**Report 4**

# **A technical summary of experiments conducted in the lab (steps, results, components, code functionality, etc.)**

## **Experiment 1: - A N-bit ALU with zero output flag**

The main purpose of this experiment is to create and simulate a 32-bit ALU with 4-bit selection lines allowing up to 16 different arithmetic and logic operations. In this experiment, the first 4 operations are only as addition, subtraction, ANDing, and ORing while the remaining 12 operations will produce zero output regardless of the input.

* **Steps**

A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.First, we will use modules from experiment 2 such as Full\_Adder module which used in An adder is a digital circuit that performs addition of numbers, and Ripple\_Carry\_Adder module which used in adding two binary number by using several Full\_Adder. where the carry-out of one Full Adder is connected to the carry-in of the next. However, the parameter n defines the bit-width of the adder, which is set to 32 in this experiment, because the main purpose of this experiment is to make 32-bit ALU.

To summarize, the main function for full adder is to perform the addition of three binary digits: two input bits (A and B) and a carry-in bit (Cin). It produces two outputs: a sum bit (Sum) and a carry-out bit (Cout), while the Ripple\_Carry\_Adder uses several Full\_Adders in order to add two binary numbers as the carry-out of one Full Adder is connected to the carry-in of the next. Furthermore, the usage of full adder and ripple carry add is to control sum processes in both cases negative and positive sum.

A screenshot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.Now, we are ready to use ALU module that performs various arithmetic and logical operations on **two n-bit inputs (a and b)** based on a 4-bit control signal (sel). The module also outputs a zeroFlag indicating whether the result of the operation is zero and the result of the operation itself (ALU\_out). The module ALU is parameterized with n = 32, meaning the default bit-width of the inputs and outputs is 32 bits. The inputs are a, b and sel. A and B are two n-bit inputs for the operands, while sell is 4-bits used for control signal that determines the operation to be performed. To illustrate, it determines the behavior of ALU based on the value of sel. There are two b\_two wire which stores either b or its complement (~b), depending on the operation (ALU\_out), and sum wire which stores the result of the addition operation (ALU\_out). In order to enable the ALU to subtract and add, both values of b whether b is positive or negative will be added b\_twos, as it will determined by the value of sel as if sell = 4'b0010 (binary 0010), b\_twos will store b otherwise it will store (~b). After then, This instantiates a Ripple\_Carry\_Adder module with n-bit inputs a and b\_twos, and a carry-in (cin) set to sel[2] where the result of the addition will be stored in sum. The always block is used to deine the behavior of the ALU based on the value of sel : In case 1 (4'b0010): Perform addition (ALU\_out = sum), case 2 (4'b0110): Perform subtraction (ALU\_out = sum), subtraction is achieved by adding a to the two's complement of b (handled by the Ripple\_Carry\_Adder), case 3 (4'b0000): Perform bitwise AND (ALU\_out = a & b), case 4 (4'b0001): Perform bitwise OR (ALU\_out = a | b), while the Default Case: If sel does not match any of the above, the output is set to 0. After computing ALU\_out, the zeroFlag is updated: If ALU\_out is non-zero, zeroFlag is set to 0, our f ALU\_out is zero, zeroFlag is set to 1.

* **Results**

The result will be a 32-bit ALU with 4-bit selection lines allowing up to 16 different arithmetic and logic operations which implement the first 4 operations only as addition, subtraction, and ORing while the remaining 12 operations will produce zero output regardless of the input.

**Snapshot of simulation output corresponding to the submitted testbench with a brief interpretation of the snapshot**

* Experiment 1

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A black background with white text and numbers

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.The test bench that lead to this simulation:

🡪 a[31:0]: - one of inputs that declared to be 60 in case of add and subtract and 25 in case of AND and OR.

🡪 b[31:0]: - one of inputs that declared to be 40 in case of add and subtract 29 in case of AND and OR .

🡪 sel[3:0]: - determine the behavior of the ALU if it will be add or subtract or AND or OR.

🡪 ALU[31:0]: - the final output of the operation. The result of any operation stored in this variable.

## **Experiment 2: A Register File with Reset**

The main purpose of this experiment is to create and simulate the RISC-V register file. It should contain 32 registers each having N-bit with 2 reading ports and one writing port that is active only if RegWrite signal is active.

A diagram of a computer program

AI-generated content may be incorrect.**Steps**

A screenshot of a computer program

AI-generated content may be incorrect.The module used in this experiment is regFile, which represents a register file commonly used in CPUs. A register file is a collection of registers that can be read from or written to, and it is a fundamental component in processors for storing data temporarily. The main purpose of register is to read and write data from it. Therefore, The inputs for this module are clk (for synchronization), rst, (to initialize the register file), reg\_write (control the signal to enable writing to the register file), [4:0] rr1, rr2 (5-bit address for reading two register), wr (5-bit address for writing to a register), wd (32 bit data to be written into the register file), rd1, and rd2 (32-bit data to read from the register specified by rr1 and rr2). After then, we have to declares a register file with 32 registers which means that means there are 32 registers, indexed from 0 to 31. The next step is to build always\_block to synchronous operations. The triggered of the always block is the edgeing clock, and the rising edge of the reset signal (posedge rst). The condition of the rest is (rst == 1). When the reset signal is high, all 32 registers in the register file are initialized to 0 using a for loop. The else condition in the always is write operation. If the reg\_write signal is high, the data wd is written into the register specified by the write address wr. The Final step is assigning assignments continuously read the values from the registers specified by rr1 and rr2 and assign them to rd1 and rd2, respectively. It is normal building an register like the once that have been taken in the previous labs. The main point is to assign variable for read and write inside the register files.

A screen shot of a computer

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.**Result**

The result of this experiment is RISC-V register file which continues 32 registers each having N-bit with 2 reading ports and one writing port that is active only if RegWrite signal is active.

**Snapshot of simulation output corresponding to the submitted testbench with a brief interpretation of the snapshot**

* A screenshot of a computer

  AI-generated content may be incorrect.Experiment 2

Register simulation that show the functionality of the regFile module, which is likely a register file used in a CPU or other digital systems.

* Clk: - clock signal that is used in synchronize operations in digital circuits. It controls the timing of data transfers and state changes.
* Rst: - reset signal that is used in initializing the circuit to a known state. When active, it typically clears registers and sets the system to a default configuration.
* reg\_write: - Control signal to enable writing to the register file.
* rr1[4:0]: - 5-bit addresses for reading two registers from the register file.
* rr2[4:0]: - 5-bit addresses for reading two registers from the register file.
* wr [4:0] :- 5-bit address for writing data to a register in the register file.
* wd [31:0] :- 32-bit data to be written to the register file.
* rd1 [31:0] :- 32-bit outputs representing the data read from the register file at address rr1.
* rd2 [31:0] :- 32-bit outputs representing the data read from the register file at address rr2.

## **Experiment 3: The control unit**

The main purpose of this experiment is to create and simulate a representation of the control unit. The control unit is a combinational circuit with the following truth table

A table with black text

AI-generated content may be incorrect.

* **steps**

A screenshot of a computer program

AI-generated content may be incorrect.The main goal of this experiment is to apply the mentioned table and simulate it. Therefore, we will create a module under the name control\_unit with the same name variables mentioned in the table. After then, we have to make a always block with if statement for each case based on the value of inst. The input is inst[6:2] and the rest variables are output.

A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

* **Results**

A representation of the control unit. The control unit is a combinational circuit with the following truth table as directing the operation of the processor by generating control signals that coordinate the activities of the other hardware components, such as the Arithmetic Logic Unit (ALU), registers, memory, and input/output devices. The control unit ensures that instructions fetched from memory are executed in the correct sequence and that the correct operations are performed at each step.

**Snapshot of simulation output corresponding to the submitted testbench with a brief interpretation of the snapshot**

* Experiment 3

A screenshot of a computer

AI-generated content may be incorrect.

Verify the functionality of the control\_unit module, which is likely a control unit in a CPU. The control unit generates control signals based on the instruction type (e.g., R-format, LW, SW, BEQ).

* Inst [6:2]: - A 5-bit register representing the opcode or part of the instruction.
* Branch: - A control signal that indicates whether the instruction is a branch (e.g., BEQ).
* MemRead: - A control signal that enables reading from memory (e.g., for LW instructions).
* MemtoReg- A control signal that determines whether the data to be written to the register file comes from memory (e.g., for LW instructions).
* MemWrite: - A control signal that enables writing to memory (e.g., for SW instructions).
* ALUSrc: - A control signal that determines whether the second operand of the ALU comes from a register or an immediate value.
* RegWrite: - A control signal that enables writing to the register file.

## **Experiment 4: - Create a module representing the ALU control unit.**

The main purpose of this experiment is to Create a module representing the ALU control unit. Use the following truth table:

A table with numbers and symbols

AI-generated content may be incorrect.

* **Steps**
* A screenshot of a computer program

  AI-generated content may be incorrect.Like the previous experiment, we will create a module under the name ALU\_C n with the same number of variables and same names mentioned in the previous table. After then, we will create an always block with begin if statement for each case based on the ALUOp (A 2-bit input that specifies the type of operation the ALU should perform.), inst[14:12] values which a 3-bit input which representing part of the instruction (likely the funct3 field in RISC-V or similar architectures), and inst[30] (A 1-bit input (likely representing the funct7 bit or another control bit in the instruction).
* **Results**

generateing the appropriate ALU selection signal (ALU\_selection) based on the input control signals (ALUOp, inst1, and inst2). The ALU selection signal determines the operation that the ALU (Arithmetic Logic Unit) should perform, such as ADD, SUB, AND, OR, etc.

**Snapshot of simulation output corresponding to the submitted testbench with a brief interpretation of the snapshot**

* Experiment 4

 test the functionality of the **ALU\_C\_U** module (the ALU Control Unit). The testbench applies different input combinations to the ALU\_C\_U module and observes the output (ALU\_selection) to verify that it behaves according to the given truth table.

A screenshot of a computer

AI-generated content may be incorrect.

* inst1[14:12]: - A 3-bit register representing part of the instruction (like the funct3 field in RISC-V or similar architectures).
* inst2: - A 1-bit register (likely representing the funct7 bit or another control bit in the instruction).
* ALUOp1:- A 2-bit register representing the ALU operation code from the main control unit.