

School of Sciences and Engineering

Fall 2021

CSCE330102 - Computer Architecture

Project 1 Report: Final Submission

Abdelrahman Said

ID: 900192935

Mostafa Ibrahim

ID:900192397

# Project Description

In the final version of the project, we transformed the single-cycle RISC-V processor implemented in milestone 1 into its pipelined version, supporting all the RV32I instructions. A major change that took place was replacing the instructions memory and the data memory with one single-ported memory placed in the IF stage. The usage of the clock was changed to the equivalent of stalling a cycle after each instruction to avoid the structural hazards introduced by the single memory approach. In addition, all other possible hazards were taken into consideration and handled.

# Technical Summary

The data path that was created consists of 20 modules, including the main data path module. Most of the modules from the single-cycle implementation were used without modifications. The following modules are the ones that we created or modified to implement the project.

## Modules

**Memory**

A single-ported byte addressable memory that replaced both the data and instruction memories. This was done by dividing the memory so that instructions are in the first half of the memory and data is in the other. The clock signal determines whether to read instructions or read/write in the data memory; at the positive edge instructions are accessed and at the negative edge data is accessed. To fetch data from the lower half of the memory, an offset parameter is added to the given address.

#### Inputs

* **clk:** The clock signal was used as a selecting signal for the memory to choose between accessing instructions or data.
* **MemRead/MemWrite:** Control signals coming from the execution stage specifying whether to read from the memory or write to it.
* **func3:** Used to differentiate between different load (lw/lh/lhu/lb) and store (sw/sh/sb) instructions in the memory.
* **addr:** The address to be read from/written to. The number of bits is determined according to the memory size. The size of the address is log2(memory size).
* **data\_in:** This is the input data that needs to be written in the memory in the case of memWrite = 1 and if the clock is on the negative edge.

#### Outputs

**[31:0] finalOutput**: This is the memory output and it could be data or instructions depending on the clock edge.

### **Forwarding Unit**

The purpose of the forwarding unit is to forward needed data from the execution or the memory stages to deal with RAW dependencies. The forwarding mechanism works by comparing the source registers’ numbers from the decoding stage to the destination registers’ numbers in the execution and memory stages. In addition, we check whether we need to write tin the register file in the first place or not, and that that destination register is not the zero register. The Forwarding Unit then outputs 2 signals (ForwardA & ForwardB) used as selectors to what will be the operands going into the ALU.

Nonetheless, it is worth mentioning that since we are stalling one cycle after each instruction anyway, the only type of RAW dependency should be dealing with is when we need to forward from the memory stage, in other words, the dependency is present in the instruction before the preceding one. However, our module handles both cases, since we used the module that was implemented in the lab without changing it.

### **Register\_32bit**

The Modifications to the Register module were minor. The module was parametrized with a default value of 32 bits. These registers were used in different sizes to implement the registers needed to pipeline the data path.

**Flushing**

For the flushing, with the clock being used at both of its edges, we only need to flush instructions at the Execution stage, and that is done by selecting whether to propagate the control signals produced by the Control Unit normally or to propagate zeros instead, using the least significant bit of PCSrc signal produced by the Branch Control Unit.

## Block Diagram



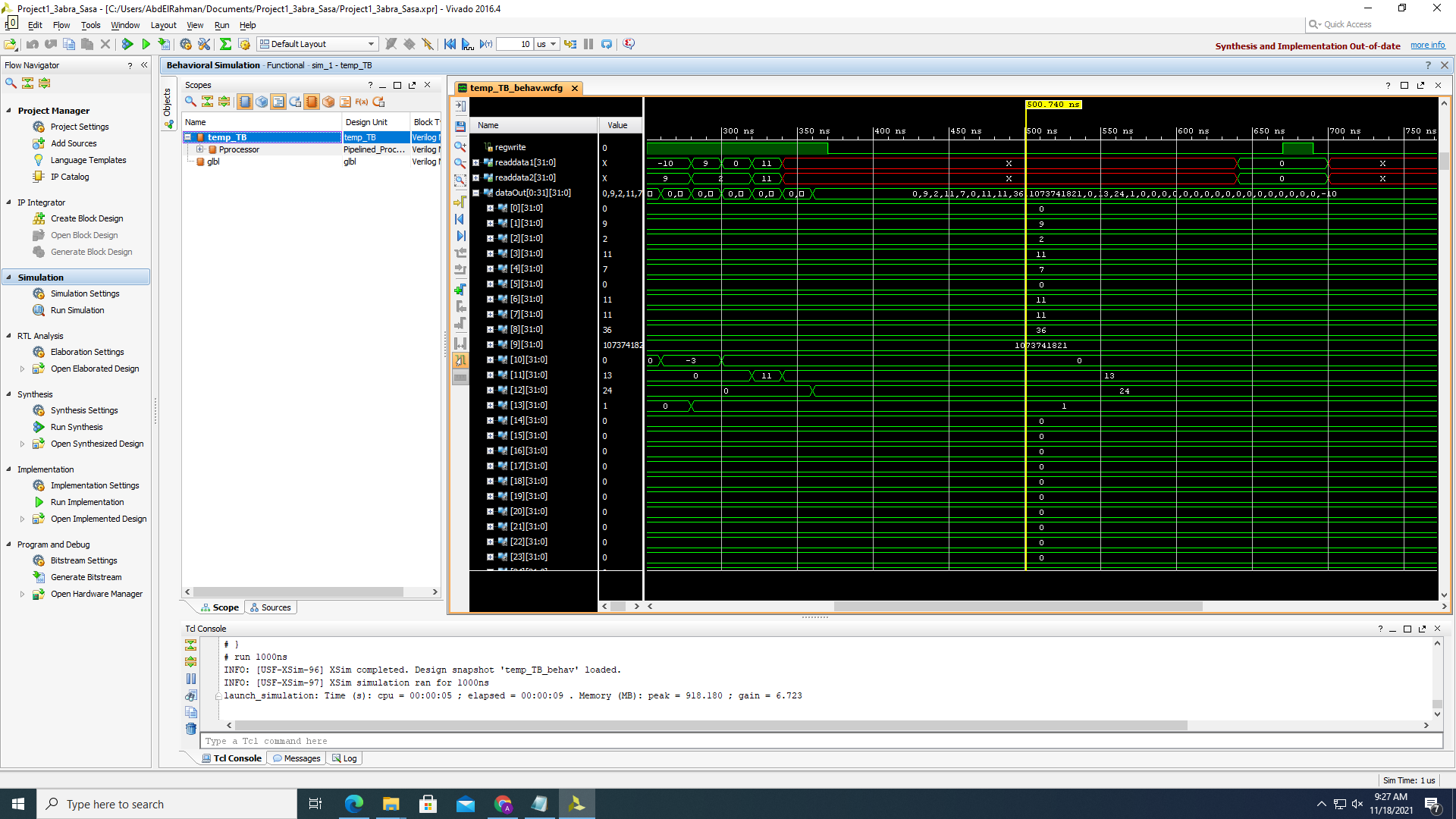
The full data path can be found [here](https://drive.google.com/file/d/1EKoZHdOfqFfHtnrniRxzuV4X2B5IBRsl/view). In this link, the editable data path can be viewed if it’s not visible in the screenshot due to dimensions constraints.

# Testing Results

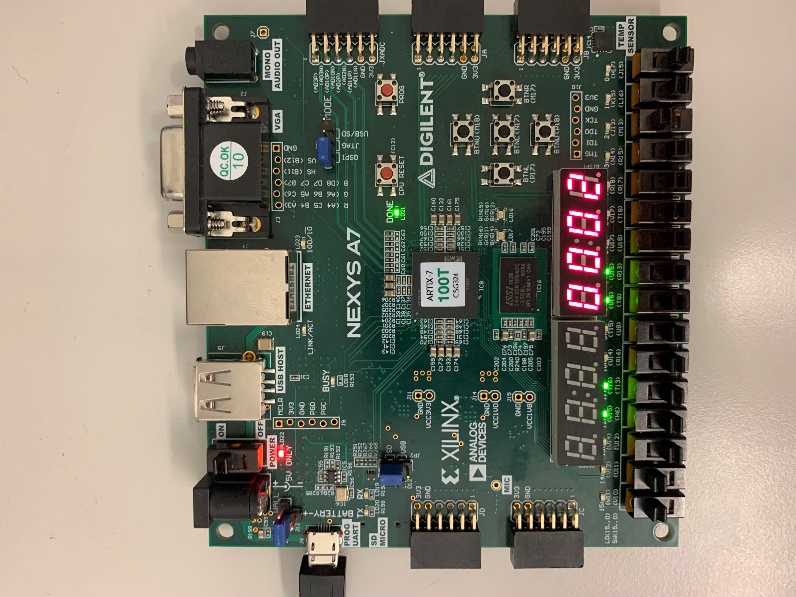
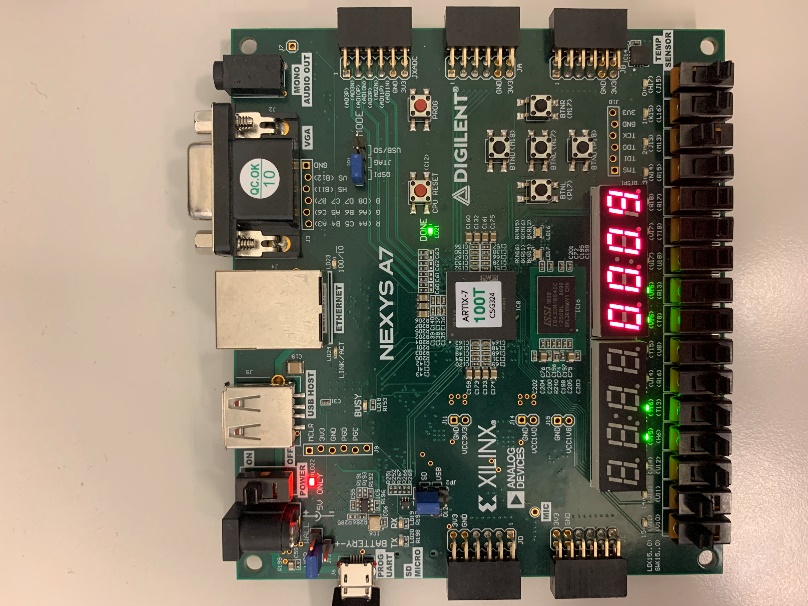
Please note that the controlling switches on the board are as follows. The leftmost switch controls the clock edge. The 4 following switched are used to control the display on the 4-digit seven-segment display. The rightmost switch is the reset. The 4 switches that follow it controls the small LEDs Display.

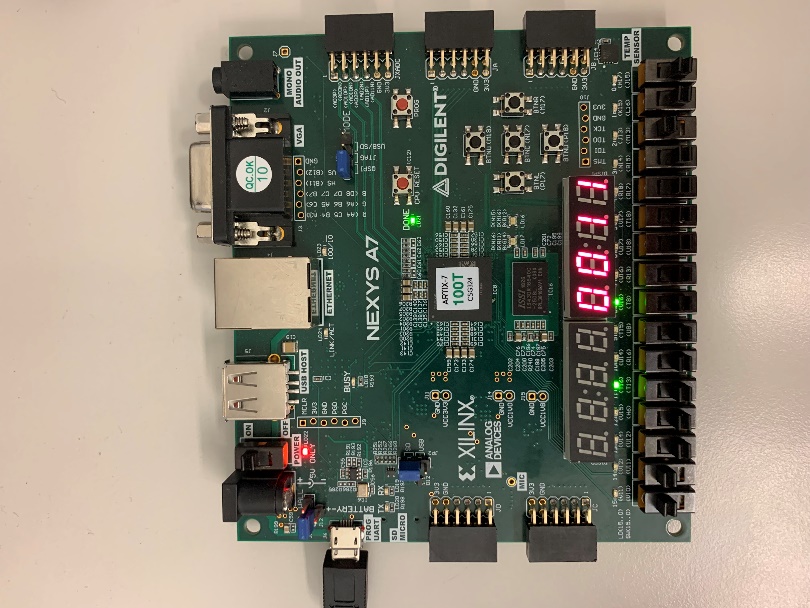
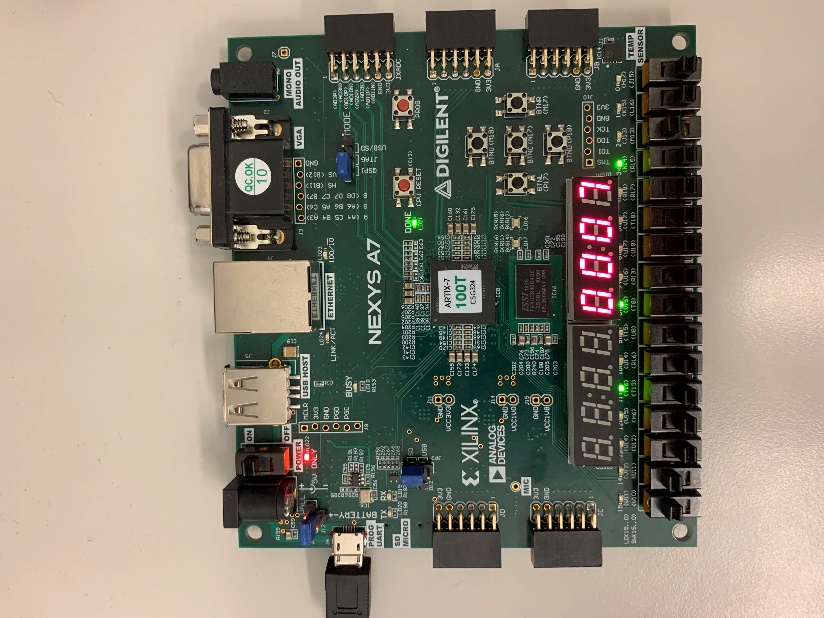
## R-Format

The simulation result shown below shows the results of the operations in the test case “RType” being written correctly to the register file.

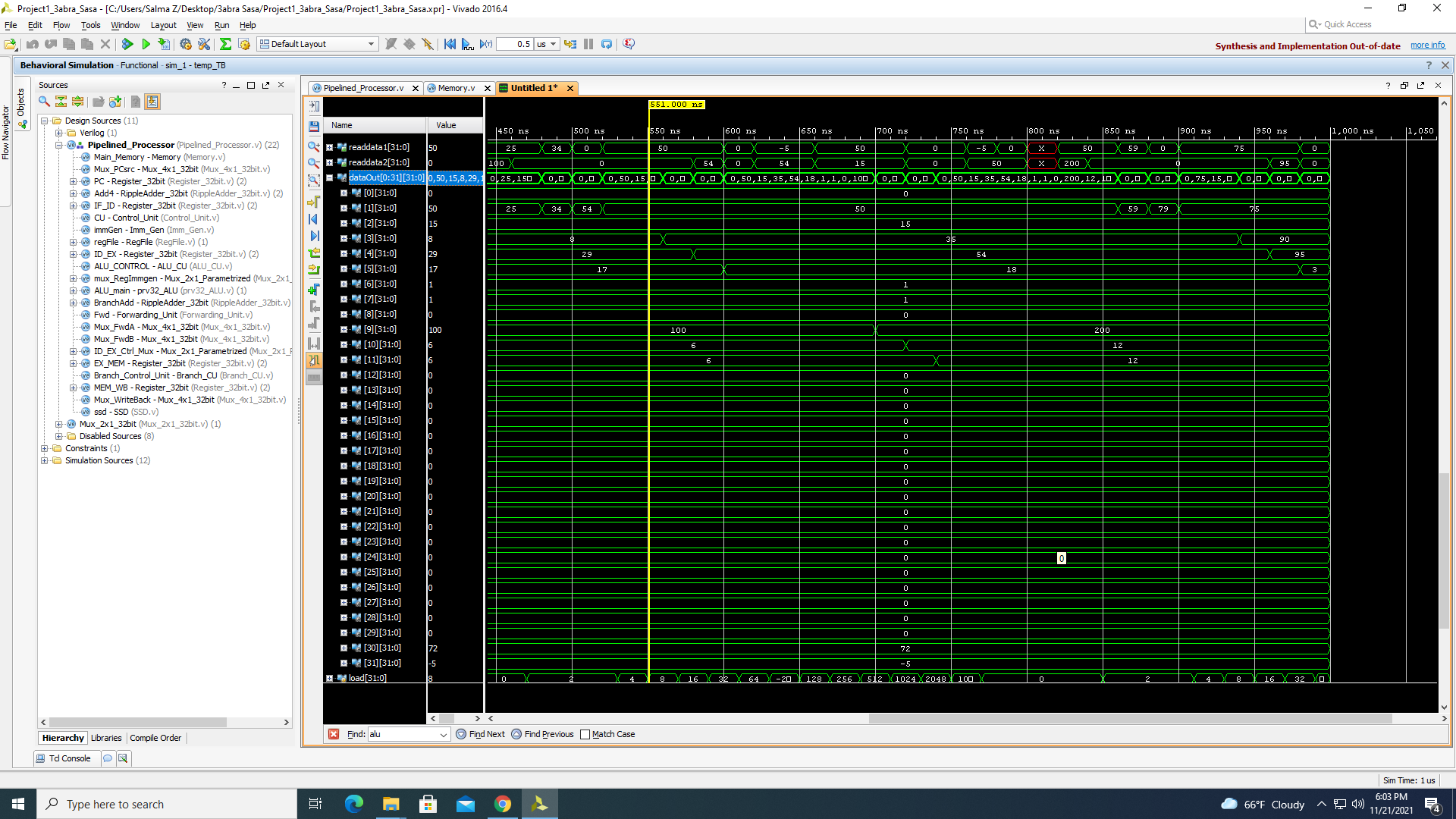


The following pictures of the FPGA board shows the ALU result of the first 5 instructions with the LED lights displaying the control signals in the following order:

{2'b00, Branch, MemRead, PCsrc, MemWrite, ALUSrc, RegWrite, [1:0]WriteBack, [1:0]ALUOp, ALUsel, JALRflag, JALflag, zf}

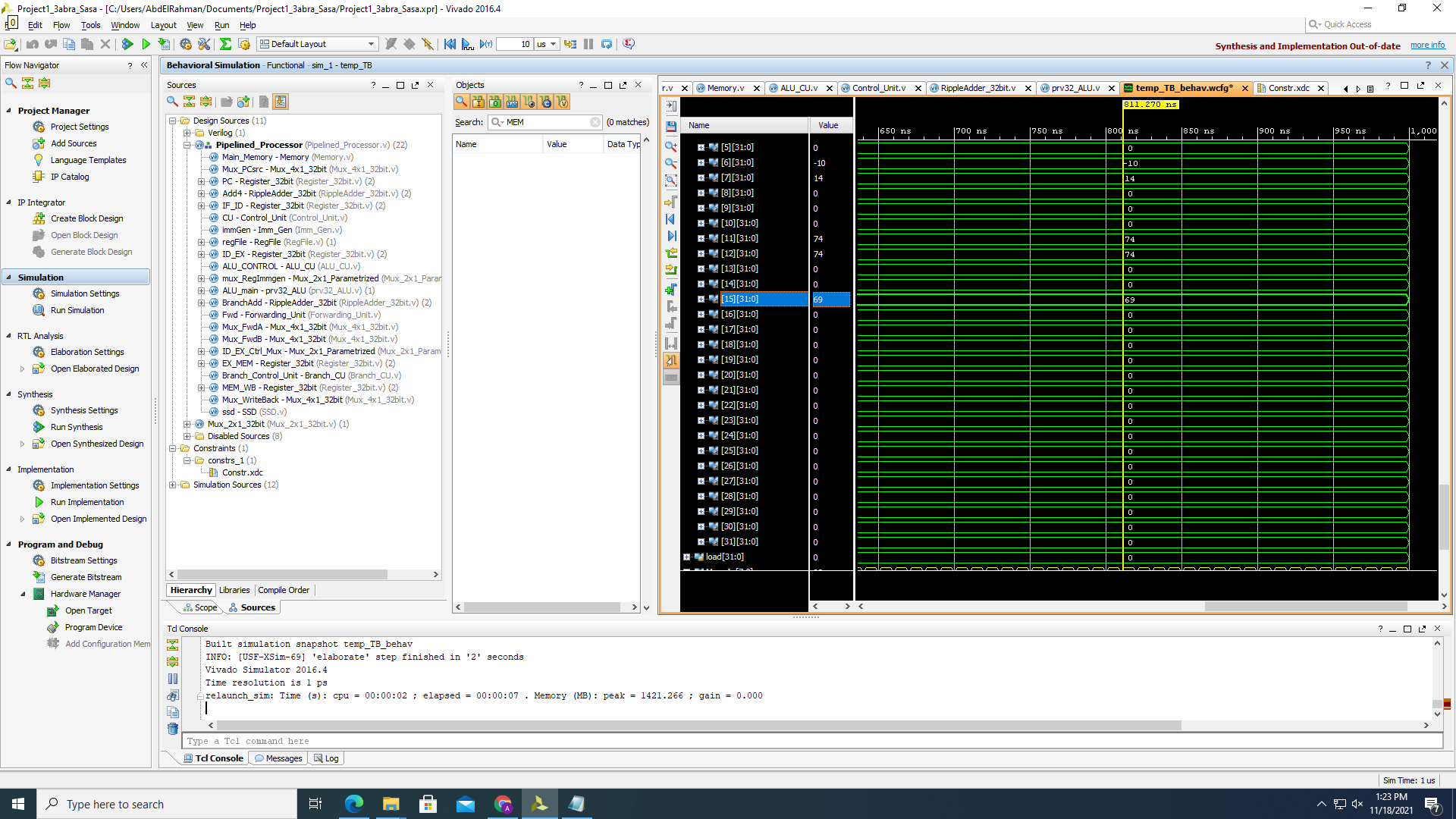
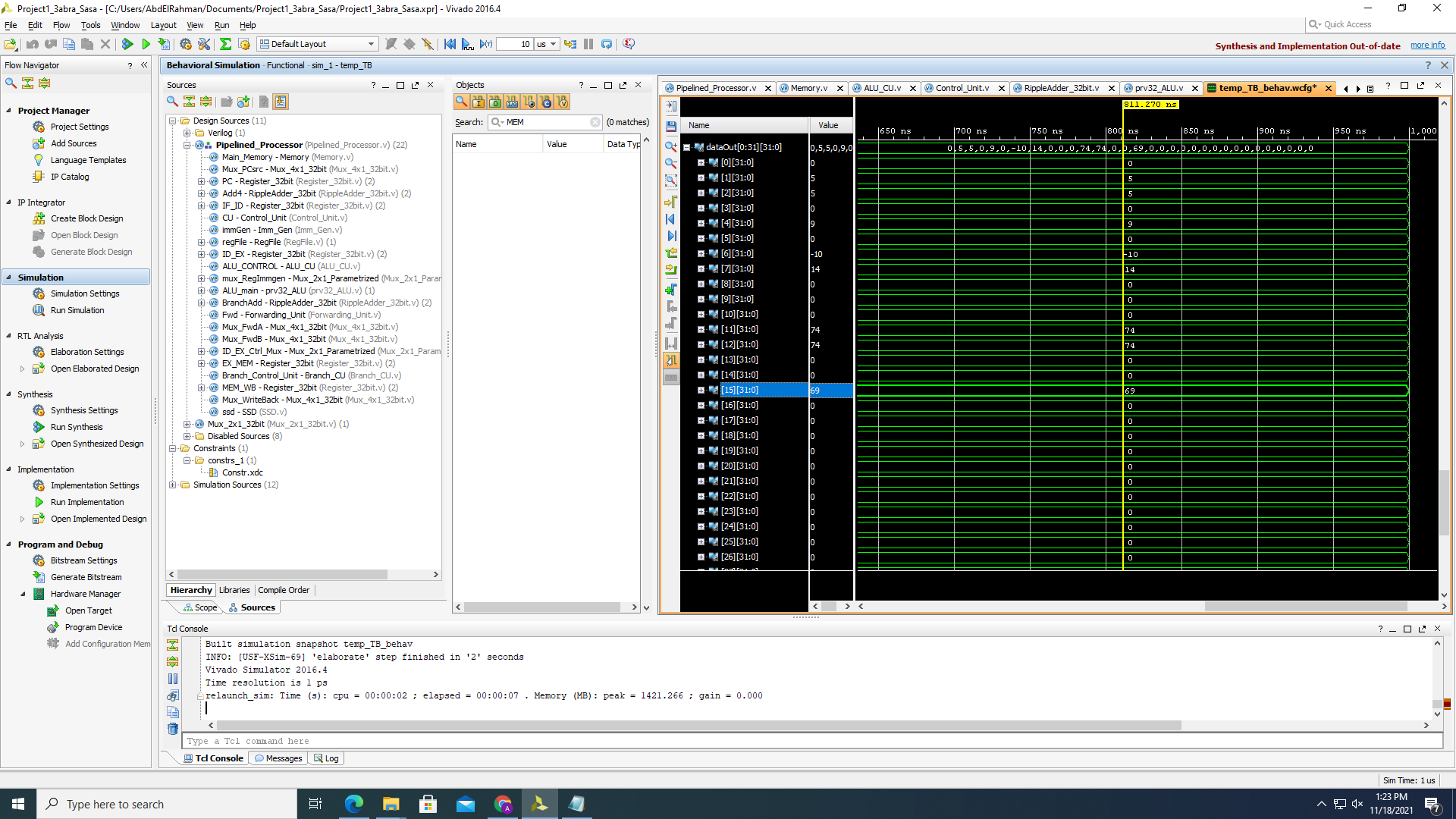


## I & J-Formats

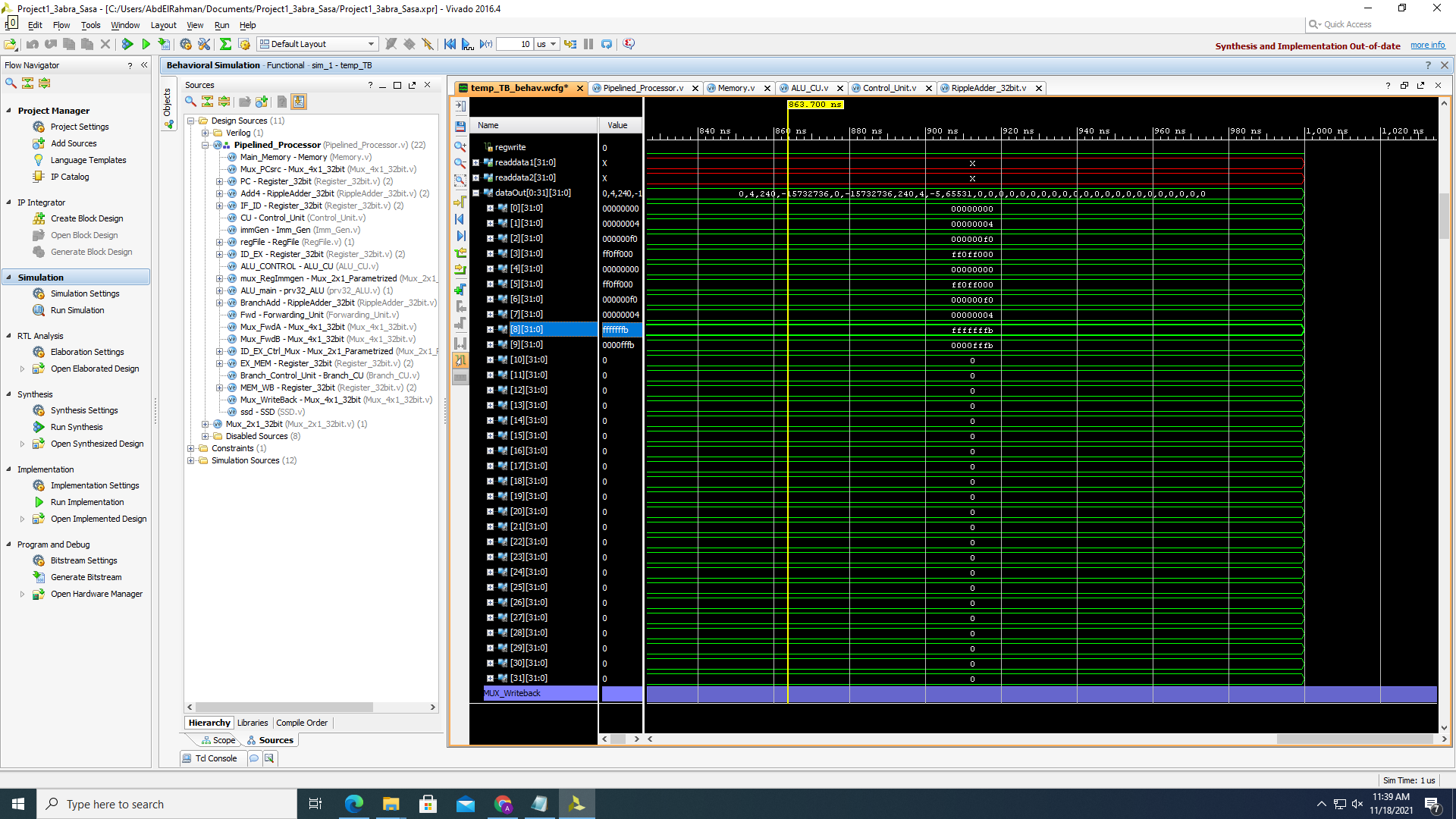
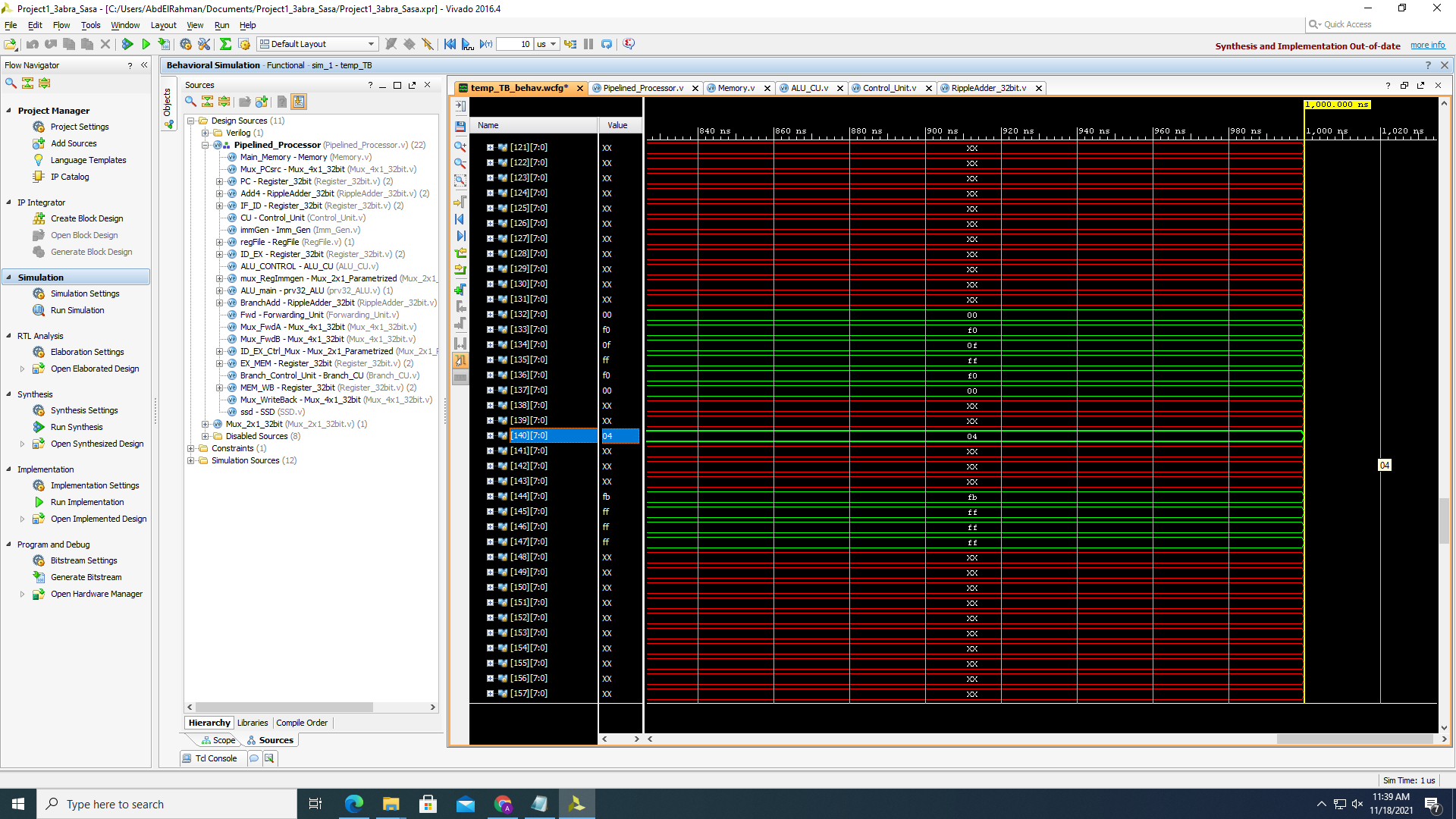
Note that the file for this test case is named “Itype\_Jump”.

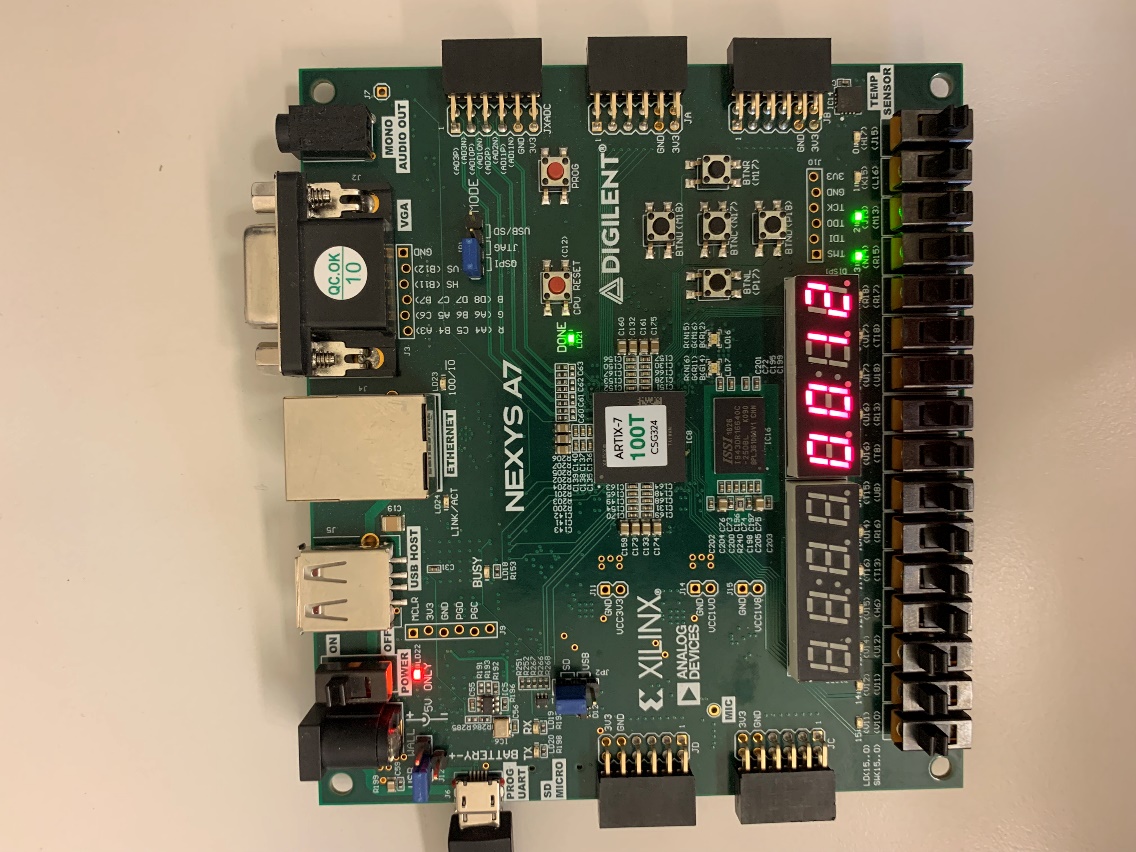
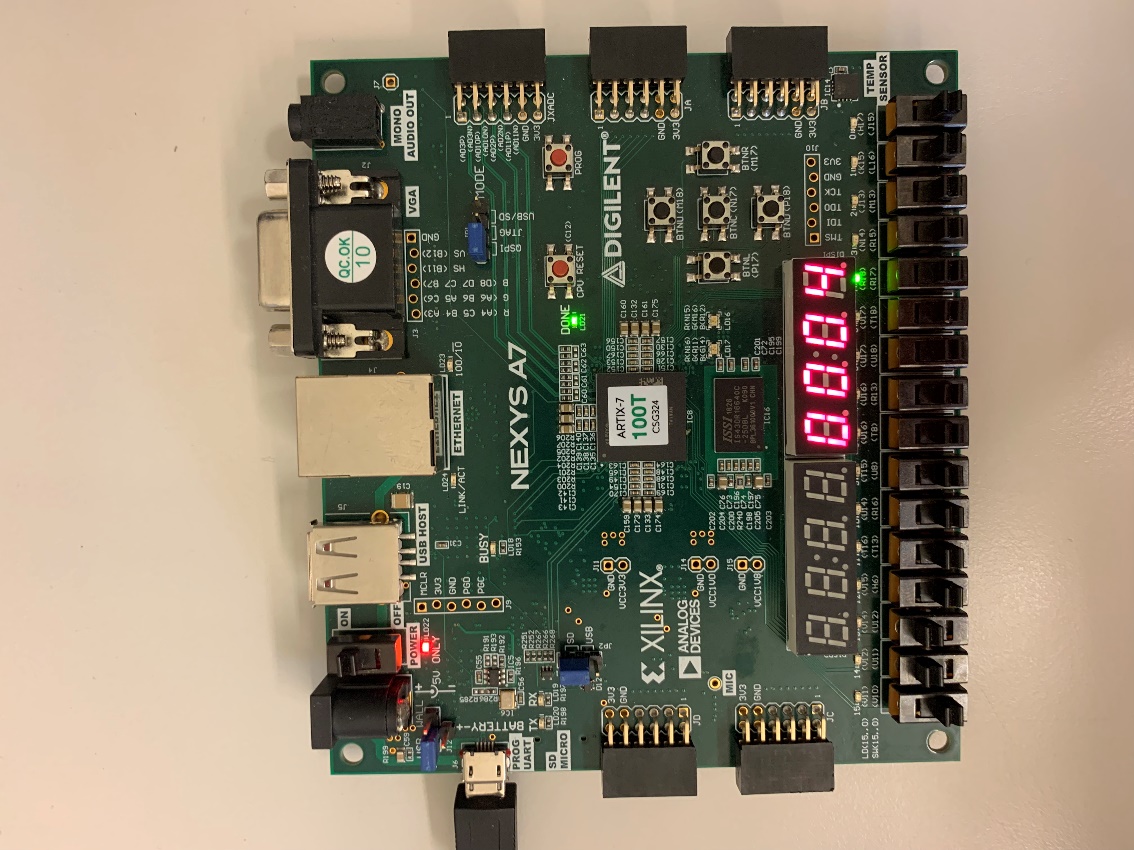
## Branches

The screenshots below show the data written in the register file as a result of branch paths and it shows that the control hazards were bypassed and the flushing is working correctly.



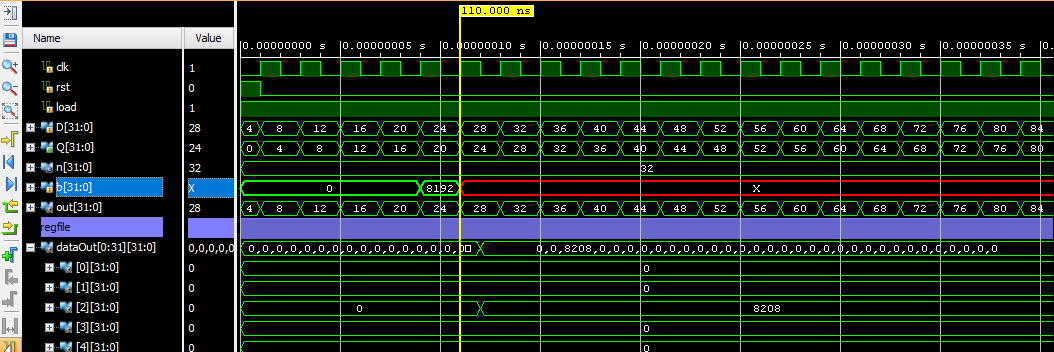
## Load/Store

In this test case we tested all the load and store instructions as well as the instruction LUI. The register file and the memory have shown correct results. This test case is named “LoadStore”

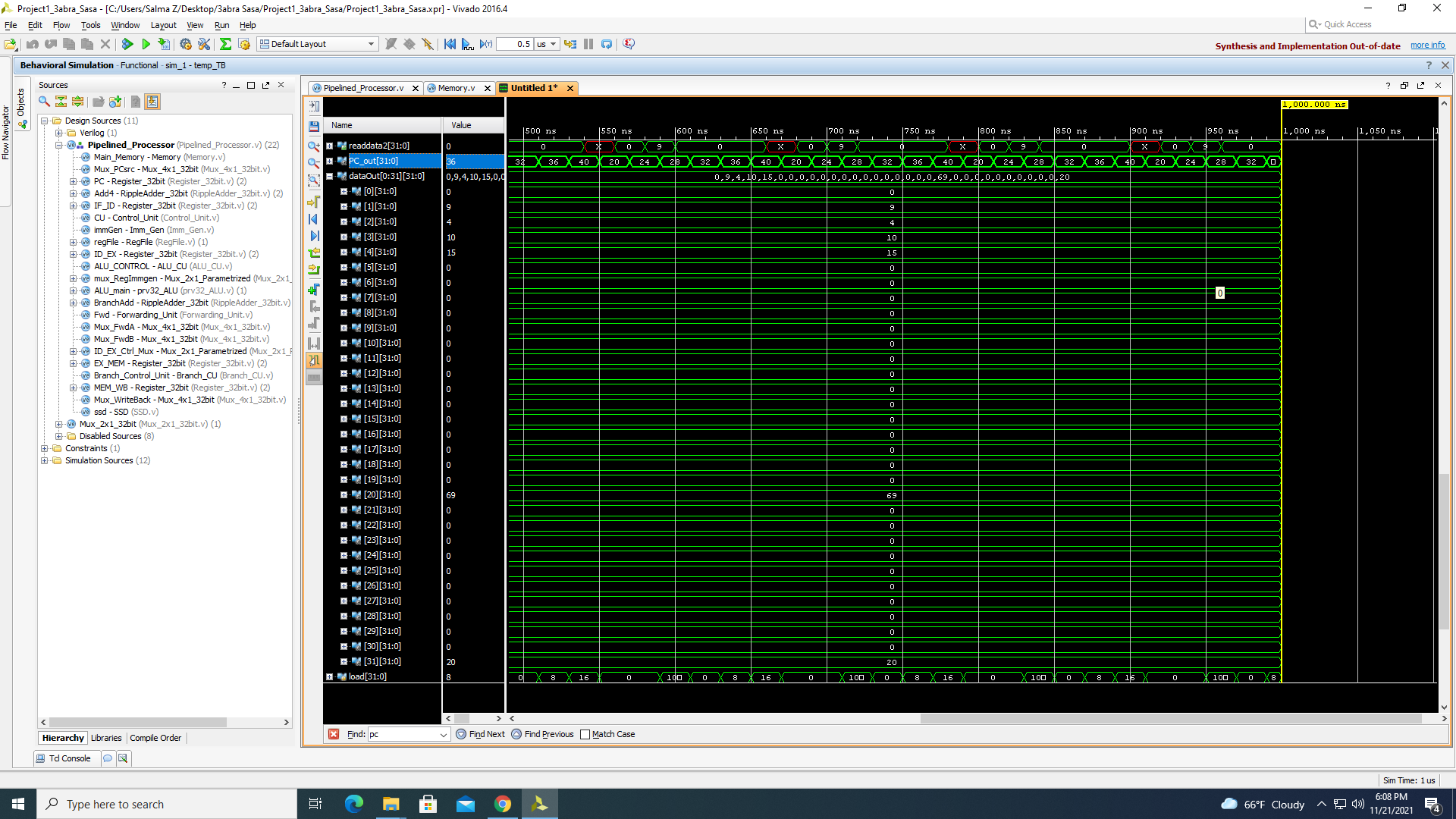
We displayed the memory input in the FPGA which showed the PC in the positive clock cycle and data address in the negative clock cycle. The LEDs are showing the program counter. The left photo shows the PC in the positive edge while the picture on the right shows the data address input.

## AUIPC

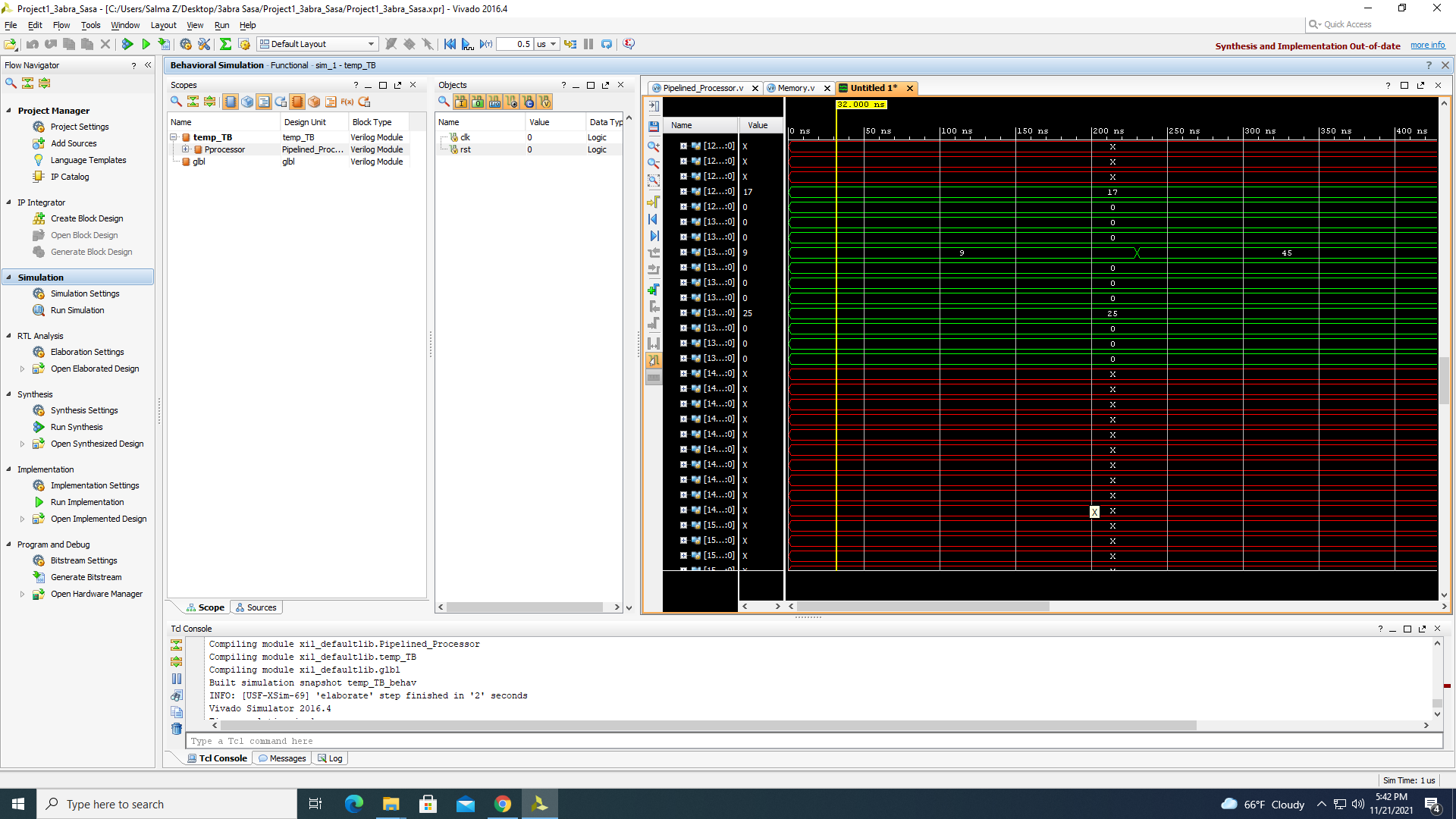
The instruction is working correctly it took the value 2 and loaded it in the upper 20 bits of the register then added the pc value to it.



## JALR



## LU Hazards

This test case shows some situations of load use hazards found in the file “Load-Use” it loaded the value 17 first and then forwarded the value to the ALU and executed the addi instruction that is afterwards that used x1 as both a source and a destination register. 