

# Habib University



Dhanani School of Science and Engineering

**Computer Architecture**  
(CS-330 / EE-471)

**Lab Manual**

# Table of Contents

---

ABOUT THE LAB MANUAL .....	1
ABOUT THE LAB EXERCISES.....	3
CONVENTIONS .....	5
LAB 3 – BASIC PROCESSOR MODULES.....	7
Objectives .....	7
a. Introduction.....	8
<i>i. Case Structure</i> .....	8
b. Multiplexer .....	9
c. Instruction Parser .....	9
d. Immediate Data Generator .....	9

## 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](mailto: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, and a decoder and perform simulation.

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 3 – Basic Processor Modules

## Objectives

In this lab, we will be developing some intermediate modules of a processor which will be required in next labs. In particular, we will develop a multiplexer, Instruction parser, and a Immediate field extractor.

Section	
a) <u>Introduction</u> 	05
A brief overview of the lab exercises and some additional Verilog elements are discussed which will be required to complete this lab.	
b) <u>Multiplexer</u> 	25
In this section, you will develop a Verilog module of 2x1 multiplexer, its testbench and simulate it using ModelSim.	
c) <u>Instruction Parser</u> 	25
In this section, you will develop a module to parse 32-bit instruction into 6 different output fields.	
d) <u>Immediate Data Generator</u> 	60
In this section, you will develop a module to obtain the 12-bits <i>immediate</i> data from the 32-bit input for different instruction format and then extend it to 64 bits output.	



## a. Introduction

From now on, you will mostly be developing modules and simulating the results at your own. In this lab you are required to develop a multiplexer, an instruction parser, and an immediate data extractor.



To complete this lab, you may need to use *if/else structure*, *case structure*, *concatenation operator* or *assignment operator* in Verilog. If you are already familiar with the syntax, jump directly to section b.

### i. Case Structure

We already have seen the syntax and usage of if-else structure in Lab01 (D\_FF implementation). The if-else structure can also be nested similar to traditional programming languages. However, the nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

The syntax of Case conditional structure, in Verilog, is shown below. The keywords `case`, `endcase`, and `default` are used in the case statement.

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
    ...
    default: default_statement;
endcase
```

Each of `statement1`, `statement2`, `default_statement` can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords `begin` and `end`. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the `default_statement` is executed.



Placing of multiple default statements in one case statement is not allowed.



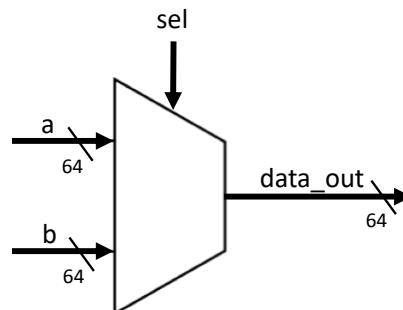
The `default_statement` is optional.





### b. Multiplexer

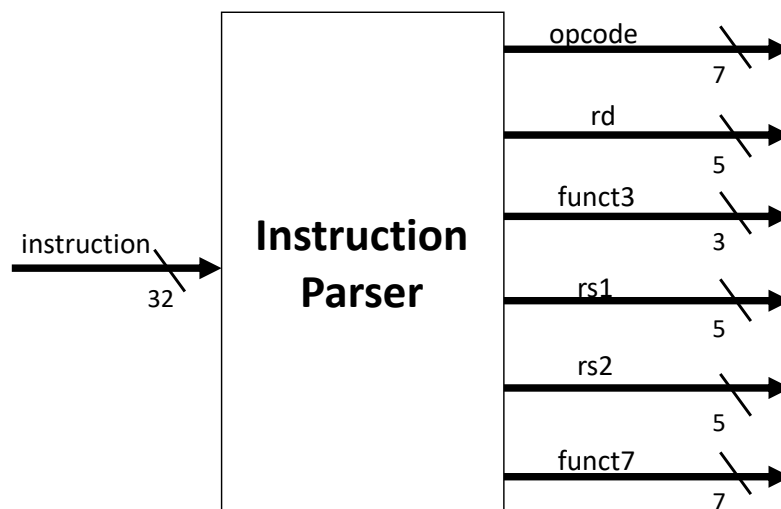
Develop a 2x1 multiplexer in which the two inputs are `a` and `b`, and each of them are 64-bits wide, as shown in the following figure.



Write a testbench to simulate its behavior in ModelSim.

### c. Instruction Parser

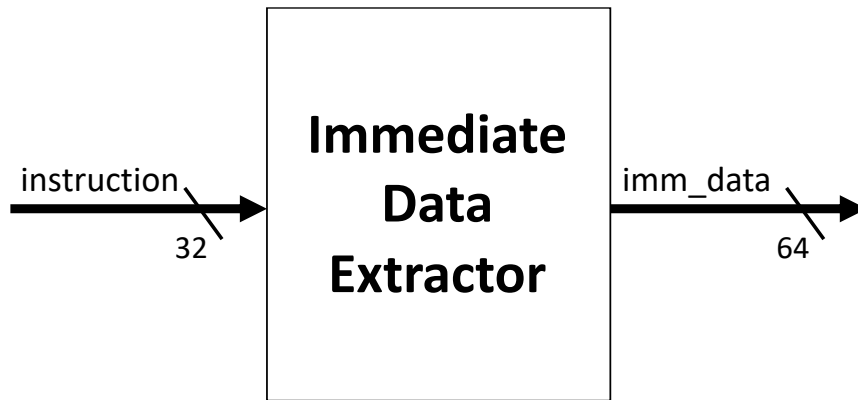
Develop a module which takes a 32-bit input, named `instruction`, and generates different outputs as shown in the following figure.



Assign the specific bits of `instruction` to the corresponding output field. Write a testbench to simulate its behavior in ModelSim.

### d. Immediate Data Generator

Develop a module which takes the 32-bit input `instruction` and extracts the 12-bit immediate data field depending on the type of instruction. Then extend these 12-bits to 64-bits output `imm_data`, as shown in the following figure.



The immediate generation logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for load instructions, bits 31:25 and 11:7 for store instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch. Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field. RISC-V opcode bit 6 happens to be 0 for data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 happens to be 0 for load instructions and 1 for store instructions. Thus, bits 5 and 6 can control a 3:1 multiplexor inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

Write a testbench for this module and verify its functionality.