PHASE #1

Presented to:

Prof. Dr.Tamer Mostafa Abdelkader

Eng.Yasmine Shaban

Ain Shams University

Faculty of Engineering

Computer Engineering and Software Systems

I-CHEP

CSE112 – Computer Organization and Architecture

Spring 2024

Table of Contents

Computer Organization and Architecture

Phase #1

Presented by:

Roger Sherif Salama 22P0112

Abdelrahman Mostafa Ali El-Din 22P0150

Omar Ahmed Dardir 22P0218

Mohamed Ashraf Mohamed 22P0210

Andrew Rami Bassily 22P0187

Abstract

With the memorable help and guidance of our professor Prof.Dr.Tamer Mostafa and our Teacher Assistant Eng.Yasmine Shaban, we managed to build this project that concerns a system with 32 registers, multiplexers, a decoder, and an ALU. Using binary inputs, users choose operands, and operations are carried out in accordance with those selections. Combined these components make up a primitive CPU.

Phase#1

1.0 INTRODUCTION

This section of the report we will be discussing our work in phase #1, going over the ideology behind designing a primitive CPU and the process of developing and implementing it as a functional VHDL code. We will delve deeper into our thought process as we designed the system and how XILINIX was used in the system's implementation.

2.0 IDEOLOGY

The basic CPU is made up of:

32 Registers: There are 32 bits in each register.
Two 32x1 multiplexers
One 5x32 ALU Decoder

Using the five selection lines that are included in every 32x1 MUX, the user designates which registers will be utilized as the operands. To accomplish this, enter their five-bit binary code into the read_sel1 and read_sel2 inputs. Read_sel1 and read_sel2 begin at the 25th and 21st bits, respectively, in the instruction input, and read_sel2 begins at the 20th and ends at the 16th bit. Prior to the user's selection, each of the 32 registers will have a predetermined value. The user must designate which registers to utilize as operands.

Nevertheless, the 5x32 decoder is used for determining the location where the result of the operations between the two operands is saved. The decoder's input lines enter the write_select input with values from the instruction input's 15th bit to 11th bit.
Data1 and data2's operands are taken from each register by the ALU. It executes and outputs into the dataout the corresponding operation between them based on the bits entered in the
ALUop. Zflag equals true if the output happens to be zero.

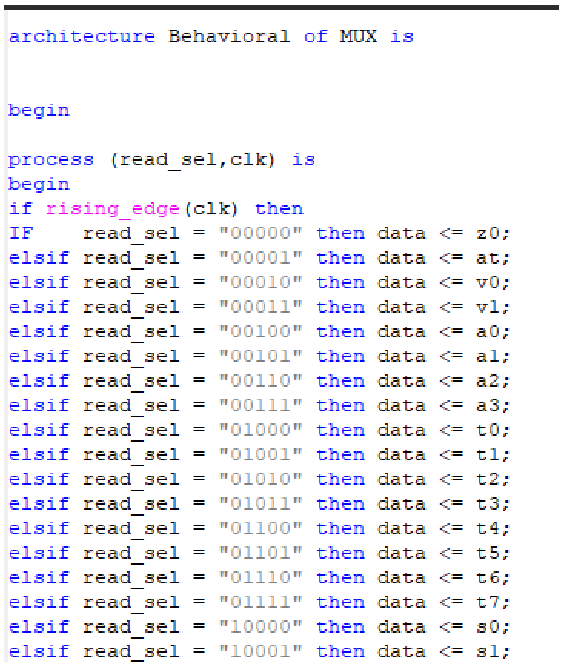
The contents of the dataout are then stored in the temp, which subsequently stores the data it received in the write_data, which correctly puts the output at the intended location.

3.0 PARTS OF THE PROJECT

3.1 Part 1

3.1.1 Mux

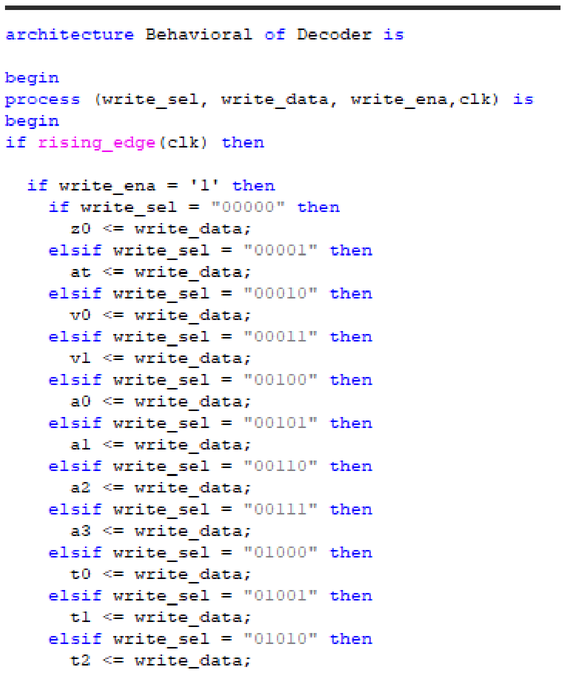
The Mux we've implemented is a 1x32 Mux, meaning it has 5 selection lines and 32 inputs, with 1 output. These 32 inputs correspond to registers, and the 5-bit selection line indicates which register to pick. In the port, we've defined all 32 registers, the 5-bit read_sel representing the selection lines, and the data inside the selected register and a clk. Inside the architecture, a process was defined using the read_sel and the clk as parameters, where through a series of if and elsif conditions, we assign the value inside the selected register to the data based on the read_sel value. This all happens synchronously with the CLK's rising edge. Below is a sample of the conditions used:



```
architecture Behavioral of MUX is


begin

process (read_sel,clk) is
begin
if rising_edge(clk) then
IF    read_sel = "00000" then data <= z0;
elsif read_sel = "00001" then data <= at;
elsif read_sel = "00010" then data <= v0;
elsif read_sel = "00011" then data <= v1;
elsif read_sel = "00100" then data <= a0;
elsif read_sel = "00101" then data <= a1;
elsif read_sel = "00110" then data <= a2;
elsif read_sel = "00111" then data <= a3;
elsif read_sel = "01000" then data <= t0;
elsif read_sel = "01001" then data <= t1;
elsif read_sel = "01010" then data <= t2;
elsif read_sel = "01011" then data <= t3;
elsif read_sel = "01100" then data <= t4;
elsif read_sel = "01101" then data <= t5;
elsif read_sel = "01110" then data <= t6;
elsif read_sel = "01111" then data <= t7;
elsif read_sel = "10000" then data <= s0;
elsif read_sel = "10001" then data <= s1;
```

3.1.2 Decoder

As for the decoder, a 5x32 decider was used, meaning it has 5 inputs and 32 outputs. Its role is to determine the destination register where we want to write data. The port includes all 32 registers, along with write_sel (the input to the decoder), write_data (the data to be written in the register), and write_en (the switch that must be 1 to enable writing into the register) and a clk. Inside the architecture, a process was defined with the parameters write_sel and write_data, write_en and clk. Using a series of if and elsif conditions, the data in the write_data is stored inside the desired register based on the write_sel value when the write_en is 1. This all happens synchronously with the CLK's rising edge. Below is a sample of the conditions used:



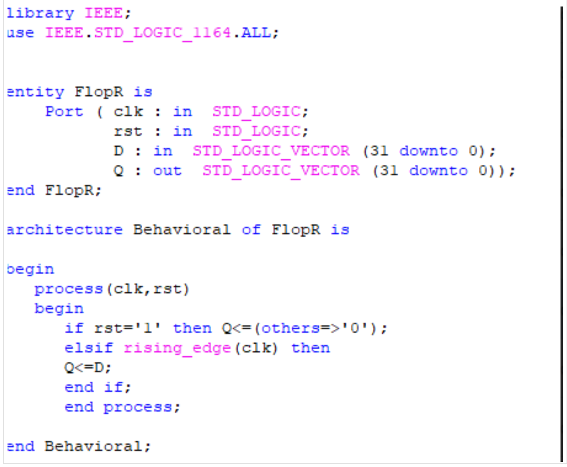
```
architecture Behavioral of Decoder is

begin
process (write_sel, write_data, write_ena,clk) is
begin
if rising_edge(clk) then

  if write_ena = '1' then
    if write_sel = "00000" then
      z0 <= write_data;
    elsif write_sel = "00001" then
      at <= write_data;
    elsif write_sel = "00010" then
      v0 <= write_data;
    elsif write_sel = "00011" then
      v1 <= write_data;
    elsif write_sel = "00100" then
      a0 <= write_data;
    elsif write_sel = "00101" then
      a1 <= write_data;
    elsif write_sel = "00110" then
      a2 <= write_data;
    elsif write_sel = "00111" then
      a3 <= write_data;
    elsif write_sel = "01000" then
      t0 <= write_data;
    elsif write_sel = "01001" then
      t1 <= write_data;
    elsif write_sel = "01010" then
      t2 <= write_data;
```

3.1.3 FlopR

The flopR component is a 32 bit d-flip-flop that stores the value inside the register (D) and outputs it synchronously with the rising edge of the clock (Q). In the port we defined D and Q as a 32 bit input and output respectively, alongside the clk and rst as inputs. Inside the architecture, there's a process sensitive to changes in the clock (`clk`) and the reset signal (`rst`).

- If the reset signal is asserted (`rst='1'`), all bits of the output vector `Q` are set to '0'.
- Otherwise, if there's a rising edge on the clock (`rising_edge (clk)`), the data input `D` is assigned to the output `Q`.

This essentially implements a synchronous reset flip-flop, where the stored data (`Q`) is updated on the rising edge of the clock signal, and reset to zero when the reset signal is 1.Below is the VHDL code for the FlopR:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity FlopR is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           D : in  STD_LOGIC_VECTOR (31 downto 0);
           Q : out  STD_LOGIC_VECTOR (31 downto 0));
end FlopR;

architecture Behavioral of FlopR is

begin
   process(clk,rst)
   begin
      if rst='1' then Q<=(others=>'0');
      elsif rising_edge(clk) then
      Q<=D;
      end if;
      end process;

end Behavioral;
```

3.1.4 Register files

The register files is where we put all of the previous components to actual use as declare them in packages and import these packages in the register file module. The aimed functionality of the registerfile is to combine all the previous components to be able to pick and read the argument registers and write in the desired destination register. In the port the following ports were defined: \

- read_sel1 and read_sel2 were which will be responsible of reading the data from the argument registers
- write_sel which. selects the desired register to write in
- write_en which is the switch that allows data to be written inside a register
- ,the clk
- write_data which is the data to be written inside the destination
- data1 and data2 which are the output that stores the data stored inside the argument register files we selected.

64 signals were then defined, 32 for the inputs of the register files to write in them and 32 for their outputs to read from them.

Moreover, we defined each one of the register files as a FlopR made of 32 bits with the port map (CLK, '0' ,r (index), out (index)); , this way we make sure each register includes 32 bits, and correctly attached to its corresponding input and output with the rst always being 0.

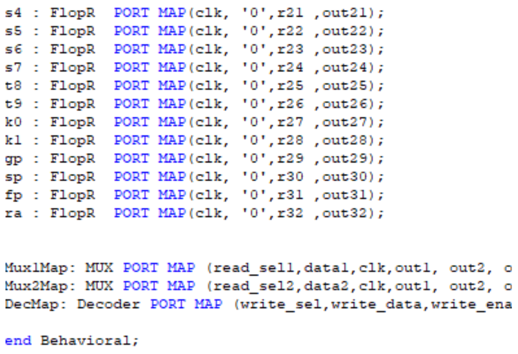
Additionally, 2 new Muxs were defined called Mux1Map and Mux2Map with a port map sending the following parameters:

- read_sel1 for the first mux and read_sel2 for the second mux, which is the 5 bit selection line for the Mux
- data1 for the first Mux and data2 for the second Mux which will be the output containing the data stored inside the selected argument register.
- All 32 output signals for the registers were sent to each Mux to be able to read from them.

Finally, a decoder was defined, called DecMap with a port map sending the following parameters:

- write_sel which is the input to the decoder that points out which register is going to be picked and written in.

- write_data which is the data that will be written inside the picked register
- write_en which is the 1 bit switch that allows the writing process to take place.
- All 32 inputs signals of the registers are also sent to the decoder file to be able to write in them. Below is a sample of the VHDL code used for this unit:



```
s4 : FlopR  PORT MAP(clk, '0',r21 ,out21);
s5 : FlopR  PORT MAP(clk, '0',r22 ,out22);
s6 : FlopR  PORT MAP(clk, '0',r23 ,out23);
s7 : FlopR  PORT MAP(clk, '0',r24 ,out24);
t8 : FlopR  PORT MAP(clk, '0',r25 ,out25);
t9 : FlopR  PORT MAP(clk, '0',r26 ,out26);
k0 : FlopR  PORT MAP(clk, '0',r27 ,out27);
k1 : FlopR  PORT MAP(clk, '0',r28 ,out28);
gp : FlopR  PORT MAP(clk, '0',r29 ,out29);
sp : FlopR  PORT MAP(clk, '0',r30 ,out30);
fp : FlopR  PORT MAP(clk, '0',r31 ,out31);
ra : FlopR  PORT MAP(clk, '0',r32 ,out32);


Mux1Map: MUX PORT MAP (read_sel1,data1,clk,out1, out2, o
Mux2Map: MUX PORT MAP (read_sel2,data2,clk,out1, out2, o
DecMap: Decoder PORT MAP (write_sel,write_data,write_ena

end Behavioral;
```

3.2 Part 2 (ALU)

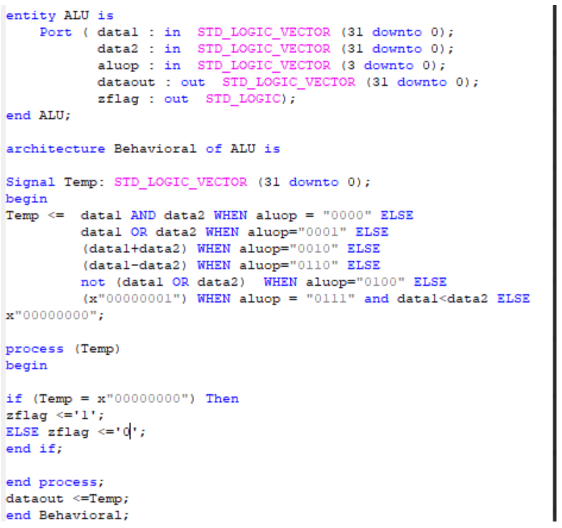
The ALU serves as the computational brain of our CPU, it functions by receiving 2 inputs and performing a certain operation on them.
The designed ALU for this CPU implements 6 main functions, which are the addition, subtraction, AND operation, OR operation, NOR operation, and finally SLT (set if less than).

As per mentioned, the ALU receives 2 inputs, those are Data1 and Data2, each one being 32-bits, and then performs a certain operation on them and outputs their result in DataOut.

The way the operation is selected is by using a different input called ALUOp, this input is a 4-bit input, the ALU is programmed to check for this input prior to any other function, and based on the value of it, a certain operation is selected and performed and has its value stored in DataOut. The following table shows the code for the ALUOp and their corresponding functions.

Aluop	Function
0000	AND
0001	OR
0010	ADD
0100	SUB
1100	NOR
0111	SLT

Lastly, a special flag is used to check whether the outputted data equals ‘0’ or not, this flag is labelled as Zflag (zero flag), where it always outputs ‘1’ if DataOut has the value of ‘0’, otherwise, outputs ‘0’.



```
entity ALU is
    Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
           data2 : in  STD_LOGIC_VECTOR (31 downto 0);
           aluop : in  STD_LOGIC_VECTOR (3 downto 0);
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
           zflag : out  STD_LOGIC);
end ALU;

architecture Behavioral of ALU is

Signal Temp: STD_LOGIC_VECTOR (31 downto 0);
begin
Temp <=  data1 AND data2 WHEN aluop = "0000" ELSE
         data1 OR data2 WHEN aluop="0001" ELSE
         (data1+data2) WHEN aluop="0010" ELSE
         (data1-data2) WHEN aluop="0110" ELSE
         not (data1 OR data2)  WHEN aluop="0100" ELSE
         (x"00000001") WHEN aluop = "0111" and data1<data2 ELSE
x"00000000";

process (Temp)
begin

if (Temp = x"00000000") Then
zflag <='1';
ELSE zflag <='0';
end if;

end process;
dataout <=Temp;
end Behavioral;
```

3.3 Part 3 (Connecting the modules in main module)

The components Alu and RegisterFiles were each declared in a package, and the packages were then imported in the main module ready for use. The port for the main module was the following:

Port (clk, reset: in STD_LOGIC;

instr: in STD_LOGIC_VECTOR (31 downto 0);

aluoperation : in STD_LOGIC_VECTOR(3 downto 0);

zero: out STD_LOGIC;

regwrite: in STD_LOGIC;

aluout: out STD_LOGIC_VECTOR(31 downto 0));

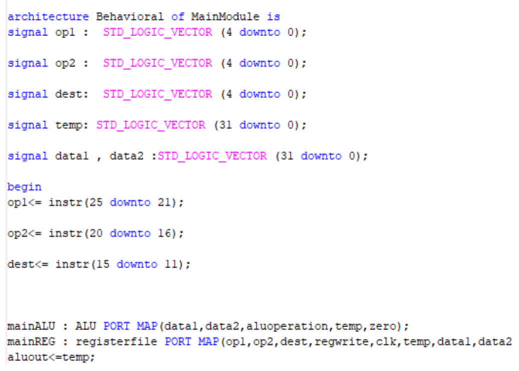
The instr is the 32 bit R-format instruction that defines the desired operation, argument registers and destination register. From the 25th bit to 21st bit, is the register that contains the first operand. From the 20th bit to 16th bit, is the register that contains the second operand. From the 15th bit to 11th bit, is the register that we want

to write the output in.The regwrite bit is the write enable bit.

The aluoperation is the operation that was executed inside the Alu, the aluout is the output of the Alu and Zero is the Zflag.

A total of 6 signals were defined in the main module:

- op1 and op2 which are the signals for the registers of the operands sent to the Register File to pick the desired 2 registers to operate on.
- dest which is the signal for the destination register sent to the Register File to pick the desired destination register.
- temp which is the signal sent to the Register File instead of the write_sel because we need to store the value in dataout from Alu in write_data in the Register File. The signal is needed because dataout is an output and latter is an input therefore value can not be transferred directly
- data1 and data2 which are the signals for for the operands sent to the Alu to process on



```
architecture Behavioral of MainModule is
signal op1 :  STD_LOGIC_VECTOR (4 downto 0);

signal op2 :  STD_LOGIC_VECTOR (4 downto 0);

signal dest:  STD_LOGIC_VECTOR (4 downto 0);

signal temp: STD_LOGIC_VECTOR (31 downto 0);

signal data1 , data2 :STD_LOGIC_VECTOR (31 downto 0);

begin
op1<= instr(25 downto 21);

op2<= instr(20 downto 16);

dest<= instr(15 downto 11);



mainALU : ALU PORT MAP(data1,data2,aluoperation,temp,zero);
mainREG : registerfile PORT MAP(op1,op2,dest,regwrite,clk,temp,data1,data2
aluout<=temp;
```

The following components are then defined in the main module mainALU : ALU with the following parameters sent to it PORT MAP(data1,data2,aluoperation

,temp,zero);

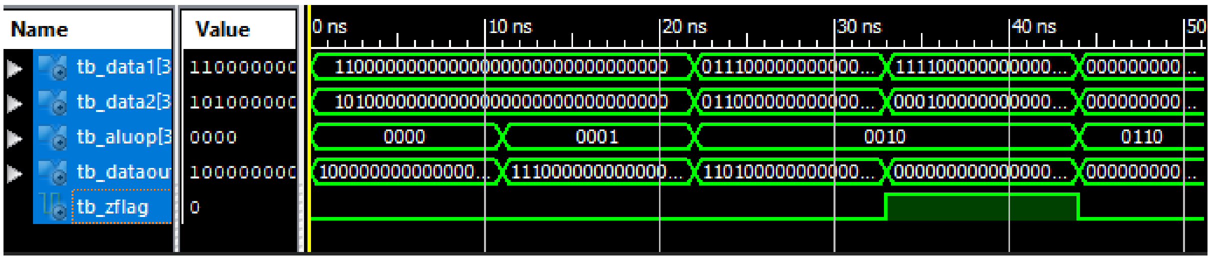
mainREG : registerfile with the following parameters sent to it PORT MAP(op1,op2,dest,regwrite,clk,

temp,data1,data2);

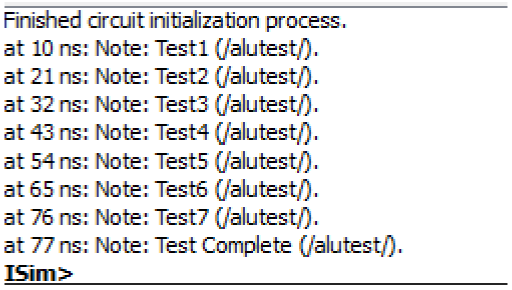
Finally, the value in temp is stored in aluout.

4.0 SIMULATION AND TESTING

4.1 Part 1 Alu Simulation And Testing



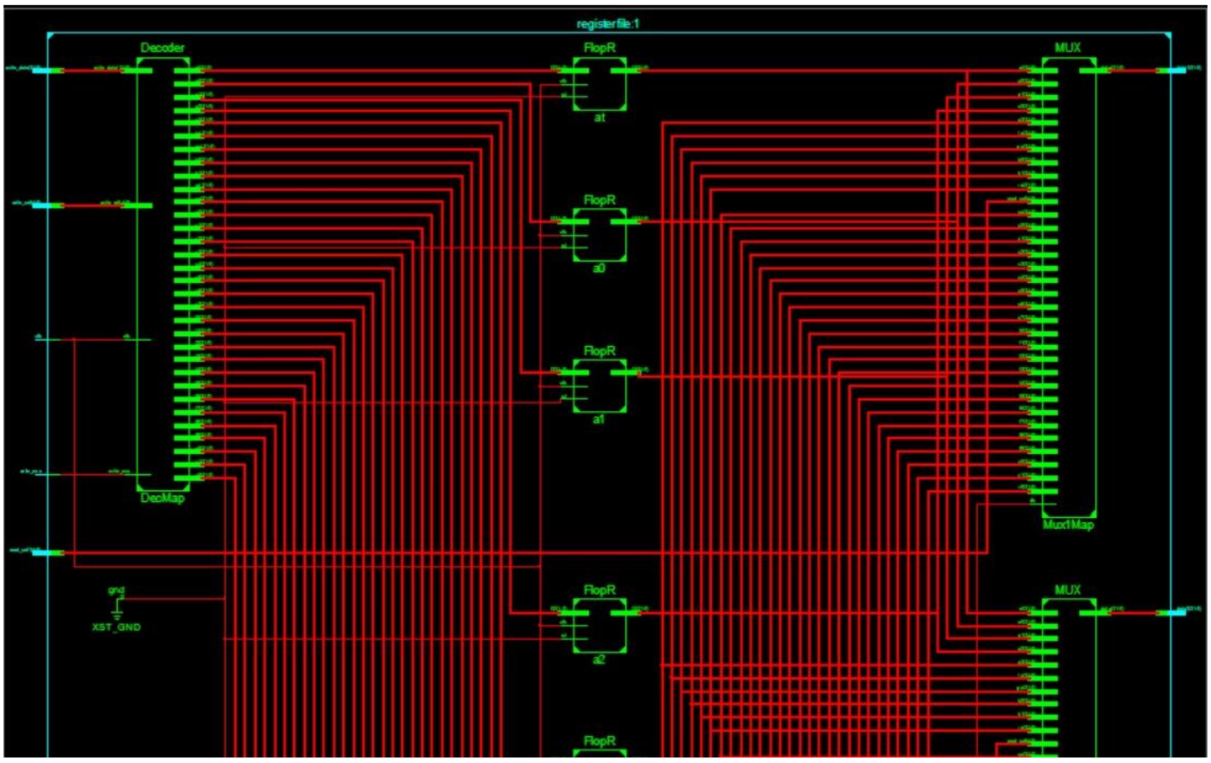
Using the provided test bench, the ALU’s functionality was tested and these were the results:



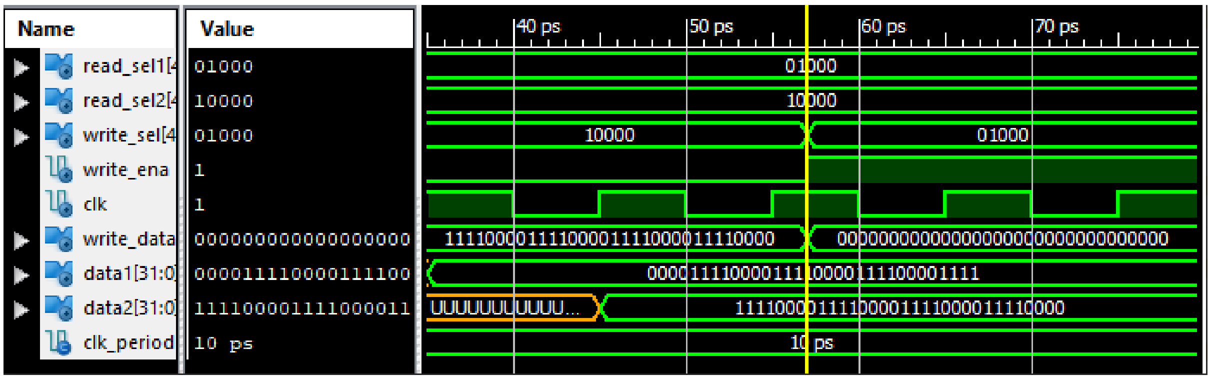
```
Finished circuit initialization process.
at 10 ns: Note: Test1 (/alutest/).
at 21 ns: Note: Test2 (/alutest/).
at 32 ns: Note: Test3 (/alutest/).
at 43 ns: Note: Test4 (/alutest/).
at 54 ns: Note: Test5 (/alutest/).
at 65 ns: Note: Test6 (/alutest/).
at 76 ns: Note: Test7 (/alutest/).
at 77 ns: Note: Test Complete (/alutest/).
ISim>
```

4.2 Part 2 Register File Simulation And Testing

In the picture below is the RTL of the register file visualizing how all the components are connected inside the register file.

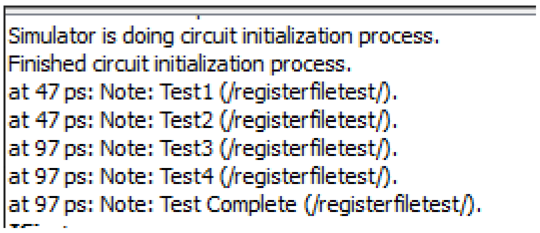


The simulation:



```
Name              Value
read_sel1[4       01000
read_sel2[4       10000
write_sel[4       01000
write_ena         1
clk               1
write_data        000000000000000000
data1[31:0]       000011110000111100
data2[31:0]       111100001111000011
clk_period        10 ps
```

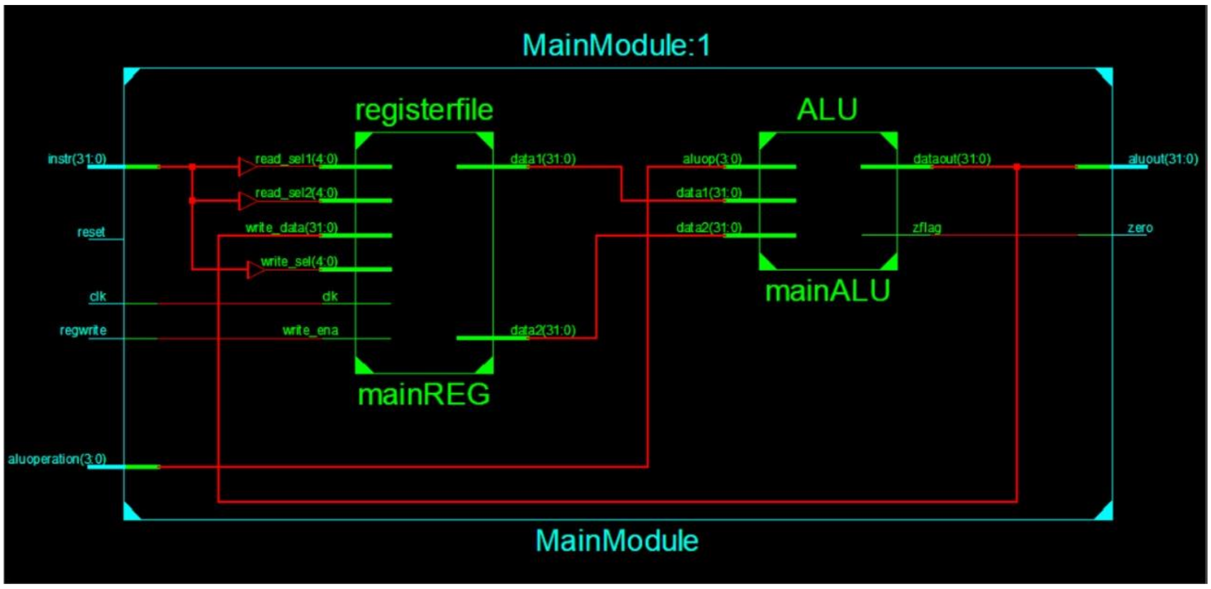
Using the provided test bench, the register file’s functionality was tested and these were the results:



```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 47 ps: Note: Test1 (/registerfiletest/).
at 47 ps: Note: Test2 (/registerfiletest/).
at 97 ps: Note: Test3 (/registerfiletest/).
at 97 ps: Note: Test4 (/registerfiletest/).
at 97 ps: Note: Test Complete (/registerfiletest/).
```

4.3 Part 3 Main Module Simulation And Testing

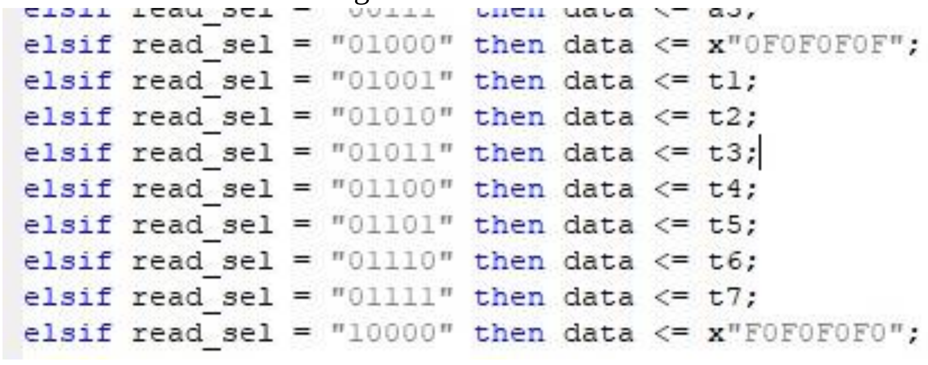
Below is the RTL of the main module:



To simulate and test the main module, a new test bench was created as shown below:

```
-- Stimulus process
stim_proc: process
begin
   wait for clk_period - 3ps;
      --and between t0 and s0 in t1
      reset<='0';
      instr<="00000001000100000100100000000000";
      aluoperation<="0000";
      regwrite<='1';

      wait for clk_period * 2;
      aluoperation<="0001";
      regwrite<='1';
      wait for clk_period * 2;

   wait;
end process;
```

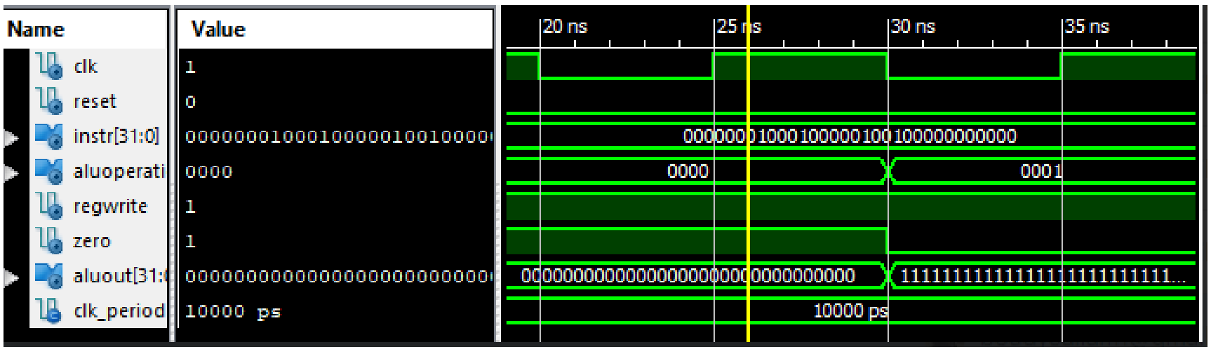
However, an obstacle was in the way of using this test bench, and that being all the registers initialized to null therefore the output is always null. To overcome this problem, values were stored inside registers $s0 and $t0, and operated on to store the output in register $t1. The values were stored in the registers as shown below:



```
elsif read_sel = "01000" then data <= x"0F0F0F0F";
elsif read_sel = "01001" then data <= t1;
elsif read_sel = "01010" then data <= t2;
elsif read_sel = "01011" then data <= t3;
elsif read_sel = "01100" then data <= t4;
elsif read_sel = "01101" then data <= t5;
elsif read_sel = "01110" then data <= t6;
elsif read_sel = "01111" then data <= t7;
elsif read_sel = "10000" then data <= x"F0F0F0F0";
```

A 32 bit instruction was then inputted in the test bench as shown above, that were then translated by the main module to use $s0 and $t0 as the argument registers and $t1 as the destination register The aluoperation was also inputted for the ALU to perform OR with the values stored in the 2 argument registers as the operands.

Here were the results:



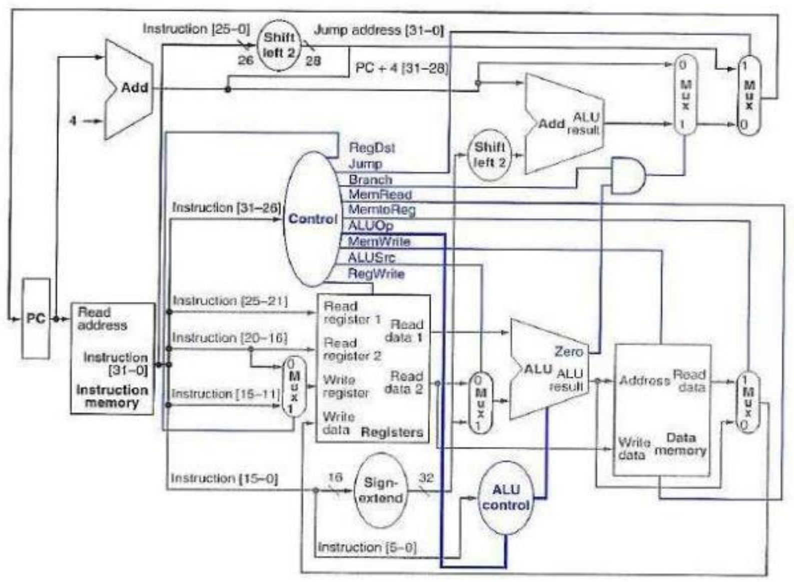
5.0 CONCLUSION

All in all, register files, an arithmetic logic unit (ALU), and the Mux and Decoder modules are covered in this report. Data is selected by the Mux, written by the Decoder, stored in register files for improved storage, and calculated by the ALU. Reliable data storage is ensured by the VHDL code for a synchronous reset flip-flop.

6.0 CONTRIBUTION SHEET #1

Team Member	**Contributions**
Roger Sherif Salama 22P0112	ALU, Mux, Report
Abdelrahman Mostafa Ali El-Din 22P0150	Mux, Decoder, testing
Omar Ahmed Dardir 22P0218	FlopR, Register File, testing
Mohamed Ashraf Mohamed 22P0210	Main module, Decoder, Report
Andrew Rami Bassily 22P0187	Register file, Main module, Report

Phase#2



7.0 Introduction

In this section of the report we will be going through our work in phase2 delving into the details of our implementation of the circuit shown above; Processor module containing Mips {Controller (control unit, ALU control), Datapath(flopr, 5 muxs, ALU, registerfiles, sign extension, shift left and PC)}, instruction memory and data memory.

8.0 Parts of the project

8.1 Controller

8.1.1 Control Unit

Control unit implements 9 signals. It takes the bits from 31 to 26 from the instruction(which determine the operation) to determine which signal to use. These 9 signals are then inputted to the datapath.

8.1.2 ALU Control

The Alu control takes 2 inputs, the first being the bits 5-0 from the instruction to determine the function. The second input is the ALUop signal from the controller to which operation is going to be used in the ALU (e.g. add for load and store ,subtract for beq or determined by funct(10) for R-format).

8.2 DataPath

8.2.1 PC

PC is the signal that refers to the instruction the program is currently processing. It is always stored in a flopr. This is always incremented by 4 after the processing on the current instruction is finished to get the next instruction. After incrementing the PC, it is then sent to jumpPC and beqPC. It is then sent to 2 muxs alongside the output of the jumpPC and beqPC to determine the address of the next instruction. That address is then returned as an input to the flopr holding the PC.

8.2.2 jumpPC

jumpPC takes the bits 25 to 0 as an input from the instruction memory. These bits are shifted to the left by 2 and then concatenated with the last 4 bits of the PC after it was incremented.The output is then sent to the 2 muxs to determine the address as mentioned in the PC section.

8.2.3 beqPC

In the instruction, the bits 15 to 0 are sent to the sign extension then shifted left by 2 and finally added to the PC after it was incremeneted. The output is then sent to the 2 muxs to determine the address as mentioned in the PC section.

8.2.4 Register Files

Register files takes 5 inputs. These are Read register1(instruction bits 25 to 21) , Read register2(instruction 20 to 16) , Write register(the output of the mux that picks the instruction bits according to the register destination sent from the control), Write data (output of the mux in data memory that will be discussed later) and finally Register write (sent from the controller to enable the Write data to be stored in the destination register).It has 2 outputs; read data1(to be inputted in the ALU) and read data2(to be sent to the mux alongside the output of the sign extension).

8.2.5 ALU

The ALU takes inputs read data1 from the register files and the output of the mux that picks between the output of the sign extension and read data2 that depends on the ALUSrc coming from the controller. It also determines the operation according to 2 bits sent from the ALU Control. It then outputs the zflag and the result that is sent to the data memory and a mux.

8.3 Data memory

The data memory takes the address (which is the result of the ALU), write data (which is read data2 from the register file) and MemWrite (that determines wether the data will be written or read).It then outputs read data which will be an input in a mux with the ALU result as mentioned before, the selection line of this mux is memtoreg (coming from control). The output of this mux is then the write data of the register file.

8.4 Instruction memory

It takes 1 input which is PC(bits 7 to 2) ; the PC must be divided by 4 because the instruction memory increments by 1 only and not by bytes. It then outputs the new instruction to be processed.

8.5 MIPS module

It's the module that includes the port maps (the controller and the data path). It takes all the signals from the controller to input them in the data path.

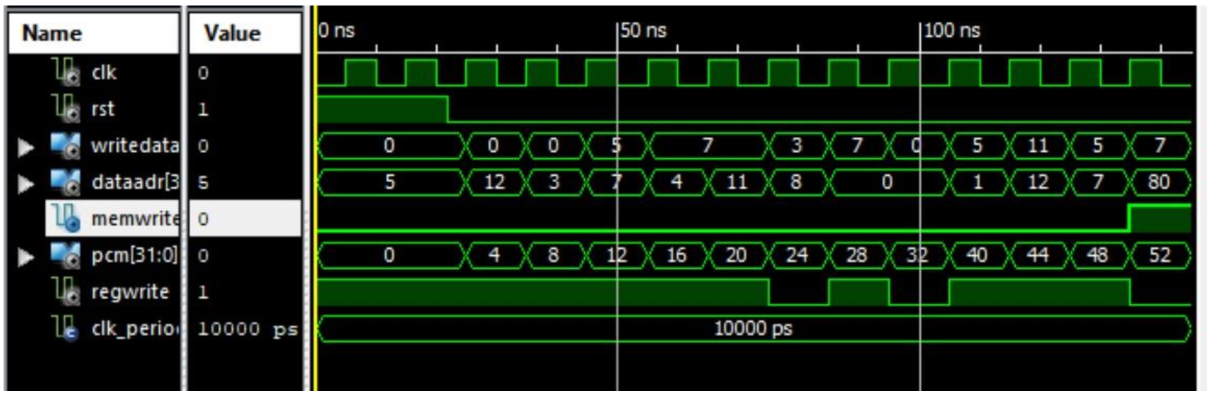
8.6 Processor module

It's the module that includes the port maps of everything (MIPS module, Instruction memory and data memory) and connect all 3 modules with each other.

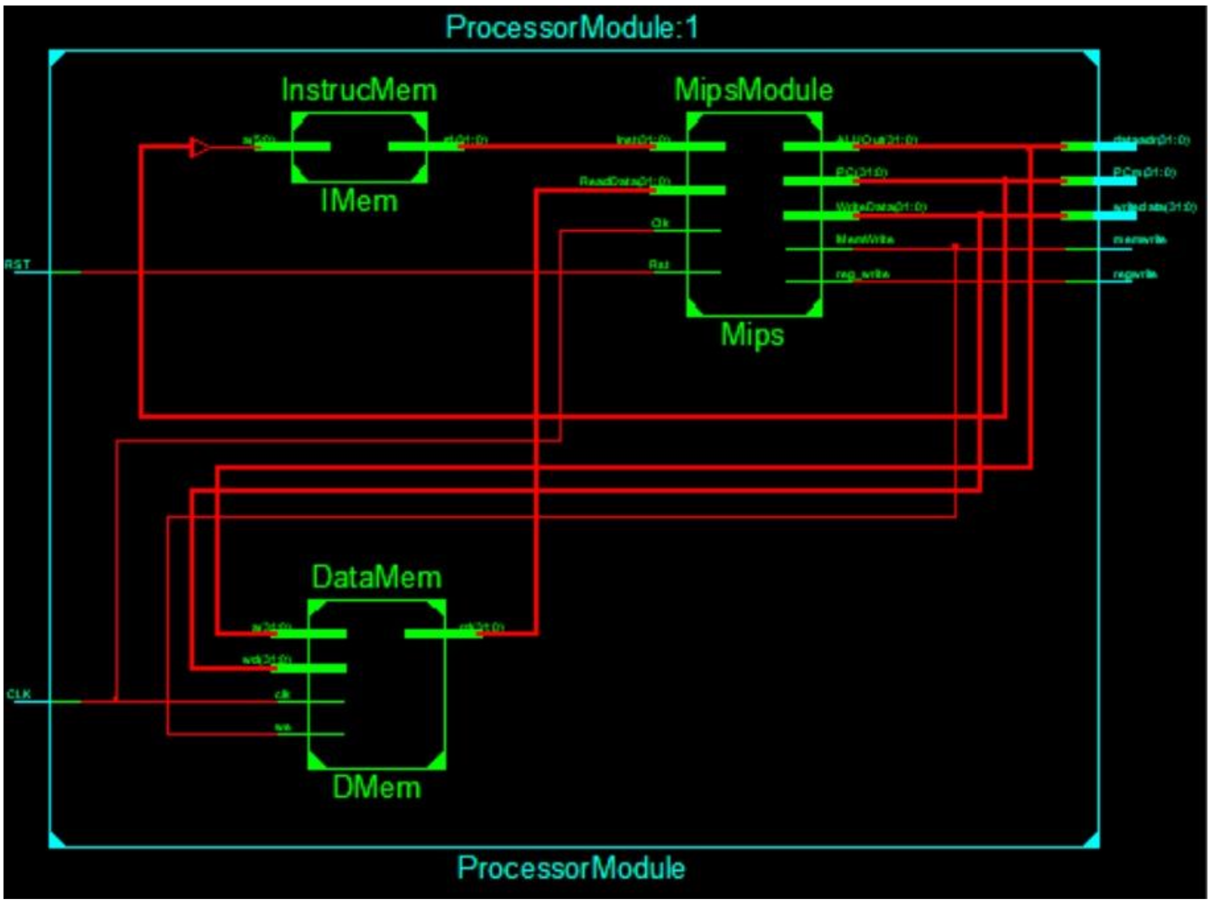
9.0 Simulation and Testing

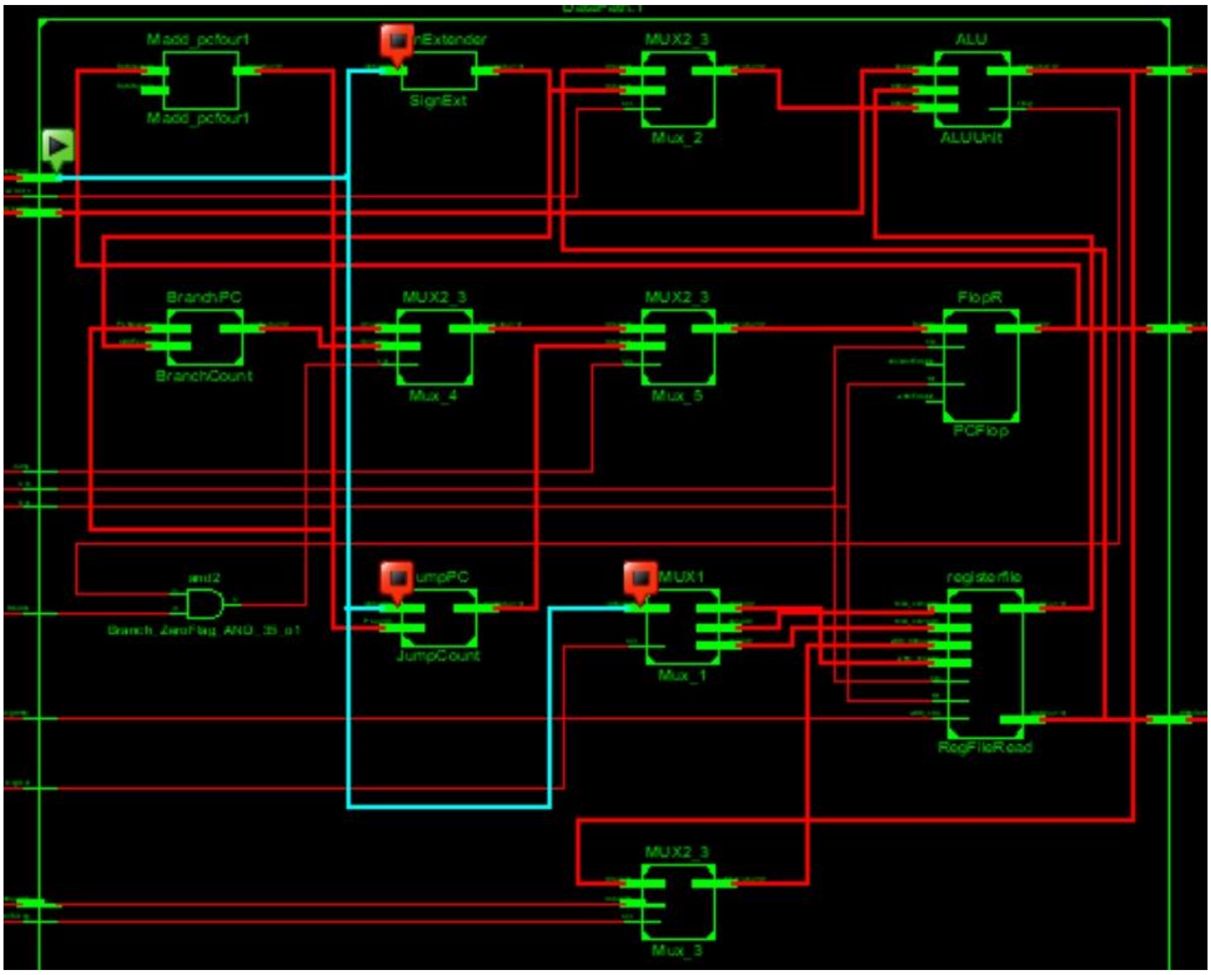
Using the test bench and the data we were provided with here were the results

9.1 The simulation



9.2 The RL schematic





10.0 Contribution Sheet 2

Roger Sherif Salama 22P0112	Alu Control, Datapath, Testing, report
Abdelrahman Mostafa Aly 22P0150	Controller, Datapath, Testing, report
Omar Ahmed Dardir 22P0218	Datapath, Testing, jumpInst, report
Andrew Rami Bassily 22P0187	Datapath, Testing, BranchInst, report
Mohamed Ashraf Mohamed 22P0210	Datapath, Program counter, Testing, report

11.0 REFERENCES

- Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design (5th ed.). Elsevier.
- Lab Slides by Eng. Yasmeen Shaban
- https://youtu.be/GaH5UIKMEus?si=DwZjGiwUQQBWzb8o
- https://youtube.com/playlist?list=PLW7Cvy3HywwzSx7xuE0e_eq8n4TeY1eWo&si=D00ZbRS55n2ajSSA