Faculty of Engineering & Technology

Department of Electrical and Computer Engineering

First Semester, 2023/2024

ENC3310– Advanced Digital Design Course Project

Student name: Katya Kobari

Student Id: 1201478

Instructor: Dr.Abdellatif Abu-Issa

Section: 2

21. Jan. 2024

Table of Contents

Table of figures

Introduction:

The aim of this project is to create a simple microprocessor using Verilog code. The microprocessor consists of three modules: an Arithmetic Logic Unit (ALU) that performs various arithmetic and logic operations, a Register File that stores the data, and an Instruction Format module that defines the structure of the instructions. The microprocessor can execute different operations depending on the instructions that I provide to it.

Microprocessor Components:

- ALU Module:

The Alu module consists of Verilog code that defines an Arithmetic Logic Unit (ALU) for a simple microprocessor. The ALU accepts three inputs: a 6-bit opcode, and two 32-bit signed operands and b. The ALU produces a 32-bit signed output result, which is the result of the operation specified by the opcode. The Alu module support 10 different operations: addition, subtraction, absolute value, negation, maximum, minimum, average, not, or and. The Alu module also handles the issue of invalid opcode by setting the result to zero. The Alu module uses a sequence of always block and if-else statements to implement each implementation logic.

- Register File Module:

A Register File is a memory unit that holds 32 32-bit signed values in a “ram” array. The reg_file module has four inputs: a clk signal for the clock, a valid opcode signal for the instruction validity, and two 5-bit addresses addr1 and addr2 for the registers to read. The module has two outputs: out1 and out2, which are the values from the registers addr1 and addr2.

The reg_file module also has another input: a 32-bit signed value in, which is the data to write to the register addr3, another 5-bit address. The ram array is initialized with some values in an initial block. Always block does the read and write operations on the clock’s positive edge. The operations only happen if the valid opcode signal is high, so the module only reads and writes when there is a valid opcode.

- Instruction Format Module:

The Instruction Format module takes two inputs: a 32-bit instruction and a clock signal clk. The Instruction Format module produces four outputs: a 6-bit opcode, and three 5-bit addresses addr1, addr2, and addr3. The opcode is the first 6 bits of the instruction, and it specifies the operation to be performed by the ALU. The addr1 and addr2 are the next 10 bits of the instruction, and they specify the source registers to read from. The addr3 is the next 5 bits of the instruction, and it specifies the destination register to write to. The Instruction Format module also produces a valid output, which is a 1-bit signal that indicates whether the opcode is valid or not. The Instruction Format module uses an always block and a series of if-else statements to extract the opcode and the addresses from the instruction, and to check the validity of the opcode. The Instruction Format module supports 10 valid opcodes and sets the valid output to high if the opcode matches any of them. Otherwise, the valid output is set to low.

- Mp_top Module:

The mp_top module defines a simple microprocessor. The microprocessor takes two inputs: a clock signal clk and a 32-bit instruction. The microprocessor produces one output: a 32-bit result, which is the outcome of the operation performed by the ALU. The mp_top module uses three submodules: an instruction format module, a reg_file module, and an alu module. The instruction format module extracts the opcode and the addresses from the instruction and checks the validity of the opcode. The

reg_file module reads and writes data from and to the registers specified by the addresses. The alu module performs the arithmetic and logic operation specified by the opcode on the data read from the registers. The mp_top module uses a wire valid to indicate whether the instruction is valid or not, and a reg store opcode to store the opcode until the next clock cycle. The mp_top module uses an always block to update the store opcode and to connect the submodules.

The ideas behind the design and how it works:

The microprocessor has four modules: alu, reg_file, instruction format, and mp_top. The alu module does arithmetic and logic operations on two numbers. The reg_file module stores 32 numbers in memory. The instruction format module takes an instruction and gets the code and its addresses. The mp_top module connects the other modules and gives the result.

I used a clock signal to make the data flow in my microprocessor. The clock signal has a positive edge when the voltage goes up. The clock signal makes some modules change their outputs on the positive edge. These modules can store data. For example, the instruction format module changes its outputs on the positive edge of the clock signal. It gets the code and the addresses from the instructions. It also checks if the code is valid and sets the valid output.

The modules that store data are connected to other modules that do operations. These modules do not store data. For example, the alu module does the operation on the data from the reg_file module and gives the result. The ALU module does not depend on the clock signal but on the changes in the code and the data. The alu module has a delay, which is the time it takes for the result to change after the code or the data changes. The delay is shorter than the clock period, which is the time between two positive edges of the signal. the reg_file module also changes its outputs on the positive edge of the clock signal. It reads the data from the source registers and writes the data to the destination register. It only does this if the valid output from the instruction format module is high. The mp_top module stores the code in a register on the positive edge of the clock signal. It uses this register to connect the alu module to the instruction format module. It also connects the reg_file module to the instruction format and ALU modules. It produces the result output from the ALU module.

So, the clock cycle decides when the instruction is decoded, when the data is read and written, and when the code is stored. The clock cycle does not decide when the operation is done by the ALU, but it affects when the result is ready. The result is ready after the delay of the ALU, which is less than the clock period.

Test the microprocessor:

I wrote a testbench module that simulates the inputs and outputs of my microprocessor. The testbench module has a clock signal clk and an instruction signal instruction as inputs, and a result signal result as output. The testbench module instantiates the mp_top module, which is the top level of my microprocessor design.

The testbench module has an initial block that defines an array of instructions, expected values, and operation names. The instructions are 32-bit binary numbers that encode the opcode and the addresses for the ALU operations. The expected values are the correct results of the ALU operations. The operation names are the names of the ALU operations. The testbench module also has a variable sysFlag that indicates whether the system passes or fails.

The testbench module has another initial block that iterates over the array of instructions. For each instruction, it assigns it to the instruction signal and waits for 20 ns. Then, it compares the resulting signal with the expected value for that instruction. If they match, it displays a message that the test case passes. If they do not match, it displays a message that the test case fails. It also sets the sysFlag to 0 if any test case fails. At the end of the iteration, it displays a message that the system passes or fails based on the sysFlag value. Then, it ends the simulation.

To interpret the results of my tests, I looked at the messages displayed by the testbench module. I also looked at the waveforms of the signals in the simulation tool. I checked if the resulting signal matched the expected value for each instruction and if the valid signal was high for valid opcodes and low for invalid opcodes. I also checked if the timing of the signals was correct and if there were any glitches or errors in the simulation.

I have tried more than one case using available operations, and I have tried a non-existent operation, and the results were all correct. In the end, these operations were fixed on the values in address one and address two, and the result was stored in address 3.

- **All Parts of the project work correctly.**

appendix 1:

- **A = 15034, B = 8854**

- **Case 1 (a + b)** ➔ ins = 0031041 → [addr3] = value[addr1] + value [addr2

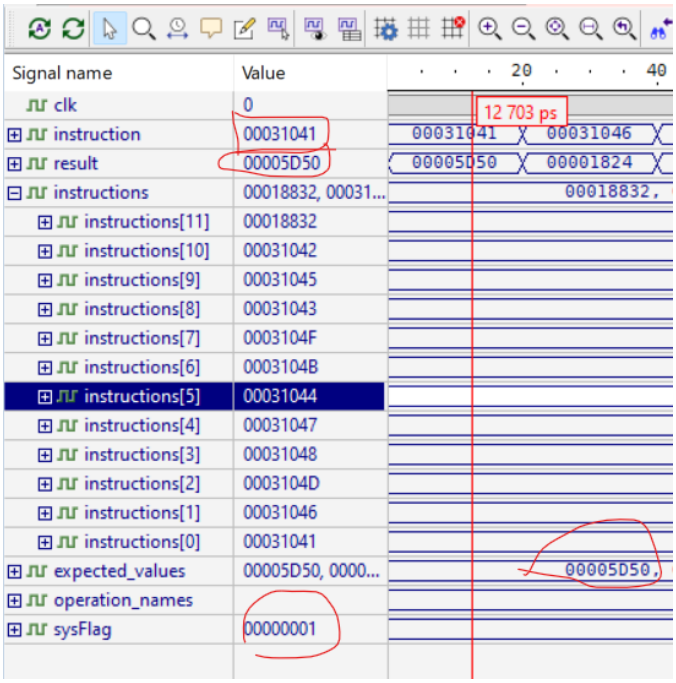

Figure 1. a + b

- **Case 2 (a - b)** ➔ ins = 00031046 → [addr3] = value[addr1] - value [addr2]

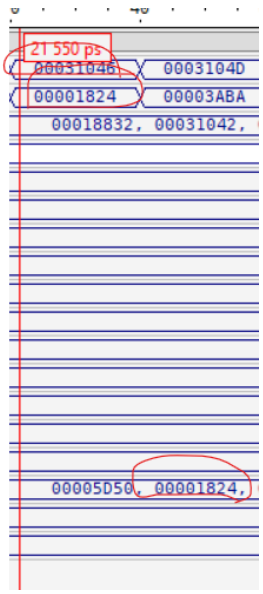

Figure 2. a – b

- **Case 3 (abs (A))** ➔ ins = 003104D → [addr3] = Abs(value[addr1])

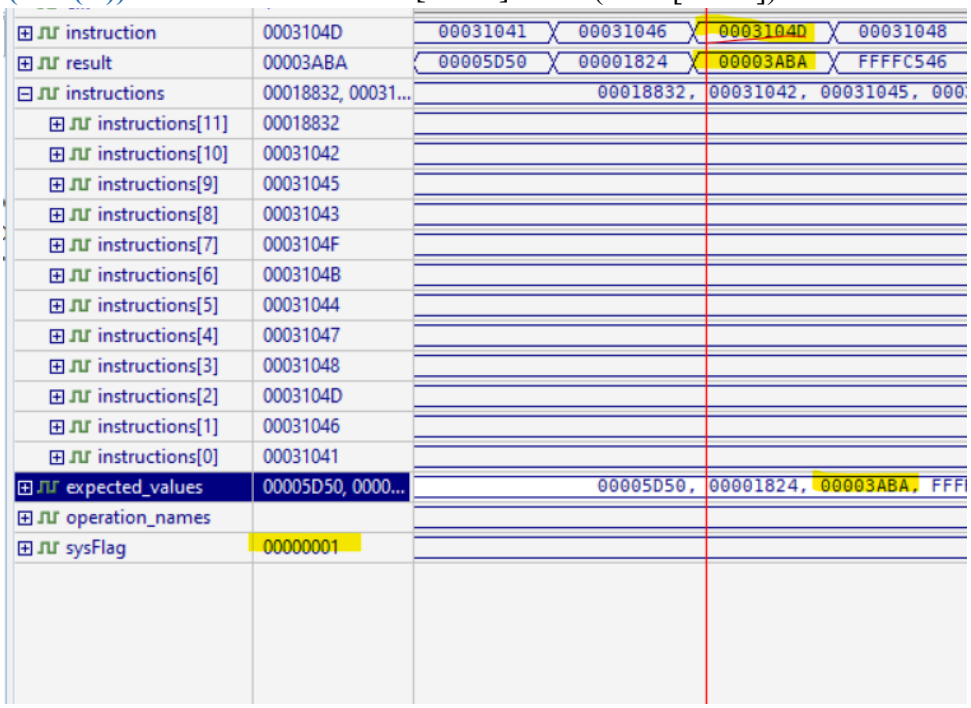

Figure 3. abs(A)

- **Case 4 (Invalid Instruction)** ➔ ins = 00018832 → Result =0

Figure 4. Invalid op code

- **Full Wave Form that Covers all cases.**

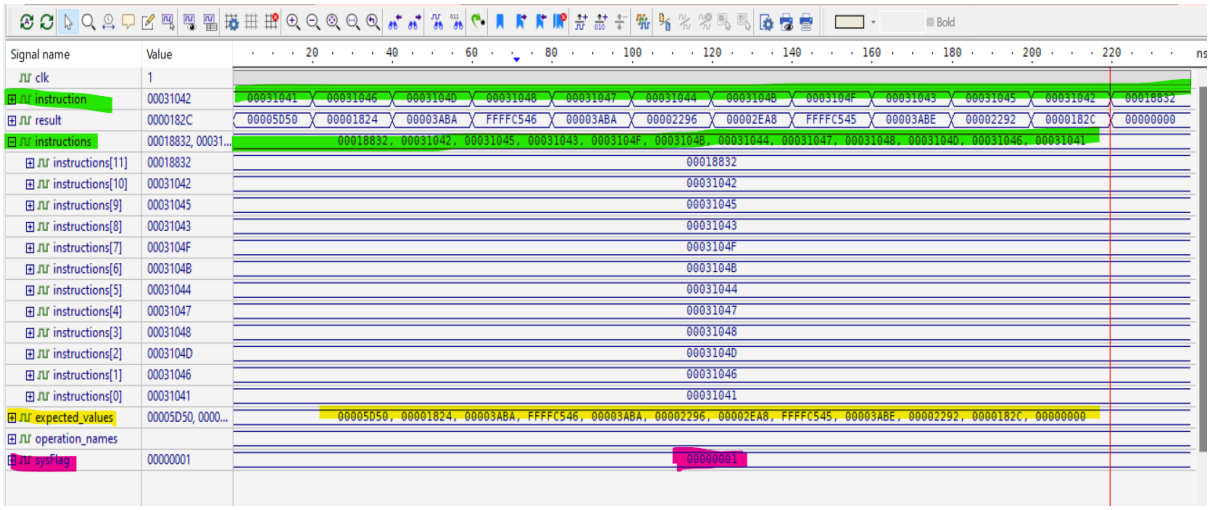

Figure 5. Wave form for the system

- System flag = 1 → Means that all results true.

Print all tested cases on the screen:

```
run 500 ns
# KERNEL: ---------------------------- microprocessor start runing----------------------------------
# KERNEL: ---------------------------- A = 15034 , B = 8854 ----------------------------------------
# KERNEL: ----PASS---- for test case 0 (add).
# KERNEL: Expected:       23888, Result:       23888
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 1 (sub).
# KERNEL: Expected:        6180, Result:        6180
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 2 (abs).
# KERNEL: Expected:       15034, Result:       15034
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 3 (-a).
# KERNEL: Expected:      -15034, Result:      -15034
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 4 (max(A,B)).
# KERNEL: Expected:       15034, Result:       15034
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 5 (min(A,B)).
# KERNEL: Expected:        8854, Result:        8854
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 6 (avg(A,B)).
# KERNEL: Expected:       11944, Result:       11944
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 7 (~a).
# KERNEL: Expected:      -15035, Result:      -15035
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 8 (a or b).
# KERNEL: Expected:       15038, Result:       15038
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 9 (a and b).
# KERNEL: Expected:        8850, Result:        8850
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 10 (a xor b).
# KERNEL: Expected:        6188, Result:        6188
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ----PASS---- for test case 11 (Invalid).
# KERNEL: Expected:           0, Result:           0
# KERNEL: ------------------------------------------------------------------------------------------
# KERNEL: ********************System Pass**********************
```

Figure 6.Results (Print)