**Data Register and Instruction Register**

**CENG 3151**

Submitted by

Michael Lanford

1816337

Computer Engineering

University of Houston-Clear Lake

Houston, Texas 77058

February 27, 2023

# Table of Contents

**1** [**Abstract 3**](#_Toc126060766)

[**2** **Introduction** 3](#_Toc126060767)

**3 Requirements……………………………………………………………………….……......3**

**4 Prelab………………………………………………………………………………..……….4**

**5 Implementation……………………………………………………….…………………......4**

5.1 Design Code / Design Diagrams**……………………………………………………………5**

5.2 Schematics**……………………………………………………………………………….....7**

5.3 Testbench**…………………………………………………………………………………....8**

5.4 Waveform / Results**………………………………………………………………………...13**

[**6** **Conclusion** 13](#_Toc126060768)

**Abstract**

# Processor registers are fast memory locations that are used by the CPU to store information that needs to be easily accessed. The memory it stores is temporary and gets constantly accessed by the CPU. MIPS has 3 types of instruction formats but for this lab we will only talk about 2 of them: I-Type and R-Type. R-Type instructions are used when the data that is accessed is all located in the registers. I-Type instructions are used when the data that is accessed is an immediate value. In this lab, we will be using Xilinx Vivado to build two 32-bit registers: one that has a synchronous reset and one that has a similar data format to the MIPS I-Type instruction. The major results of this experiment will be a waveform that shows the correct output for each input, which will reflect our input values.

# Introduction

For this project, we will be using Xilinx Vivado to build and test two 32-bit registers that will accept some input and produce some output.

1. **Requirements**

Design a 32-bit register with synchronous reset and design another 32-bit register that has the data format similar to MIPS I-Type instruction. The first circuit has four inputs: RegIn, Reset, Load, and Clock along with one output labeled RegOut. The second circuit has three inputs: Instruction, Load\_IR, and Clock along with four outputs: OpCode, Rs, Rt, and Const\_Addr. The figures of the circuits can be seen below:

A picture containing diagram

Description automatically generated

**Figure 1:** Diagram for the first 32-bit circuit to be designed.

Diagram

Description automatically generated with medium confidence

**Figure 2:** Diagram for the second 32-bit circuit to be designed.

1. **Prelab**

For this prelab, we were required to study processor registers and MIPS instruction formats and to also write down the instructions that use MIPS I-Type format and to give some examples.

Processor registers are fast memory locations that are used by the CPU to store information that needs to be easily accessed. The memory it stores is temporary and gets constantly accessed by the CPU. MIPS has 3 types of instruction formats but for this lab we will only talk about 2 of them: I-Type and R-Type. R-Type instructions are used when the data that is accessed is all located in the registers. These operations are usually add, sub, and, or, and slt. The MIPS instructions that use I-Type format are the load/store operations, branch operations, and immediate operations. Some examples of an I-type format instruction in MIPS are: addi $t0, $s6, 4 and sw $t1, 0($t0).

1. **Implementation**

The first step of implementation was to create a new project in Xilinx Vivado. After setting up the project with the correct options, we added a design source file for the first 32-bit instruction register and added the necessary inputs and outputs to it. We then coded the clock and load process to set conditions for reset and load. After that, we created the simulation file and added in the component instantiation, interface signal declarations, instance port map, the clock process, and the test cases to it then tested the waveform. For the second 32-bit instruction register, we created another new project in Xilinx Vivado. After setting up the project with the correct options, we added a design source file for the new 32-bit instruction register and added the necessary inputs and outputs to it. We then coded the clock and load process to set conditions for the Load\_IR. After that, we created the simulation file and added in the component instantiation, interface signal declarations, instance port map, the clock process, and the test cases to it then tested the waveform.

**4.1 Design Code / Design Diagrams**

------Part 1

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

--Input and Output declarations

entity Lab5Design is

Port ( RegIn : IN std\_logic\_vector(31 downto 0);

Load : in STD\_LOGIC;

Clock : in STD\_LOGIC;

Reset : in STD\_LOGIC;

RegOut : OUT std\_logic\_vector (31 downto 0));

end Lab5Design;

architecture Behavioral of Lab5Design is

begin

process(clock, load) is begin

if(rising\_edge(Clock)) then

if(reset = '1') then--if the reset it 1, the output should be all 0's regardless of load

RegOut <= x"00000000";

elsif(load = '1') then--If the reset is 0 then we check if load is 1 to provide data, otherwise it will output whatever previous data was stored

RegOut <= Regin;

end if;

end if;

end process;

end Behavioral;

------Part 2

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

--Input and Output declarations

entity Lab5Pt2 is

Port ( Instruction : in STD\_LOGIC\_VECTOR (31 downto 0);

Load\_IR : in STD\_LOGIC;

Clock : in STD\_LOGIC;

OpCode : out STD\_LOGIC\_VECTOR (31 downto 26);

Rs : out STD\_LOGIC\_VECTOR (25 downto 21);

Rt : out STD\_LOGIC\_VECTOR (20 downto 16);

Immediate : out STD\_LOGIC\_VECTOR (15 downto 0));

end Lab5Pt2;

architecture Behavioral of Lab5Pt2 is

begin

process(clock, Load\_IR ) is begin

if(rising\_edge(Clock)) then

if(Load\_IR = '1') then

OpCode <= Instruction(31 downto 26);

Rs <= Instruction(25 downto 21);

Rt <= Instruction(20 downto 16);

Immediate <= Instruction(15 downto 0);

end if;

end if;

end process;

end Behavioral;

**4.2 Schematics**

**Diagram

Description automatically generated**

**Figure 3:** First 32-bit instruction register circuit.

Diagram, schematic

Description automatically generated

**Figure 4:** Second 32-bit instruction register circuit.

**4.3 Testbench**

--Part 1

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Lab5Sim is

-- Port ( );

end Lab5Sim;

architecture Behavioral of Lab5Sim is

component Lab5Design is--Instantiate the component

Port ( RegIn : IN std\_logic\_vector(31 downto 0);

Load : in STD\_LOGIC;

Clock : in STD\_LOGIC;

Reset : in STD\_LOGIC;

RegOut : OUT std\_logic\_vector (31 downto 0));

end component;

signal RegIn: std\_logic\_vector(31 downto 0);--Signal declarations

signal Load, Clock, Reset: std\_logic;

signal RegOut: std\_logic\_vector(31 downto 0);

constant Clock\_period : time := 10 ns;

begin

uut: Lab5Design PORT MAP(RegIn, Load, Clock, Reset, RegOut);--Port maps

Clock\_process: process--Clock process

begin

Clock <= '0';

wait for Clock\_period/2;

Clock <= '1';

wait for Clock\_period/2;

end process;

process

begin

reset <= '0';--Case 1, output should equal the input

wait for Clock\_period;

load <= '1';

wait for Clock\_period;

RegIn <= x"226a0004";

wait for Clock\_period;

reset <='0';--Case 2, output should remain the same as case 1 despite new RegIn

wait for Clock\_period;

load <= '0';

wait for Clock\_period;

RegIn <= x"226a000f";

wait for Clock\_period;

reset <= '1';--Case 3, output should be 0 due to reset, even with load = 1

wait for Clock\_period;

load <= '1';

wait for Clock\_period;

RegIn <= x"226a0004";

wait for Clock\_period;

reset <= '0';--Output should change, to a new output from the first RegIn

wait for Clock\_period;

load <= '1';

wait for Clock\_period;

RegIn <= x"226a000f";

wait;

end process;

end Behavioral;

--Part 2

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Lab5Pt2Sim is

-- Port ( );

end Lab5Pt2Sim;

architecture Behavioral of Lab5Pt2Sim is

component Lab5Pt2 is--Instantiate the component

Port ( Instruction : in STD\_LOGIC\_VECTOR (31 downto 0);

Load\_IR : in STD\_LOGIC;

Clock : in STD\_LOGIC;

OpCode : out STD\_LOGIC\_VECTOR (31 downto 26);

Rs : out STD\_LOGIC\_VECTOR (25 downto 21);

Rt : out STD\_LOGIC\_VECTOR (20 downto 16);

Immediate : out STD\_LOGIC\_VECTOR (15 downto 0));

end component;

signal Instruction: std\_logic\_vector (31 downto 0);--Signal declarations

signal Load\_Ir, Clock: std\_logic;

signal OpCode: std\_logic\_vector (31 downto 26);

signal Rs: std\_logic\_vector(25 downto 21);

signal Rt: std\_logic\_vector(20 downto 16);

signal Immediate: std\_logic\_vector(15 downto 0);

constant Clock\_period : time := 10 ns;

begin

uut: Lab5Pt2 PORT MAP(Instruction, Load\_IR, Clock, OpCode, Rs, Rt, Immediate);--Port maps

Clock\_process: process--Clock Process

begin

Clock <= '0';

wait for Clock\_period/2;

Clock <= '1';

wait for Clock\_period/2;

end process;

process

begin

wait for Clock\_period;--Case 1

Load\_Ir <= '1';

wait for Clock\_period;

Instruction <= x"226a0004";

wait for Clock\_period;

wait for Clock\_period;--Case 2

Load\_Ir <= '0';

wait for Clock\_period;

Instruction <= x"226a000f";

wait for Clock\_period;

wait for Clock\_period;--Case 3

Load\_Ir <= '1';

wait for Clock\_period;

Instruction <= x"226a0004";

wait for Clock\_period;

wait for Clock\_period;--Case 4

Load\_Ir <= '1';

wait for Clock\_period;

Instruction <= x"226a000f";

wait;

end process;

end Behavioral;

**4.4 Waveform / Results**

The waveforms below shows that the four programs we made above in the Testbench and Design Code / Design Diagrams sections was able to produce correct results for each of our inputs that we created.

Graphical user interface

Description automatically generated

**Figure 5:** First 32-bit instruction register circuit Waveform.

A screenshot of a computer

Description automatically generated

**Figure 6:** Second 32-bit instruction register circuit Waveform.

# Conclusion

In this lab, we were able to successfully code two 32-bit instruction register circuits in Xilinx Vivado by using the little code snippets given to us in our prelab as a base for our code. These programs were made to be able to simulate the register and be able to load, store, and split into the correct data format, which can be seen in the waveforms due to the correct output being produced.