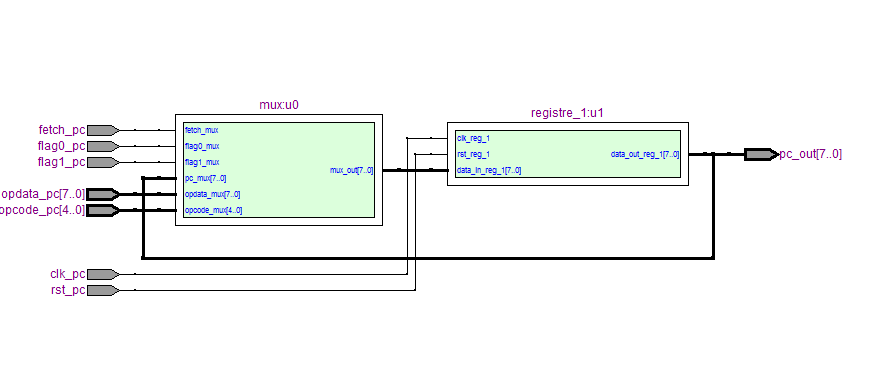
-Getting Instruction from the program counter PC.

-Execute the instruction (arithmetic or logic operations, read from ram or write..).

The Fetch & OpCode Manager takes care of the alternation between the fetch and instruction cycle, and keeps the instruction to be executed throughout the instruction cycle, after each fetch.

In addition, it ‘s used by the rest of the CPU (and in particular the ALU). It is very important to note that it takes the value zero during fetch. This corresponds to a NOP, because the rest of the system must not perform operations during a fetch cycle.

* **Program counter:**

-The Program Counter (PC) holds the address information of the next instruction to be loaded by the Fetch & Opcode Manager. It increments normally after each instruction, but can be directly changed during a jump (GoTo instruction).

The Program Counter has only one output: the following address value to be fetched. It can change at each rising edge of the clock for three possible values:

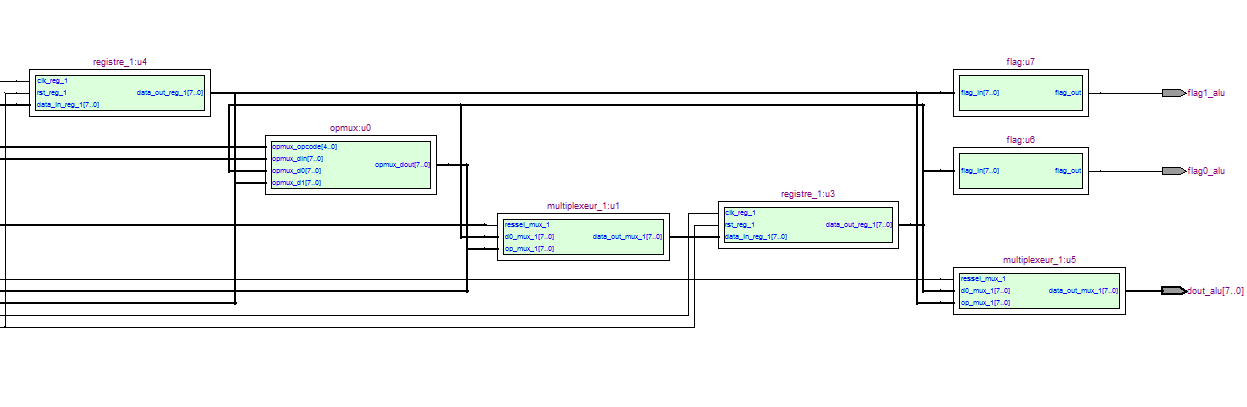
• PC + 1, always after a fetch;

• PC, itself. Always at the end of the instruction cycle, except in the following case;

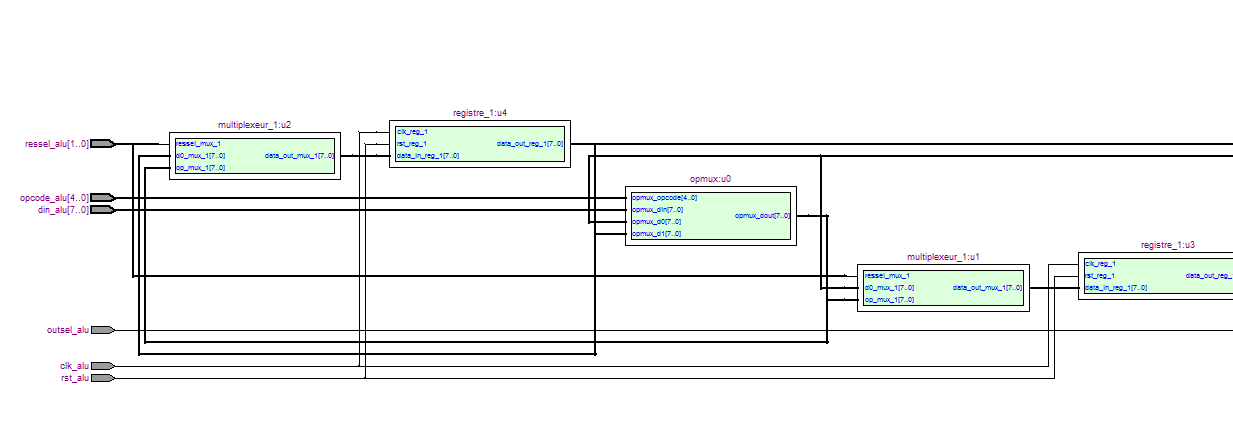
• OP Data, during a conditional or unconditional jump, we go to the address passed in argument by checking, if necessary, the flags of the ALU.

* **ALU:**

The ALU (Arithmetic and Logical Unit) is the CPU core: it can perform logical (AND, OR, etc.) and arithmetic (+, -, etc.) operations on its registers.

*All registers are assumed to be connected to the asynchronous reset, and active on a rising clock edge*.

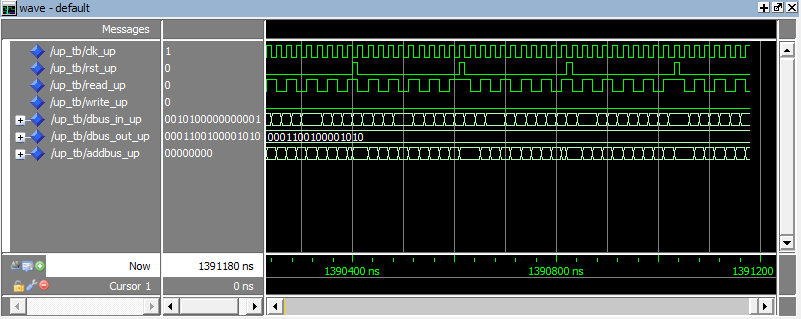
It works on the following principle:

* The OPMux is the multiplexer allowing to choose the operation. As input, we find all the possible operations, while only one is chosen as output, according to the code of the operation;
* **The two registers contain the variables used by the processor, which are the operands of the ALU;
* The two multiplexes allow you to choose whether you want to copy the result of the operation into the register or keep their previous value;
* The "flags" are Boolean outputs (True or False) which allow a condition to be checked (here, register1 or 2 = 0?). This allows conditional jumps such as "Go to address @ 1 if register0 = 0".
* Finally, a last multiplexer makes it possible to choose to which register the output is connected (using outsel bit).

To add instructions, it would suffice to add inputs to the OPMux multiplexer. The instruction code (OPCode) being, according to my specifications, on 5 bits, we have a total of 2 ^ 5 = 32 possible codes. I only used 12 but the rest Is for further development.

Note: The output of the ALU is only used by the "Get" command. The OutSel bit is therefore only used in this case. It can easily be used to encode the OPCode for other operations.

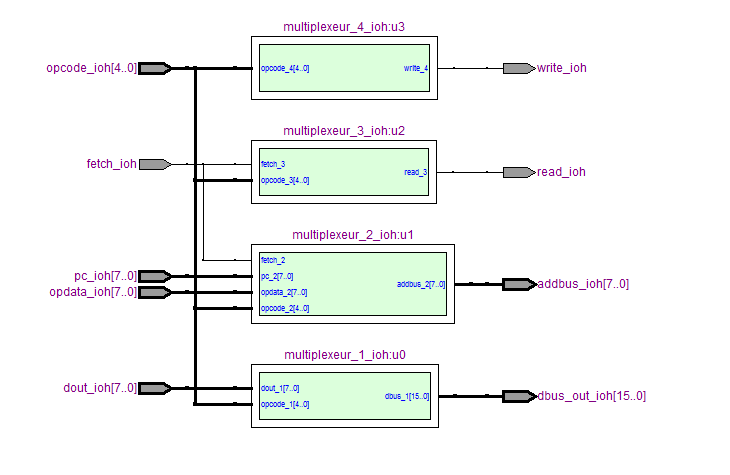
**Simulation:**

****

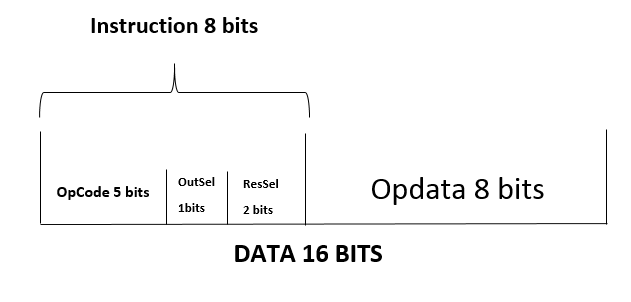
**Input output manager:**

-The I / O block is controlled by the Fetch & Opcode Manager. It takes care of the state of the inputs / outputs, for example: connect the output

of the ALU to the data bus, put the contents of the PC in the address bus, etc..



The input / output manager simply adapts to the rest to choose the state of the ports. For example, if the OPCode is a Get, it will set Read to 0, Write to 1, the address to the OP Data (the argument of the instruction) and connect the data bus to Dout of the ALU.

While designing the architecture of my CPU, I had to determine an instruction set and structure for them in memory. The instruction set is the set of commands that the CPU is capable of executing. An instruction will be structured as follows:

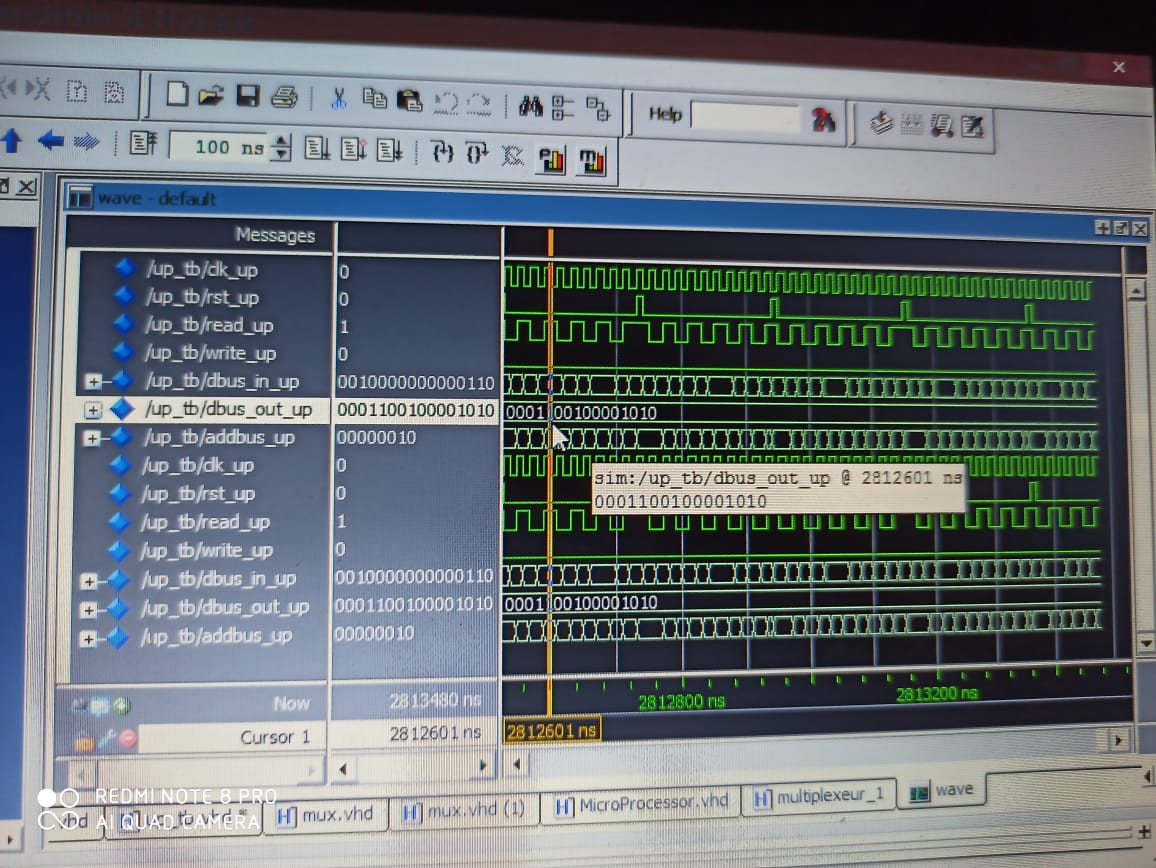
The length is fixed at 16 bits and the execution time at a clock period (so two, if we count the fetch). The OP Data field contains the arguments of the instruction. The OutSel and ResSel fields are data used for the microprocessor to know if he has to change the data in registers or keep the last one and for him to know If he should send an output or not.

**CPU:**

The CPU is made of 4 parts:

Alu, fetch&decode, I/O manager and Pc.

**Simulation:**

****

* **Ram:**

To "use" this processor outside simulation, it is obviously necessary to add the memory containing the data and the instructions.

For that, I realized the following module in VHDL which controls a memory (a list of registers).

It is consistent with the previously defined architecture: 16 data bits, 8 addressing bits, a Read and a Write.

The "DataBus" port is represented twice because it is a bidirectional bus, sometimes used as input, sometimes as output.

The idea is simple (as proof, the VHDL code is 3 lines!):

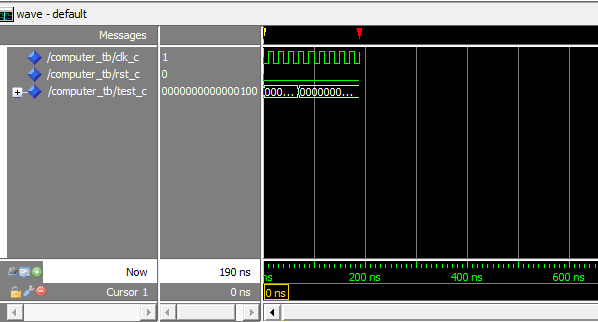
You can check the ram codes for it.

**The computer:**

This component gathers all the components bellow specially the CPU and the ram making a computer.

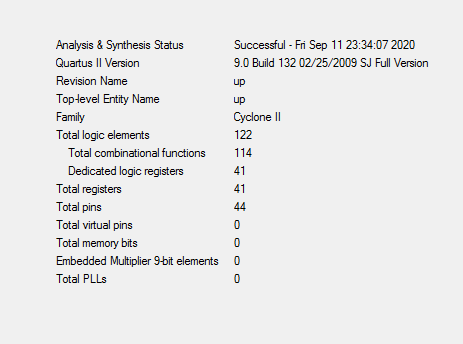
Here I added a test bit to the ram that reads me a certain case in which I worked

And charged a small program in the RAM 4 instructions and here is the result:



**Conclusion:**

**Performance:**

****

Although processor performance was never a goal of this project, it is still interesting to analyze the resulting system:

• All instructions execute on a constant time of two clock cycles and are 16-bit.

• The maximum propagation time (worst case) corresponding to a maximum clock frequency of 150 MHz, or 75,000,000 instructions per second.

In absolute terms, this frequency does not mean much because it is strongly determined by the FPGA.