Name: Abdullah Khan

Roll No. : 111801001

Course Name: CS2160 - Computer Organization Lab

**POST LAB REPORT - 1**horizontal line

# **Exp 2 & 3: Datapath for RISC-V ISA** Completed on: 28 Jan’20

**Problem Statement**

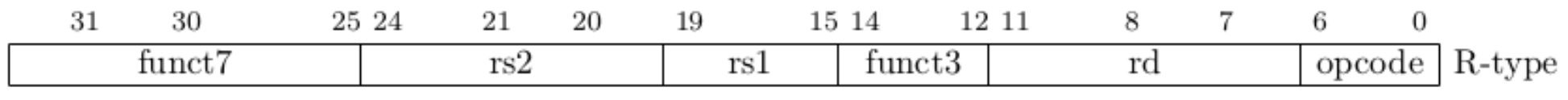
Design the data path for “ *add* ”, “ *sub* ” and “ *add i* ” instructions for RISC-V processor. The data path consists of instruction memory, register file and ALU. We have to write modules describing each of the hardware component of the data path.

**Theory**

**Datapath:** is a collection of functional units such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses. Along with the [control unit](https://en.wikipedia.org/wiki/Control_unit), it composes the central processing unit (CPU). A larger datapath can be made by joining more than one number of datapaths using a multiplexer.

**RISC-V Instructions:** RISC-V instructions are 32-bits or 4-bytes long with the each instruction subdivided into different fields according to which we as humans differentiate between different types of instructions such as R-type, I-type etc. The formats are distinguished by the values in the opcode field: each format is assigned a distinct set of opcode values in the first field (opcode) so that the hardware knows how to treat the rest of the instruction.

**R-type Instructions:** R-type instructions uses 3 registers as a source of data to be computed. Its has various fields namely func7, rs2, rs1, func3, rd, opcode. Each field is viewed as its own unsigned int. Opcode partially specifies the operation to be performed along with func7 and func3 whereas rs1,rs2 are the two source registers where our desired data is stored and rd is the address of the destination register in the REGISTER FILE.



Here is the meaning of each name of the fields in RISC-V instructions:

**opcode**: Basic operation of the instruction and this abbreviation is its traditional name.

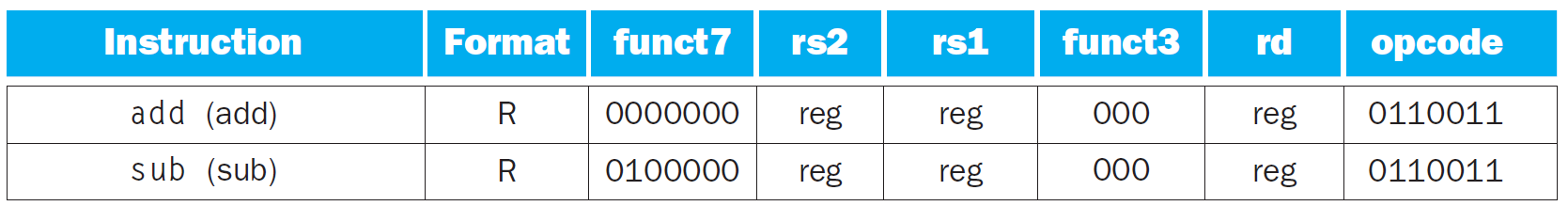
**rd**: The register destination operand. It gets the result of the operation.

**funct3**: An additional opcode field.

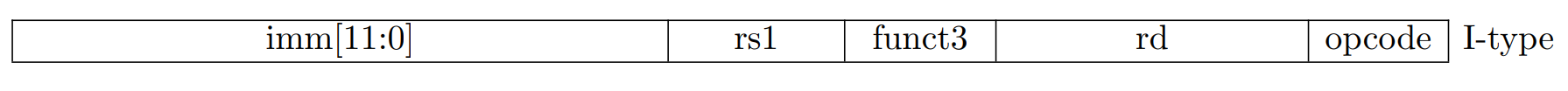
**rs1**: The first register source operand.

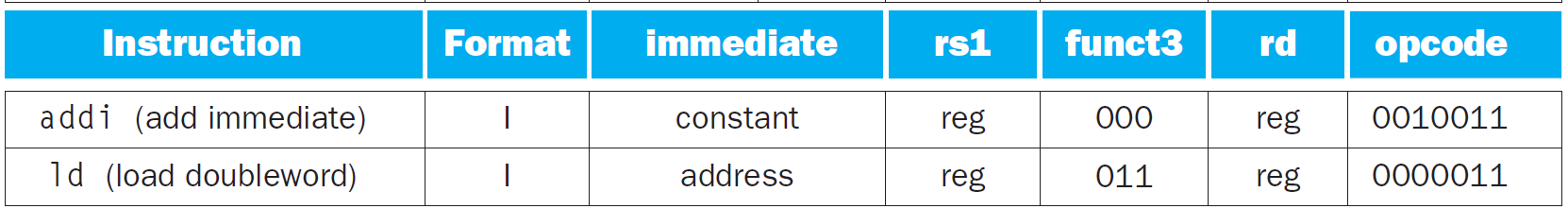
**rs2**: The second register source operand.

**funct7**: An additional opcode field.



**I-type Instructions:** I-type instructions uses 2 registers one as a source of data and another for the storage of computed output. It is different from the R-type instructions in the lack of func7 field and rs2 field, which are replaced by an ‘immediate’ field which represents a constant value or an address(in case of load operations) as the second operand. All computations are done in words, so 12-bit immediate must be extended to 32 bits where the sign bit is extended. I-type instructions can be classified into two types; arithmetic and load instructions.





**Instruction Memory:** is a part of the main memory designated to specifically store instructions to be performed in machine language. We will write a module **IM** that reads the instructions stored in the instruction memory ( text file written in binary in our case ) every 4 clock cycles.

**Register File:** is an array of [processor registers](https://en.wikipedia.org/wiki/Processor_register) in a central processing unit (CPU). The [instruction set architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture) of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. Thus there are a total of 32, 32-bit registers in RISC-V processors. The registers are named as X0 - X31.

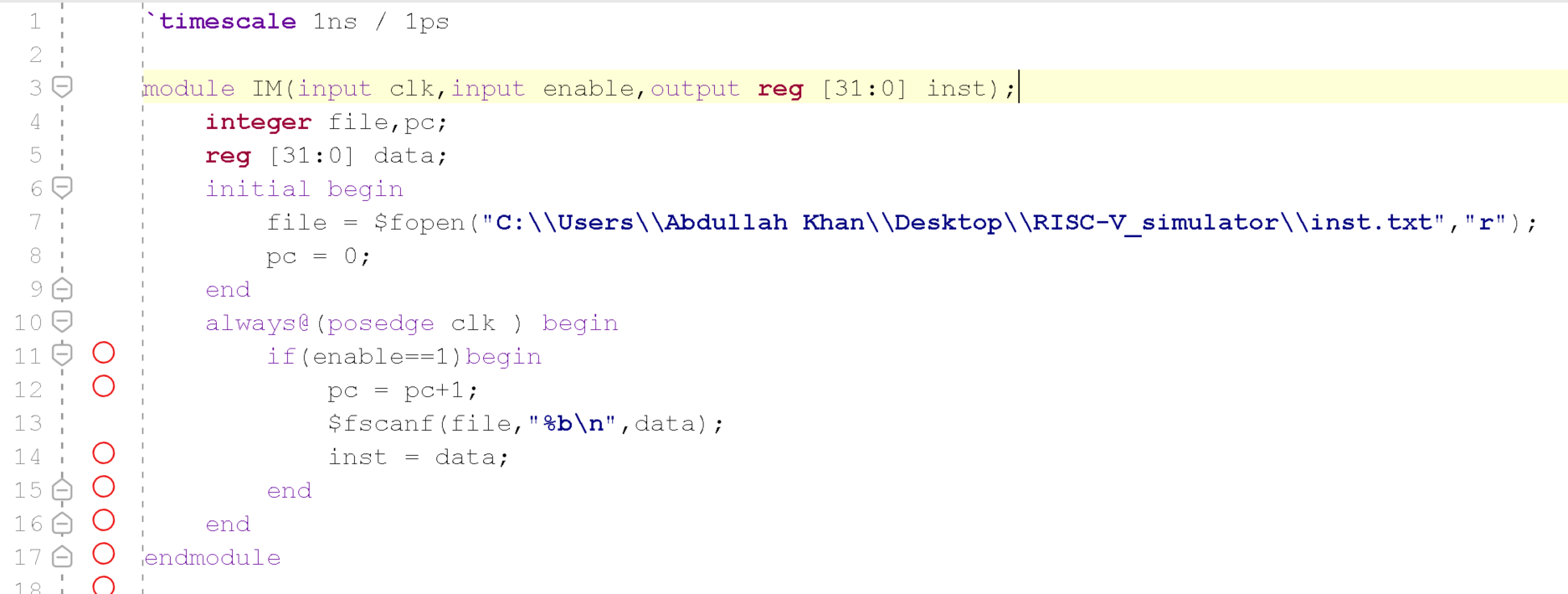
**Decoder Module:** This module will be instantiated inside Instruction memory module to decode each instruction based on RISC-V instruction format to give various fields and decide the ALU function to be performed.

## **Procedure**

* Open Vivado and create a new project, then select zynq7000-clg400 board from the drop-down list.
* Create the instructions.text file to store the instructions and RF.text file to store the state of our registers.
* First code the individual modules as described above and then create a top module that instantiates and controls all other modules.
* Check the datapath and control flow of the top module.
* Create a test-bench for ease of testing and debugging.
* Simulate and check for some basic instructions such as adding 2 registers etc. and verify the result under the simulation tab and in the register file separately.

**Code**

**Instruction Memory Module**



This module reads the instructions from the text file one line at a time whenever enable is HIGH. So we can easily decide at which cycle we want to fetch our new instruction.

**Decoder Module**



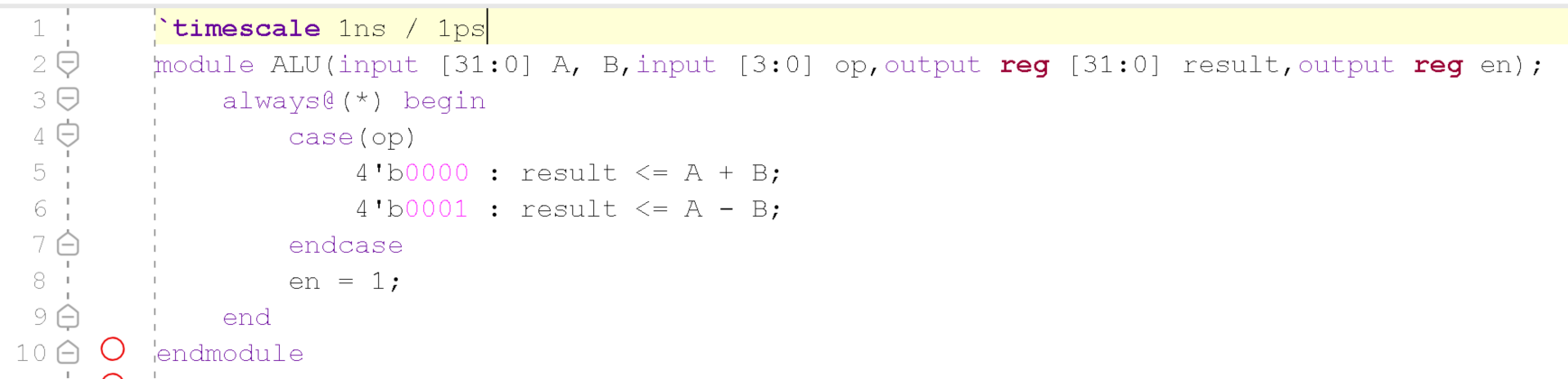
The DECODER module take the instruction as input in the next cycle and segregates the 32 bits into different fields and depending upon the type of instruction return appropriate fields along with the ALU function value.

**Register File Module**

****

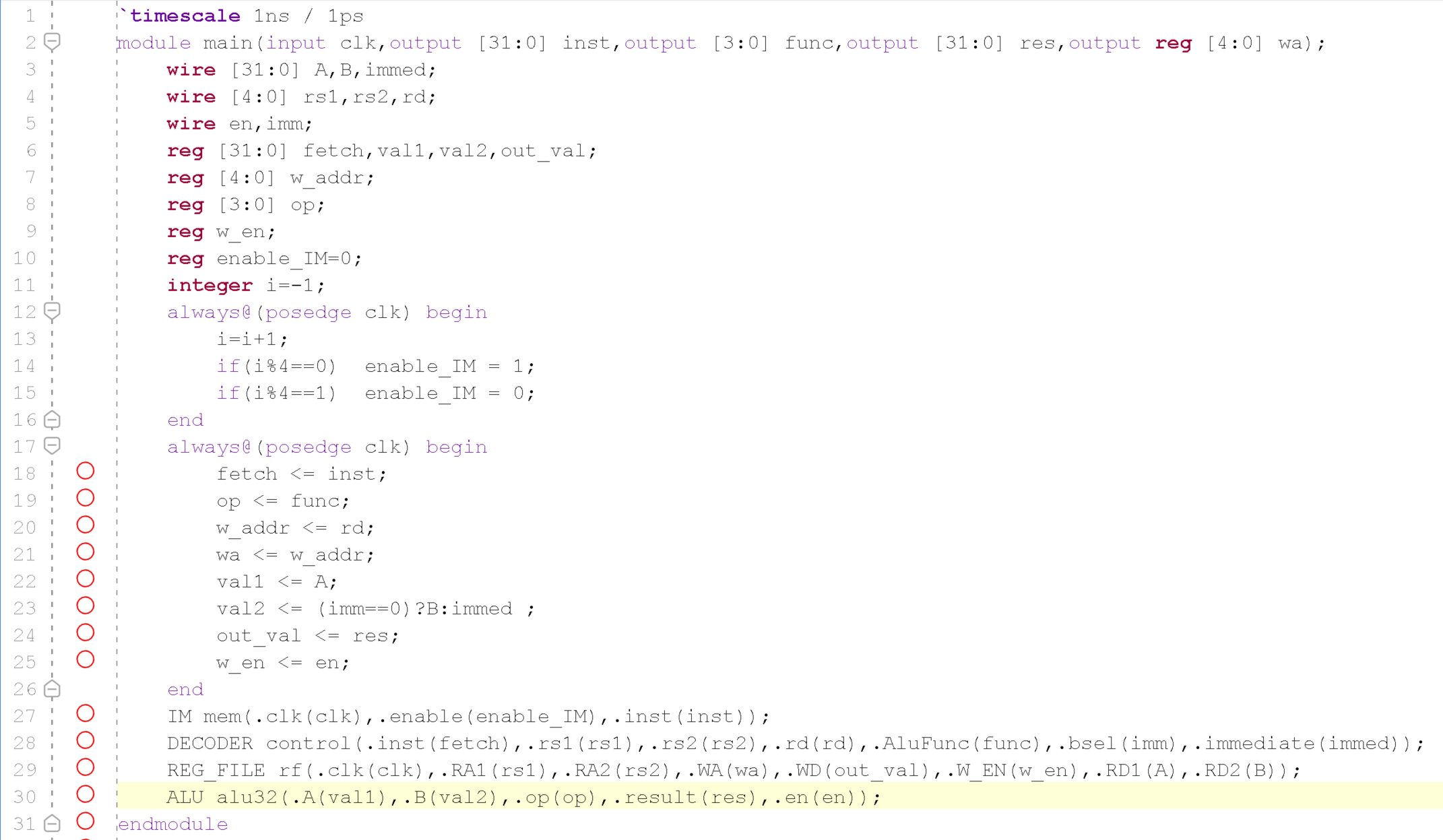
This module reads and write data values from and to given address in the REGISTER\_FILE. The read function is combinational while the write depends on the clock and thus is sequential. Initially, when the module is instantiated it reads the state of all 32, 32-bit registers and stores their value in a matrix, which it later make changes to.

**ALU Module**



This is a very simple implementation of an ALU that has only two operations i.e Add, Sub.

**Top Module**



This top module defines the datapath and control flow of our implementation of a RISC-V simulator. Every output from the instantiated modules gets stored in registers that introduce one clock cycle delay between each step in the data path. Thus in first clock cycle instructions are fetched from the file, the next clock cycle they are decoded and operand values are fetched from the REGISTER\_FILE , the next cycle the operation is executed in the ALU and the result gets stored in another register which is in turn written in the EGISTER\_FILE in the next cycle.

The instructions are currently fetched after every 4 clock cycles with the help of a counter but it is possible to fetch them continuously every cycle without any malfunction.

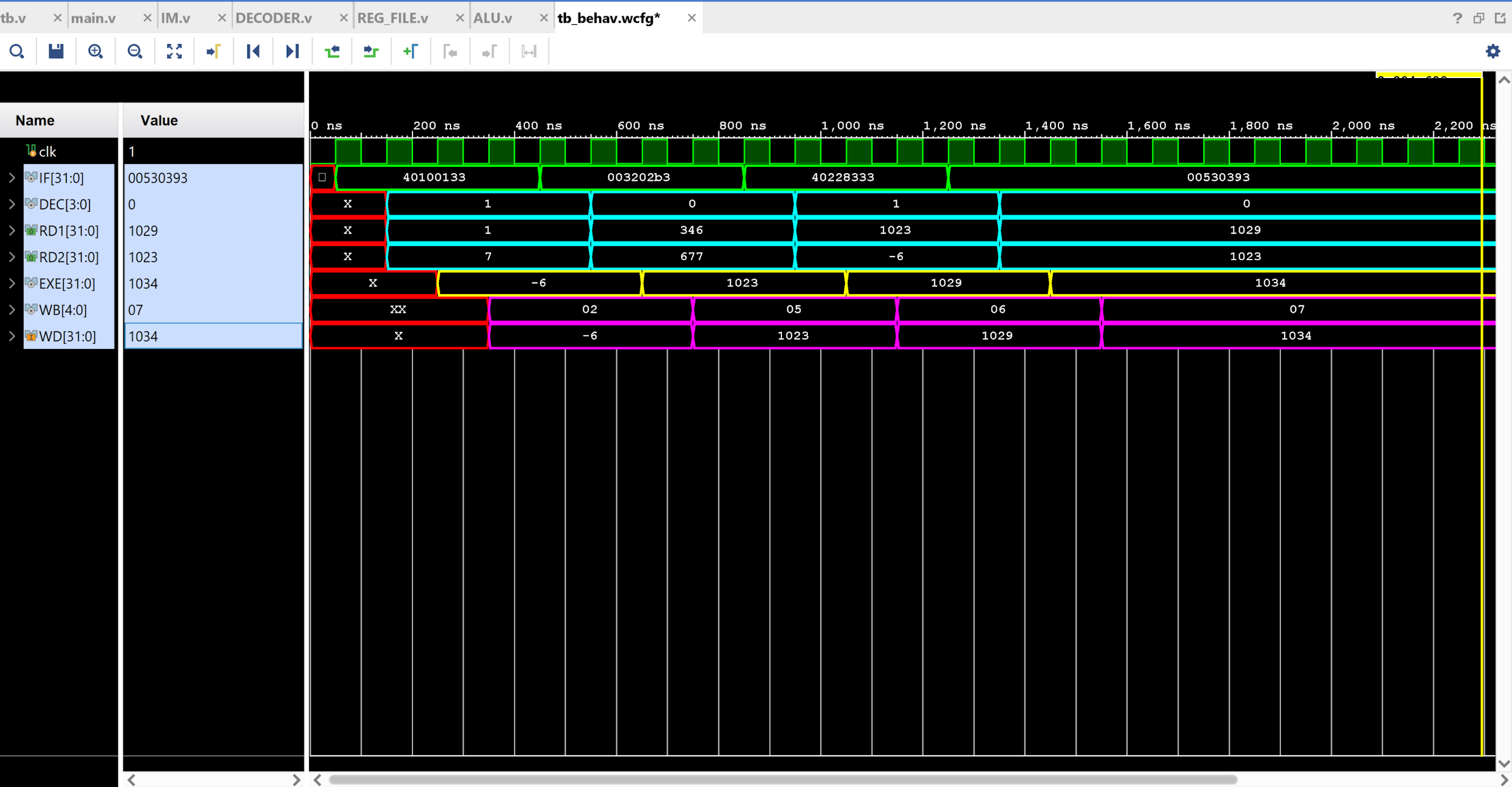
## 

## 

## **Observations**

It was observed that the program was working correctly for the all the inputs given through the test-bench and was verified manually through the simulation tab as well as manually in the REGISTER\_FILE.

Below is a figure that shows the program run for the inputs given in the test-bench (shown above).



**IF** : shows the output of the instruction memory module i.e. the instruction fetched.

**DEC** : shows the output of the Decode Module and the values of source registers rs1, rs2.

EXE : shows the output of the ALU i.e. the result obtained after the computation.

WB : shows the address of destination register at that clock cycle.

The output shown in the figure is in signed decimal form and tested against these instructions.



Instruction Set used to test the module in assembly.



Test Instruction set in binary form. (inst.txt)

Testing on these input instructions the REGISTER\_FILE was changed as follows:



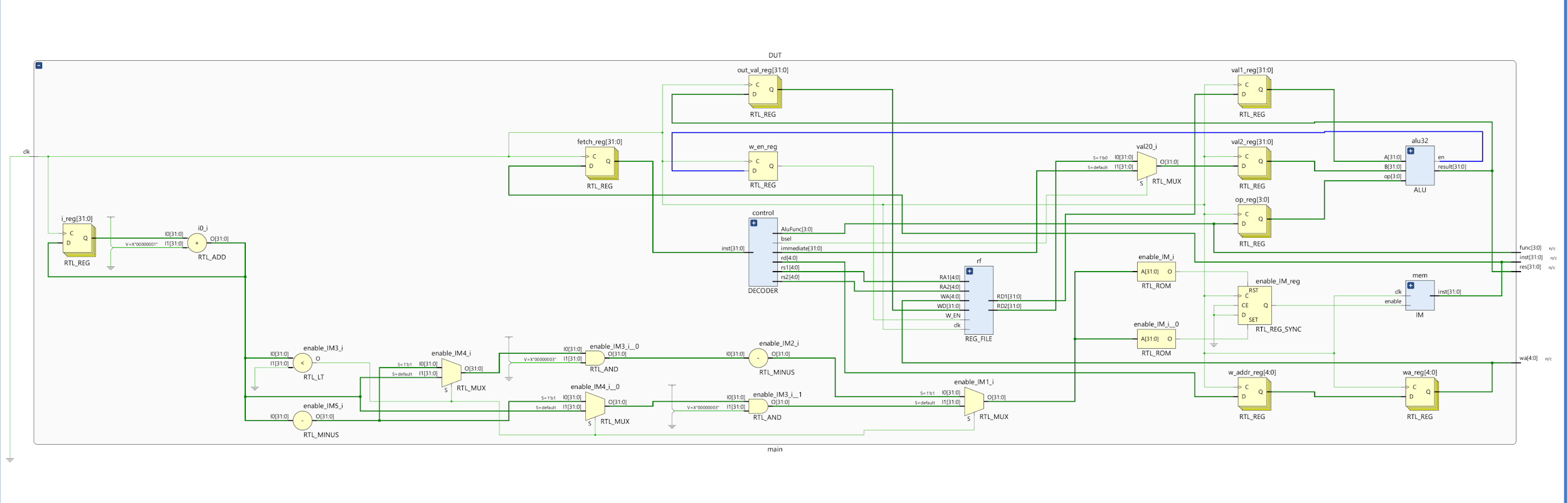
## 

## 

## 

## 

## Following is the schematic of the module created is as follows:



## **Result**

The modules for performing “ *add* ”, “ *sub* ” and “ *add i*  ” instructions are working for every valid instruction given through the testbench. The desired datapath for the basic functioning of the processor was also successfully created.