DLX PROJECT

Note: the original project's idea was conceived with the presence of three peripherals: SPI, UART and JTAG. Anyway, as the last project's deadline was too close and we had not time enough we decided (following Prof. M. G. Graziano's opinion that what was done was enough to confirm our oral exams score) to complete and implement only what follows:

Our DLX processor project has these features:

-pipeline

-basic Instruction Set Architecture

-instruction subset Instruction Set Architecture

-control hazard

-data path

-synthesis

-micro programmed control unit

-ALU optimized for high performance (high area consuming)

-Titan 2 processor logic unit

-Titan 2 processor comparator unit

-booth multiplier

PIPELINE: it is made of 5 stages:

Instruction Fetch	(IF)
Instruction Decode	(ID)
Execution	(EX)
Memorization	(MEM)
Write Back	(WB)

We decided not to add any other stage in respect of the original DLX structure.

Since we did not implement the floating-point instructions there is no operation that lasts more than one clock cycle to finish. So, the control unit does not manage this case.

CONTROL UNIT: For the control unit we made some customizations. First, we decided to make a clocked control unit instead of an asynchronous one like in the original DLX project because in this way it is easier to make a behavioral VHDL description of it. Of course, not all the signals are variable inside the control unit process but only those that split what comes in from the instruction bus.

Another feature of this control unit is that the control word is already internally properly split into stages. So external registers for splitting the CW are not needed anymore. Of course, as it is a behavioral description, the synthesizer will finally decide whether to put them or not.

It is a microprogrammed control unit. That makes the process of adding microinstructions much easier than the Final State Machine original DLX implementation. The microcode is stored inside a hardwired vector so

as possible future implementations will allow to re-program the microcode at runtime like modern Intel processors.

Another possible future optimization can be that to reduce the number of bits for each row of the microcode. In the actual implementation, each row of different operations is ORed to form the final control word but can be opportunely shifted so that ORing is still possible and bits of memory will drastically decrease.

HAZARD DETECTION UNIT: Originally, we thought to make an external unit that (like in the original DLX project) forced nop operations into the control unit when there was a hazard and stall the Program Counter register opportunely. But finally, we decided to embed it inside the behavioral control unit because, in this way, we have direct access to all the variables inside the process and we can immediately manage the hazards without waiting many clock cycles during the interaction between the hazard detection unit and the control unit. So, we put the hazard detection unit at the beginning of the behavioral control unit description. In this way the changes to the internal variables will be immediately effective before reaching the real control unit implementation. Finally, we preferred to make only one module instead of two to prevent a lot of possible glitches and, consequently, malfunctioning.

This unit does not implement the forwarding technic and for each jump, taken or not, four clock cycles are wasted both for absolute or relative jumps. We decided to adopt this implementation to keep the area of the processor not so big. By forwarding we save just a clock cycle that does not improve the performances so much. A branch prediction unit will increase the hardware area a lot.

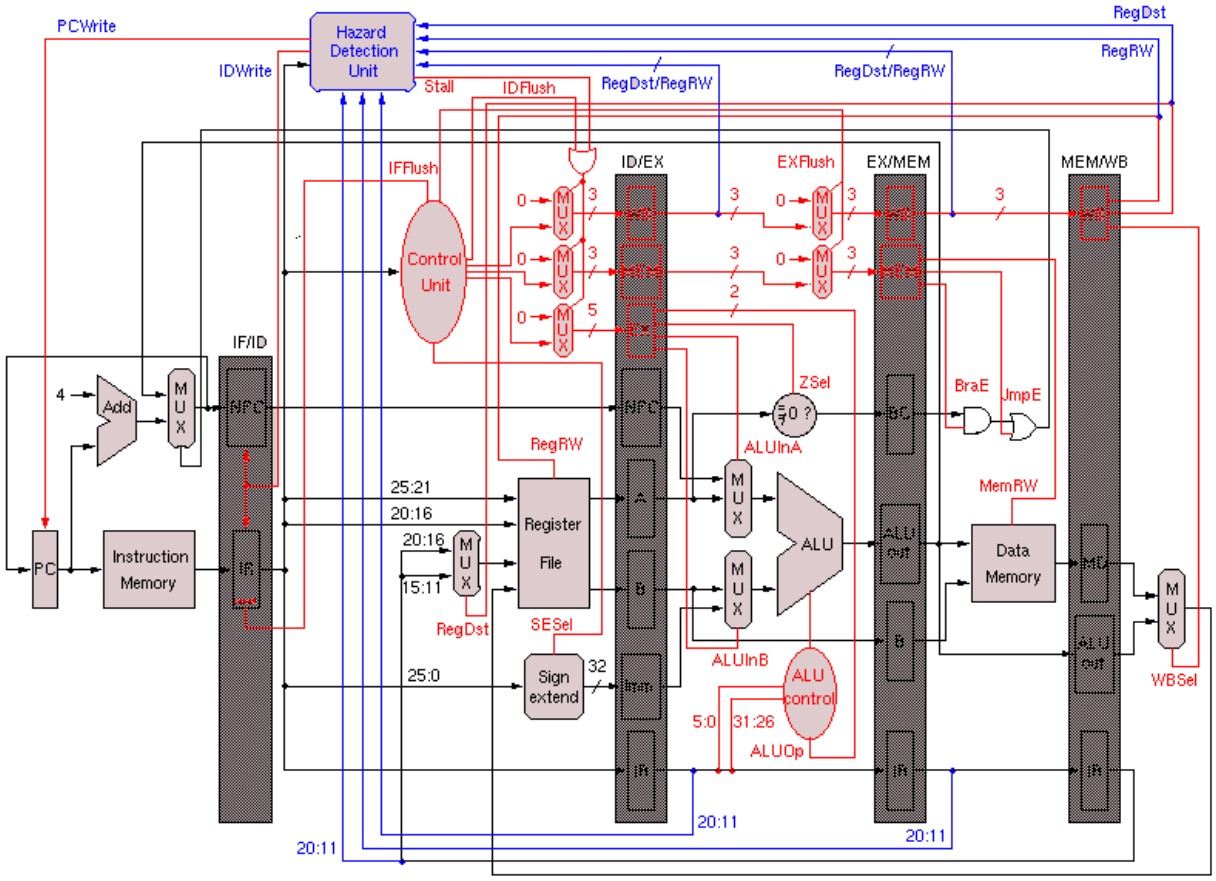
This unit recognizes hazards between all kinds of operations regardless of the stage in which there is the hazard and regardless of all the kinds of operations that originate (R or I type).

ALU: this unit, as explained above, is thought for high performances and, therefore, for bad area saving. Of course, it can be changed with a slower unit but capable to save a bigger area. For example, a behavioral description of all the operations can be done and the synthesis tool can be forced to save area.

The current ALU unit description is structural. The only exception is represented by the process inside the top-level entity that includes all the structural modules implementing each operation. We left them inside a process because the final synthesis of it should only make a logic network that programs the logic and comparator unit properly so to output the requested operation result starting from the ALU selection bits.

We made a structural description of the ALU because we want to force the synthesis tool to perform the fastest implementation possible of the module to maintain all the paths approximately the same length. Of course, this solution must be compared to the high-performance solution generated by the synthesis tool starting from a behavioral description.

DATAPATH: apart from the control unit (the red module), the hazard detection unit (the blue unit), all the control word registers (red registers) and ALU control unit that is embedded inside the control unit microcode, it reflects the standard DLX implementation.



Group "On Edge" - Pipelined DLX, with Data Hazard Detection Unit and Control Hazards

The top-level entity is characterized by the fact that all the memories (Instruction memory/Register file/Data memory) are thought to be outside. Another higher-level unit must connect the DLX with his memories. In our case the testbench does this task. We decided to take apart these memories because we know that actual memories are not made with a hardware description language but are projected at a very low level. So, we give the possibility to connect to our processor modern full custom projected memories or, in the case of FPGAs, possible internal SRAM blocks.

The data path is thought for having two different main memories: the instruction and the data one. A Memory Management Unit or a module between the Datapath and the main memory must be designed if we want these memories to refer to a single physical memory like in modern processors. It must provide solutions for multiple reading requests from the Datapath since it can easily happen that data memory must be read contemporaneously to the program memory and if they refer to a single physical memory this issue must be handled.

This implementation has the advantage to force the synthesis tool not to synthesize the memories, so we can really see the result of our constrained synthesis. For instance, the presence of these memories can increase the length of the critical path, in this way the synthesis must be performed with a lower clock frequency instead of the highest one possible.

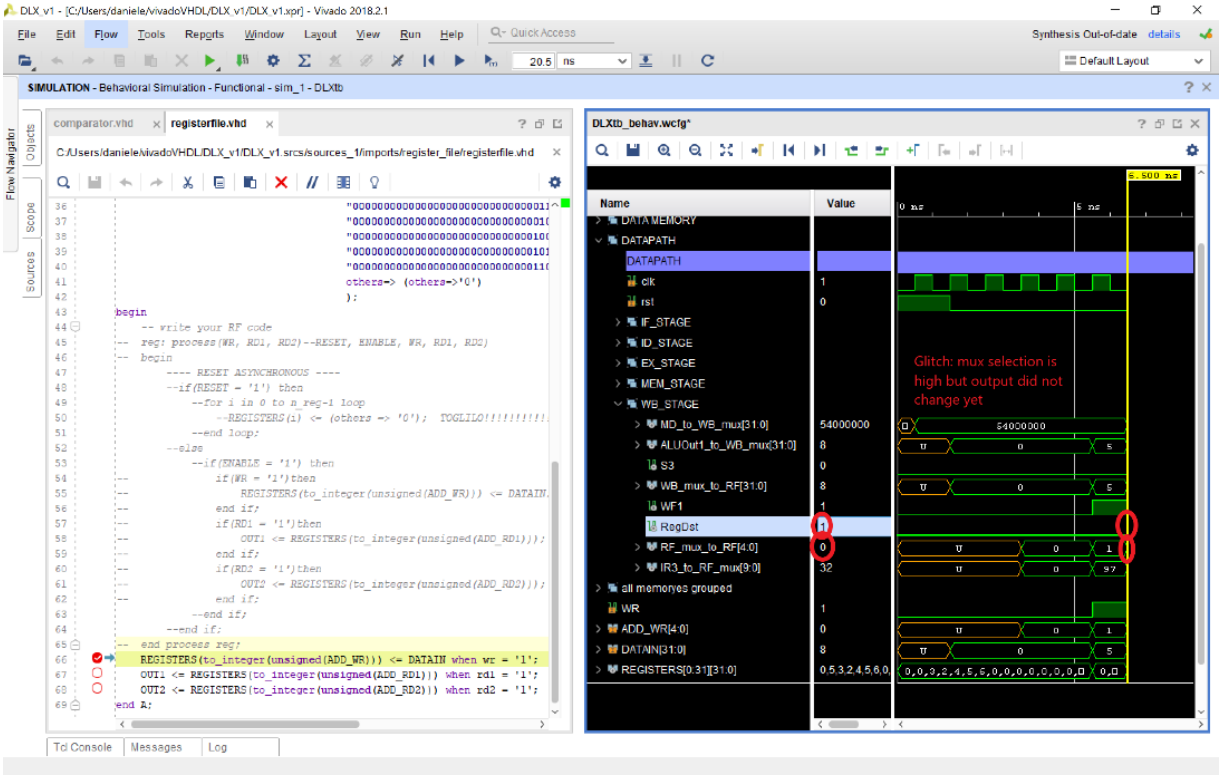
Many precautions were taken also in reducing, as much as possible, the switching activity of the buses and of the components in the Datapath.

MEMORIES (PROGRAM MEMORY /REGISTERFILE /DATA MEMORY): all of them are designed as combinational circuits respecting the original DLX implementation. Of course, modern memories are not produced anymore as combinational but as sequential ones so to prevent glitches. By a combinational circuit to make designs is very time-consuming because we must guarantee that every single signal arrives at the right time, otherwise there will be a glitch and the result will be "unpredictable". Even VHDL statements have delta delays so glitches can occur and when this happens a lot of waveform displayers do not show the delta time inferred by the glitch.

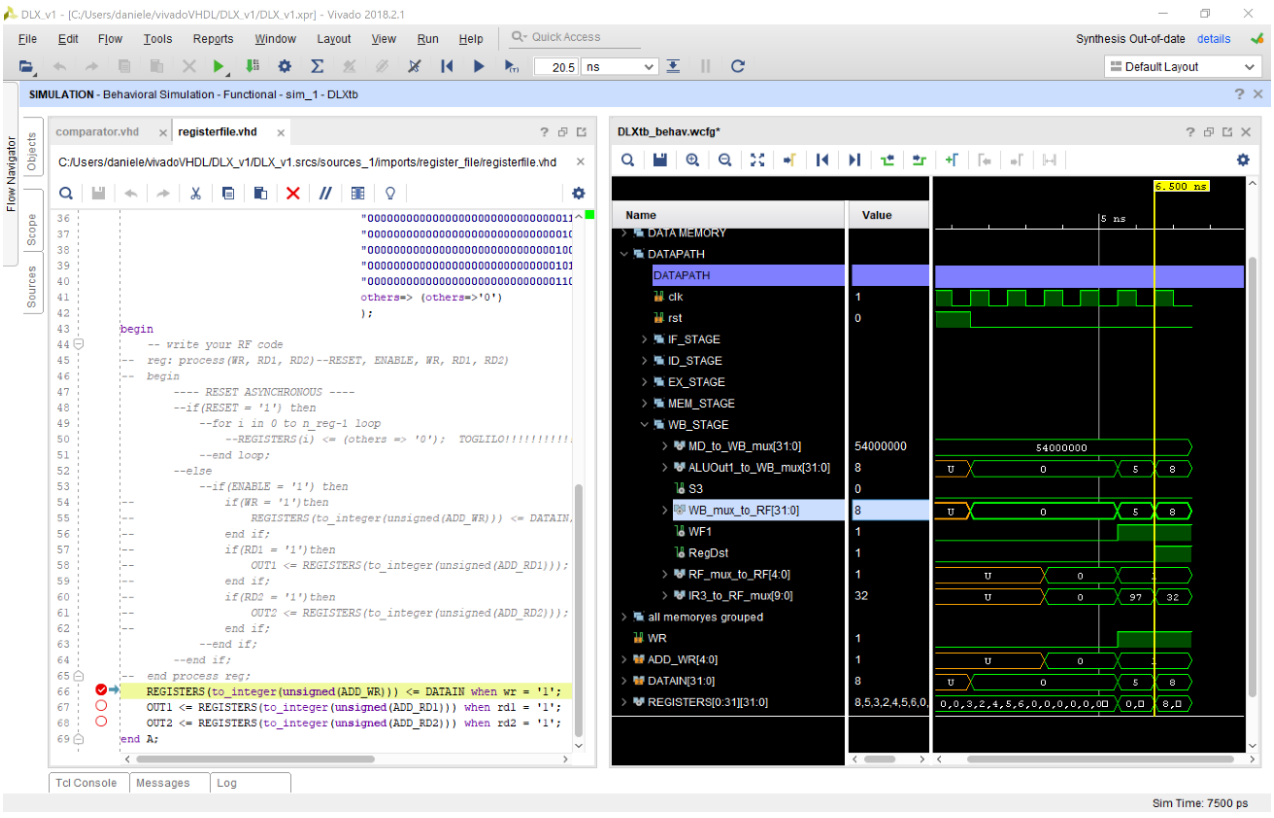
Again, the fact that we made memories external to the Datapath, gives the possibility to develop a synchronous interface between the DLX and the main memory. Designing such a synchronous block allows to care only of not violating the clock's setup and hold times.

Anyway, we had some problems while designing these memories since they were combinational. A lot of glitches came from components like multiplexers and even the control unit affected the results inside the register file. The first decision to change all the memories in sequential circuits were discarded because we noticed that even if we synchronized them no one could guarantee us this control unit signals would not glitch and so nothing would have been solved. Even clock's hold and setup times could be violated. Another possible solution was to clock the control unit making it sensitive to the falling edge of the clock instead of the rising one for the Datapath, but this solution has the big disadvantage that in this way all the ALU operations must be performed in half of the clock's period. Maybe this solution could be good for a high-performance adder like the Ladner Fisher's one, but this could not be suitable for many other operations. At the end we decided to put non-synthesizable inertial delays at the input of all the memories in the Datapath. In this way every variation below a certain threshold will be not listened and only a very small fraction of the entire clock period will be wasted to avoid glitches. Certainly, these memories are not synthesizable anymore but, as told above, we do not want to synthetize them. To solve this issue properly buffers must be put for each single instruction to guarantee that signals will vary in the correct timing. For this microelectronic course we need a more general solution. That is why we decided to put inertial delays.

Here I show two glitches I found before inserting inertial delays into the code:

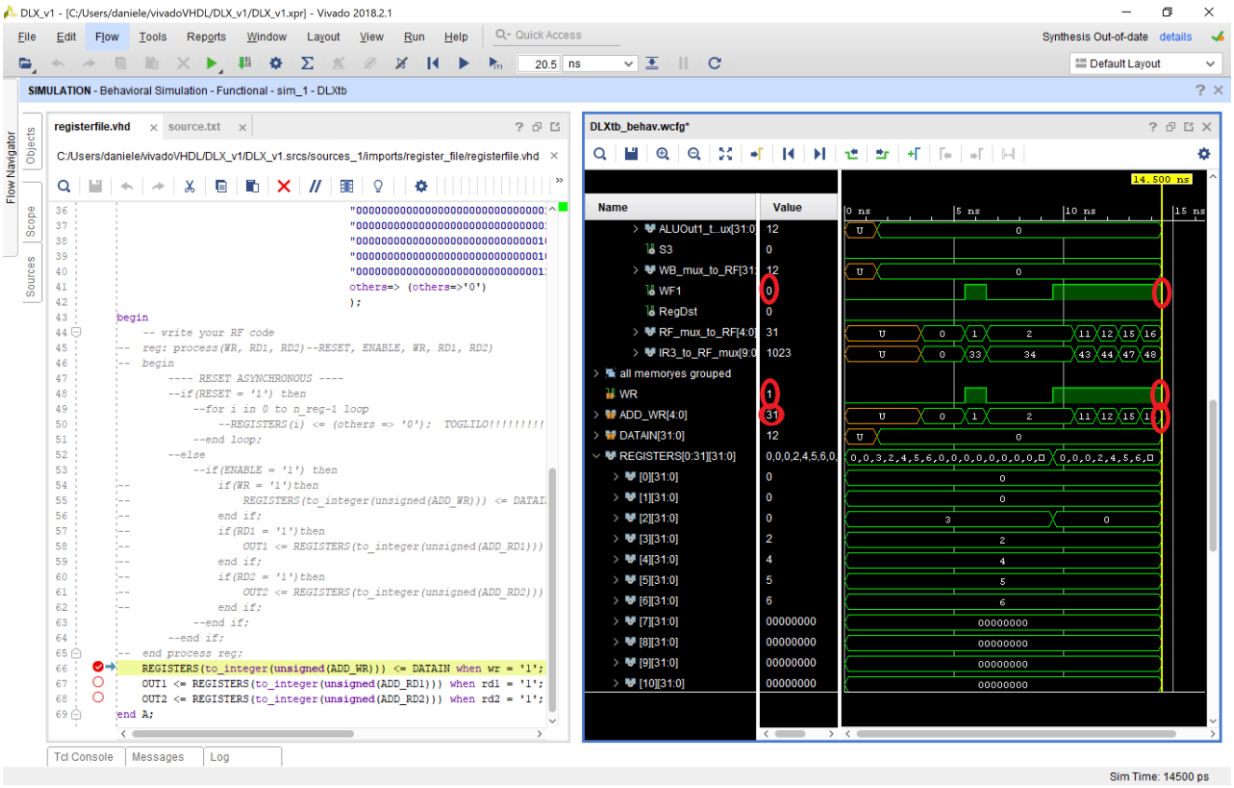


Here we can see *RegDest* that is going to be high, so the data to be written is going to change, but for a small delta time the address line goes low instead of remaining high, so we obtain a wrong writing to the R0 register.

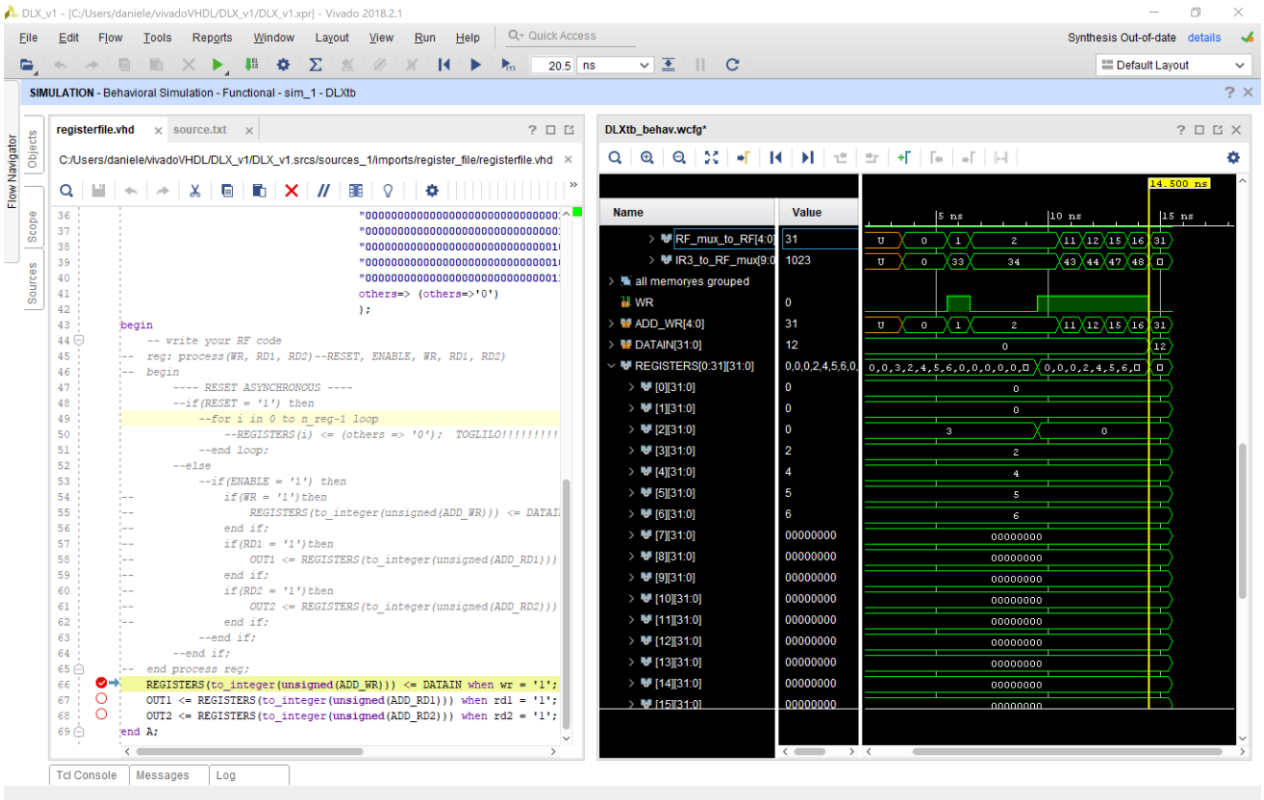


Unfortunately, when we step ahead with the simulation the waveform DOES NOT display the delta time that causes the glitch and *RF_mux_to_RF* seems to remain at value 1 but, now we know, it is not true.

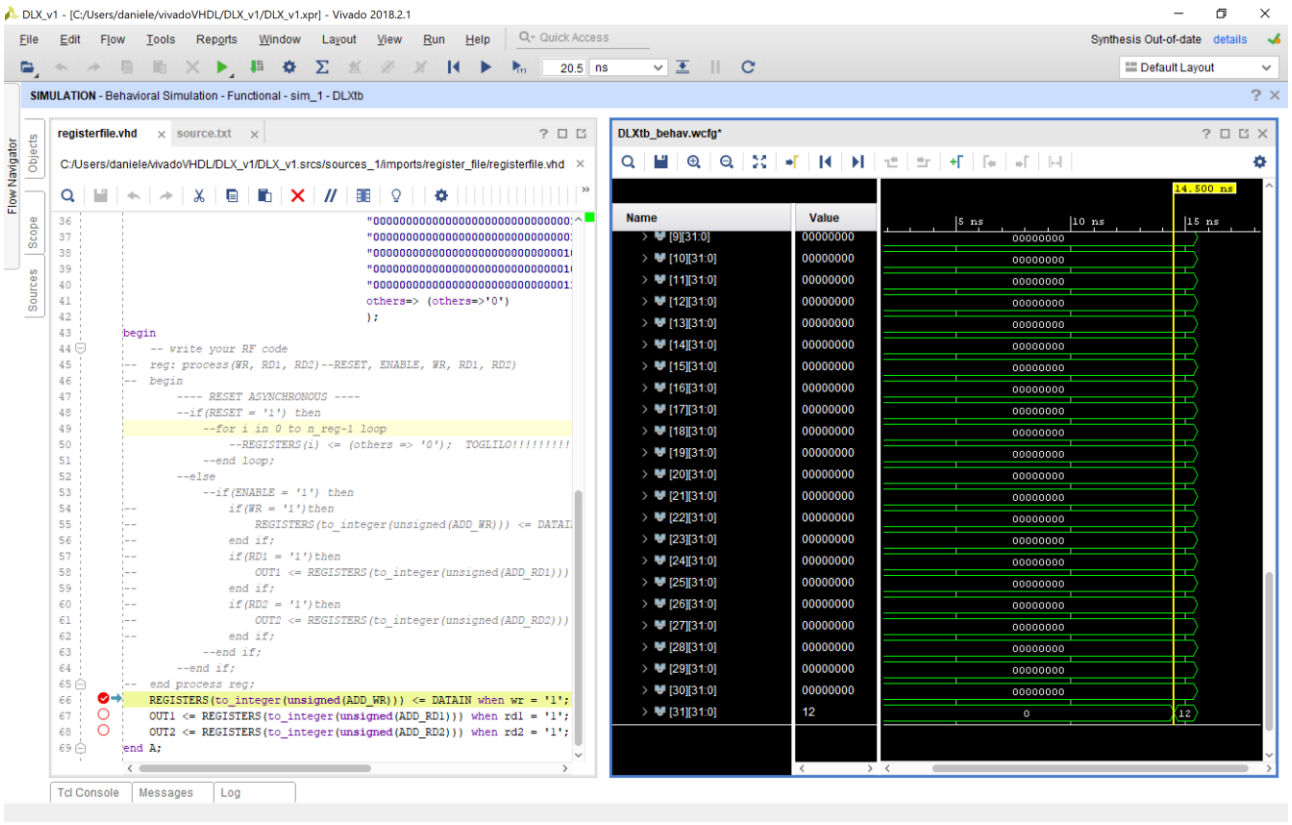
Here we show the last glitch:



The signal *WF1* from the control unit goes low but this signal variation is not propagated in time to the register file's *WR* signal that remains high for a delta time. So *ADD_WR* changes and the *DATAIN* value is inferred into the register file wrongly.



Even in this case, going ahead with the simulation the waveform does not show the delta time that caused the glitch.



So, we obtained a wrong value inside R31 register.

CODE ORGANIZATION: the Datapath is described in a structural way. Only those components like the sign extender and the "is equal to 0?" unit are described in a behavioral way but only because we are sure that the synthesis tool will generate a simple combinational circuit for those components. So it is not necessary to specify how implement these modules at low level.

All the interconnection signals are called with a descriptive name, so it is easy to understand, having the picture of the entire DLX structure, what they connect. For instance:

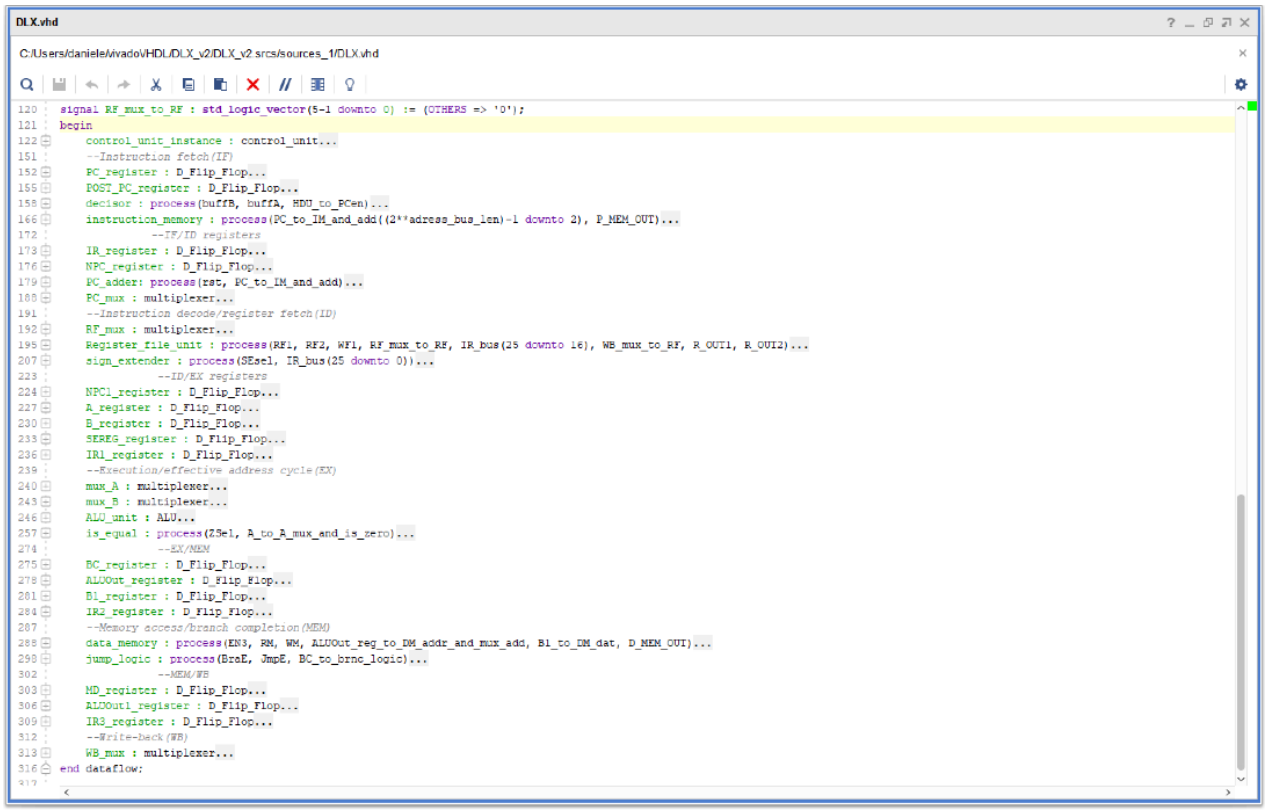
"RF_mux_to_RF" connects the output of the multiplexer in the *Instruction Decode* (ID) stage to the register file.

Descriptive comments are also provided to identify what each stage contains.

For instance:

"--Instruction fetch(IF)" describes what is inside that specific stage. It also contains *"--IF/ID registers"* that defines the pipeline registers contained in the IF stage before passing to the next one: the ID stage.

Here is an image that shows what told above:



Here, instead we can se the entity of the DLX project:

```
entity DLX is
    generic(
        --PROGRAM & DATA MEMORY GENERICS
        nbit_addr: integer := 32;
        nbit_cells: integer := 32;
        --REGISTERFILE GENERICS
        nbit_reg_addr: integer := 5;
            nbit_reg: integer := 32
       );
    port(
        clk, rst : in std_logic;
        --PROGRAM MEMORY SIGNALS
        P_ADD_RD :     OUT std_logic_vector(nbit_addr-1 downto 0);
        P_MEM_OUT :    IN std_logic_vector(nbit_cells-1 downto 0);
         --REGISTERFILE SIGNALS
        R_RD1:         OUT std_logic;
        R_RD2:         OUT std_logic;
        R_WR:          OUT std_logic;
        R_ADD_WR:      OUT std_logic_vector(nbit_reg_addr-1 downto 0);
        R_ADD_RD1:     OUT std_logic_vector(nbit_reg_addr-1 downto 0);
        R_ADD_RD2:     OUT std_logic_vector(nbit_reg_addr-1 downto 0);
        R_DATAIN:      OUT std_logic_vector(nbit_reg-1 downto 0);
        R_OUT1:        IN std_logic_vector(nbit_reg-1 downto 0);
        R_OUT2:        IN std_logic_vector(nbit_reg-1 downto 0);
        --DATA MEMORY SIGNALS
        D_EN:          OUT std_logic;
        D_RD:          OUT std_logic;
        D_WR:          OUT std_logic;
        D_ADD_WR:      OUT std_logic_vector(nbit_addr-1 downto 0);
        D_ADD_RD:      OUT std_logic_vector(nbit_addr-1 downto 0);
        D_MEM_DATAIN: OUT std_logic_vector(nbit_cells-1 downto 0);
        D_MEM_OUT:     IN std_logic_vector(nbit_cells-1 downto 0)
    );
end DLX;
```

At the beginning we thought to make a generic processor in the sense that could both work with 32- or 64-bit data length. That is the reason why we put generic variables at the beginning of the DLX entity, but we had not time enough to finish and test this feature.

Comments in `port` statement indicates what signal must be connected to a certain kind of memory.

Internally, trying to maintain descriptive variable names, we connected entity memory signals to other signals with different names, so we can easily know where the signal comes from and where it goes to like in the example below. We used a `process` to make this interconnection and tried to compact and label the group of signals all in the same place. Of course, a process is not needed to do this task. We put it only for readability reasons. The same thing must be applied to the memory signals that, in this way, are repeated each one at least twice. We are quite sure that the synthetize tool will not create two separate wires connected one another.

Here is an example of what said for the register file process:

```
Register_file_unit : process(RF1, RF2, WF1, RF_mux_to_RF, IR_bus(25 downto 16), WB_mux_to_RF,
R_OUT1, R_OUT2)
begin
    R_RD1 <= RF1;
    R_RD2 <= RF2;
    R_WR <= WF1;
    R_ADD_WR <= RF_mux_to_RF;    --destination register address
    R_ADD_RD1 <= IR_bus(25 downto 21);   --source register 1 address
    R_ADD_RD2 <= IR_bus(20 downto 16);   --source register 2 address
    R_DATAIN <= WB_mux_to_RF;
    RF_to_A <= R_OUT1;
    RF_to_B <= R_OUT2;
end process;
```

SYNTHESIS: first of all we synthetized the dlx without any constraint and we saved the report for area and timing. Then we forced a clock signal with clock period at 2.0 and, again, we saved the timing report. Finally we set the maximum delay to 2.0 and we saved the timing report.

What follows is out final DLX block diagram. Unfortunately, we had no time to make a digital one: