Habib University

Dhanani School of Science and Engineering

Computer Architecture

(CS-330 / EE-471)

Lab Manual

Table of Contents

About the lab manual

This lab manual has been created with the help of practical experiments, several supporting documents and presentations listed in the Bibliography section.

The creation process of this manual is started during the summer 2018 by Dr. Hasan Baig, and this manual is continuously being updated.

For questions, comments, or suggestions, please contact Dr. Hasan Baig at the following email address: hasan.baig@sse.habib.edu.pk.

About the lab exercises

These laboratory exercises have been designed to get the students acquainted with the hardware design skills. You will learn how to design hardware using hardware description language (HDL); how to simulate your design; and how to test it on a reconfigurable chip. Once you get familiar with the design flow, you will be required to develop processor peripherals in the following labs. A brief summary of all the lab exercises are given below.

In **Lab 1**, you will be introduced to the programmable logic and the Verilog HDL. Furthermore, you will learn how to design a simple hardware and verify its functional behavior using a professional simulation tool, named *ModelSim®*.

In **Lab 2a**, a hardware synthesis flow is discussed targeting the Xilinx FPGA technology. Furthermore, you will get to run your designed hardware on actual FPGA chip.

In **Lab 2b**, you will learn how to integrate the ready-made module (UART) with your custom design. Also, in this session, you will use desktop-based software, designed specifically for this course, to observe the output of on-chip hardware.

In **Lab 3**, you will be developing some intermediate modules of a processor which will be required in next labs. In particular, you will develop a multiplexer, an instruction parser, and immediate field extractor.

In **Lab 4**, In this lab, you will develop a Register File for a processor, and will simulate its behavior in ModelSim.

In **Lab 5**, You will develop a RISC arithmetic and logic unit and verify its functionality using simulation.

In **Lab 6**, you will learn how to use BRAM module in FPGA in order to implement the program memory of a processor. We will also implement a datapath for instruction fetch followed by functional verification.

In **Lab 7**, you will develop a single cycle RISC processor for R-type instructions and perform its behavioral simulation.

In **Lab 8**, you will design and test components for RISC processor that can handle branch instructions.

In **Lab 9**, you will design and test components for RISC processor that can handle memory reference instructions such as load and store.

In **Lab 10**, you will integrate the previously designed modules to form a single datapath for executing any type of instructions.

In **Lab 11**, you will design a control unit of RISC processor and then integrate it with the previously developed complete datapath.

Conventions

The following conventions appear in this lab manual.

📖	This icon denotes a "pre-lab exercise", which a student should complete before coming into the respective lab.
🖥	This icon denotes a "lab exercise", which a student should complete during the lab hours.
✎	This icon denotes a "post-lab exercise", which a student should complete outside the lab hours.
⏱	This icon indicates the expected time (in minutes) to complete the specific exercise.
💡	This icon denotes a tip, which notifies you to advisory information.
⚠	This icon denotes an alert, which notifies you to important information.
`Bold` `or` *`Italic`*	The text written in this font is used specifically for the syntax of HDL.
bold	Bold text denotes items that you must select or click or enter the value in the software, such as open file option or running the simulation button or entering the command in the transcript window. The bold text is also used to refer to the specific options in the software tools.
italic	Italic text denotes the name of a folder or a file path.
bold and italic	Bold and italic text denotes the name of a file.

Lab 5 – RISC ALU

Objectives

In this lab, we will develop a RISC arithmetic and logic unit and verify its functionality using simulation.

Section	⏱
a) Introduction A brief overview of this lab.	**02**
b) Implementation This section is divided into two lab tasks. First, you will implement and test a 1-bit RISC ALU. Second, you will connect 6 instances of 1-bit ALUs to construct a 6-bit ALU followed by its verification using test bench.	**40 + 60**
Exercise You will develop a behavioral model of a 64-bit ALU.	**45**

a. Introduction

The arithmetic logic unit (ALU) is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This lab constructs an ALU from four hardware building blocks (AND and OR gates, inverters, and multiplexors) and illustrates how combinational logic works.

Because the RISC-V registers are 64 bits wide, we need a 64-bit-wide ALU. Let's assume that we will connect 64 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU, shown below.

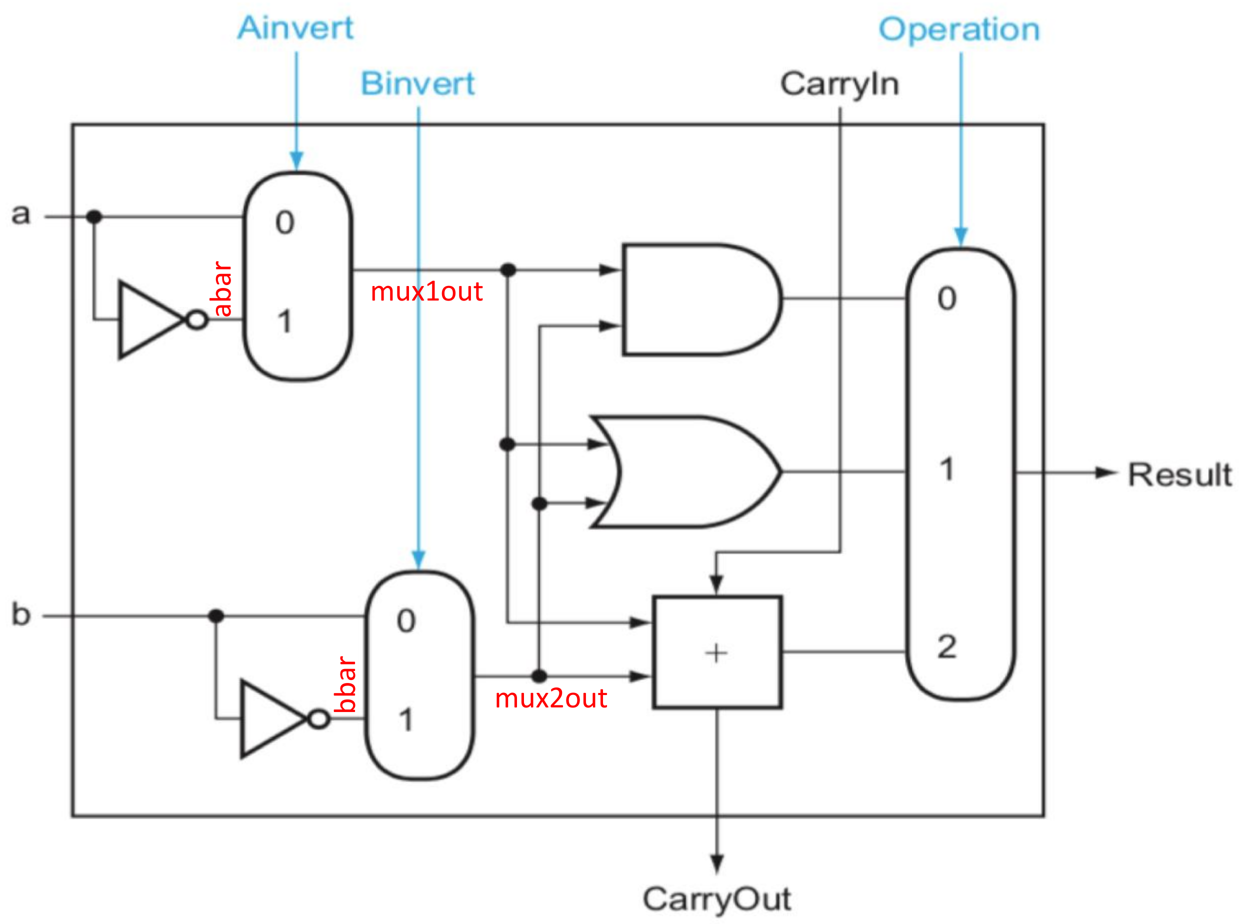


Fig. 5.1. 1-bit ALU with four control signals – Ainvert, Binvert, [1:0] Operation.

b. Implementation

In this lab, we will design an ALU that will be able to perform a subset of the ALU operations of a full RISC-V ALU. In this exercise, we will develop an ALU that will take two 1-bit inputs, `a` and `b`, and will be able to execute the following five instructions:

add, sub, and, or, nor

The 1-bit ALU will generate a 1-bit output, named "Result", and a "CarryOut" signal. The different operations will be selected by a 4-bit control signal, named "ALUop". The description of these 4 control signals are shown in the table below.

[3:0] ALUOp				**Function**
Ainvert	Binvert	Operation [1]	Operation[0]	
0	0	0	0	AND
0	0	0	1	OR
0	0	1	0	Add
0	1	1	0	Subtract
1	1	0	0	NOR

i. *Lab Task 01*

Design a module named "`ALU_1_bit`" to incorporate the functionality shown in Fig. 5.1. This module should have 1-bit inputs `a`, `b`, `CarryIn`; 4-bit input `ALUOp`; and two 1-bit outputs `Result` and `CarryOut`.

The text shown in red in Fig. 5.1. indicate the name of corresponding wires. For example, the wire at the output of upper inverter (having input `a`) is given a name `abar`.

As discussed in lecture, the output `CarryOut` should be implemented according to the following equation:

```
CarryOut = Input1.CarryIn + Input2.CarryIn + Input1.Input2
```

where,
`Input1` and `input2` are the inputs to the adder shown in Fig. 5.1.

The inputs to the adder shown in Fig. 5.1. are not the input signals `a` and `b`. Choose the appropriate signals carefully.

Write a testbench and verify the functionality of all five operations.

ii. *Lab Task 02*

Now you are supposed to use this 1-bit ALU to implement 64-bit ALU by instantiating the previously developed module, `ALU_1_bit`, 64 times. Since it will become too difficult to debug these many modules, therefore, for now, extend this 1-bit ALU to 6-bit ALU by instantiating `ALU_1_bit` module 6 times in a separate top module and make the appropriate connections as shown in Figure 5.2 (only for 6-bits ALU).

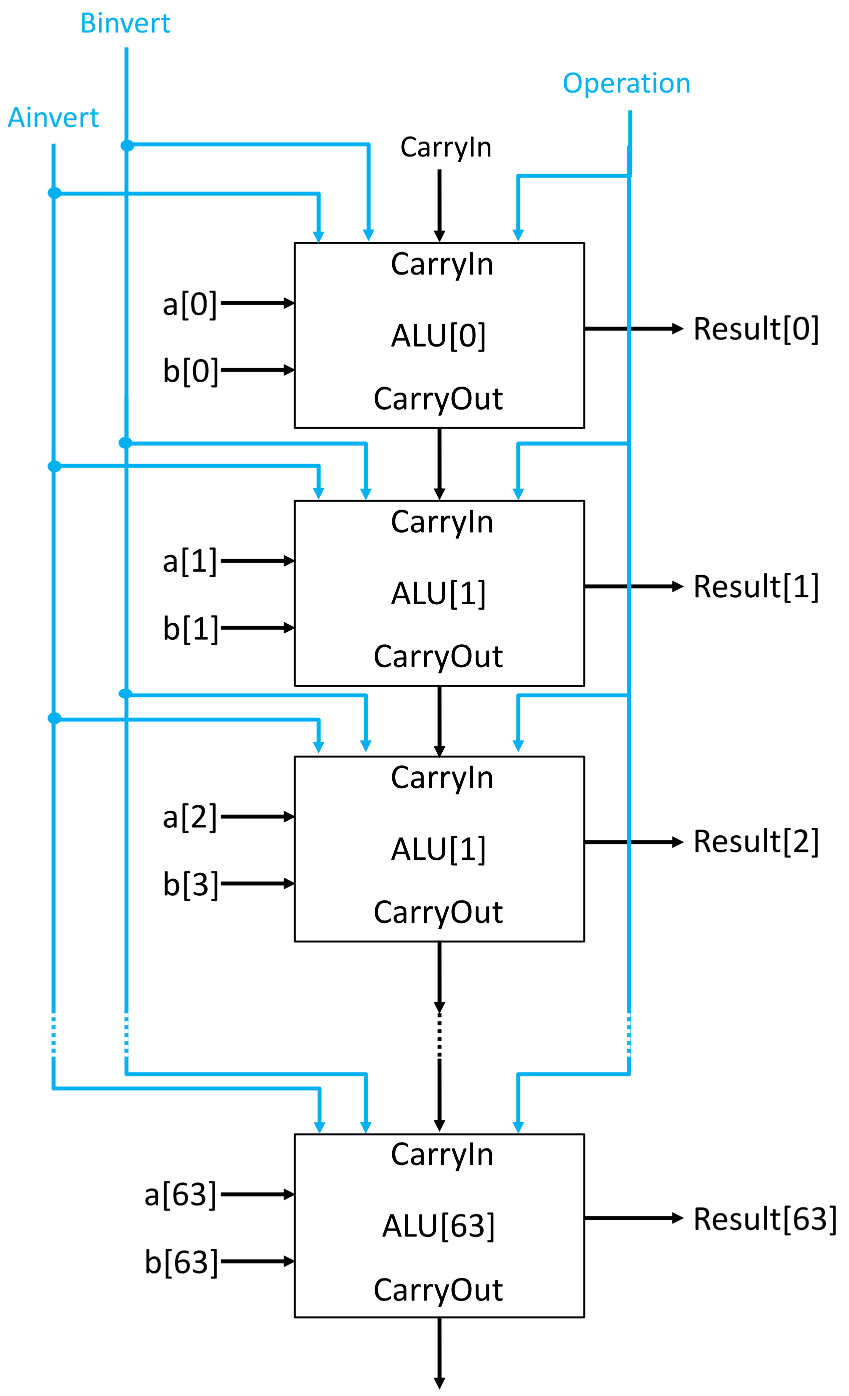


Fig. 5.2. Extension of 1-bit ALU to 64-bit ALU. In this lab, you are required to use 1-bit ALUs 6 times to construct a 6-bit ALU.

Write a testbench to test the functionality of 6-bit ALU and verify the correct functioning of all five operations.

Exercise

At this point, you have already experienced that extending `ALU_1_bit` module 64 times will result in a 64-bit ALU. However, this process would be too lengthy and cumbersome. In contrast, you are required to develop a behavioral model of 64-bit ALU just by declaring `a` and `b` 64-bits wide and declare the corresponding operations using a single multiplexer. You also need to add an additional output named ZERO in your 64-bit ALU, as shown in Fig. 5.3 below.

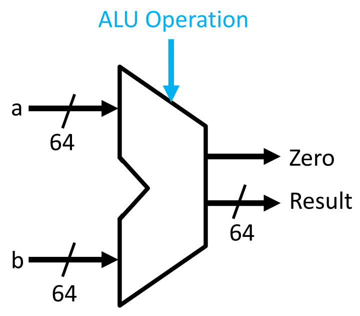
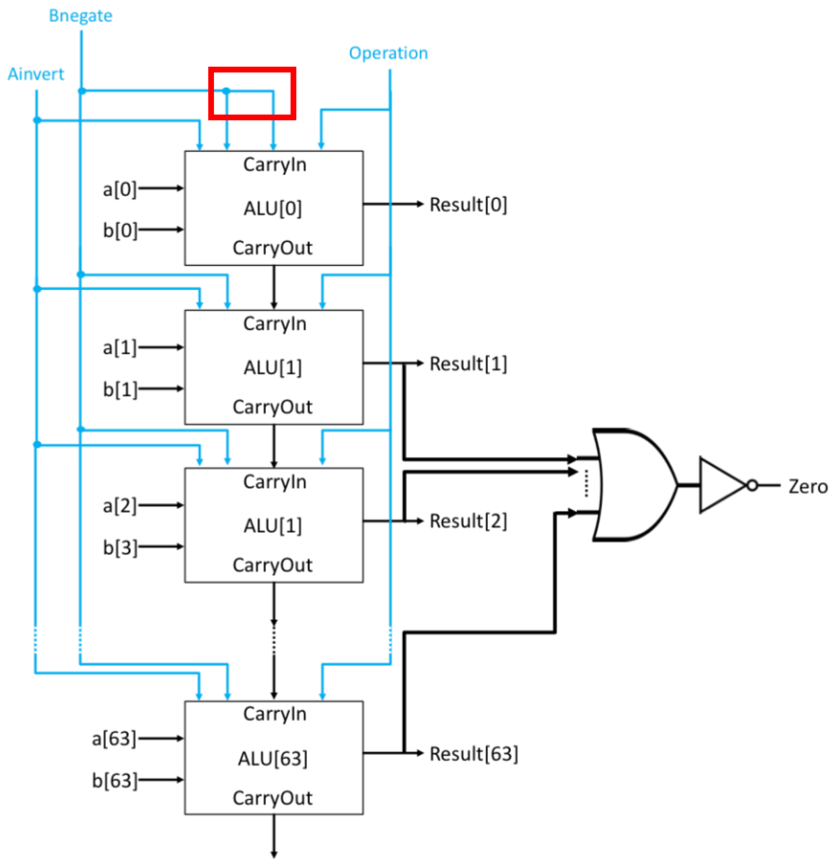


Fig. 5.3. 64-bit ALU with ZERO output.

This ZERO output should be set to 1 if the `Result` is 0, else set it to 0. The circuitry for triggering ZERO is shown in the figure below. Also make sure that you tie Binvert signal with CarryIn.


Bnegate
Ainvert
Operation
CarryIn
ALU[0]
CarryOut
a[0]
b[0]
Result[0]
CarryIn
ALU[1]
CarryOut
a[1]
b[1]
Result[1]
CarryIn
ALU[1]
CarryOut
a[2]
b[3]
Result[2]
CarryIn
ALU[63]
CarryOut
a[63]
b[63]
Result[63]
Zero