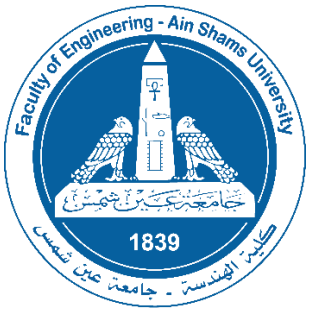


University of East London

Ain Shams University – Faculty of Engineering I-CHEP

UEL - University of East London

MIPS Processor Design in VHDL

Phase 2

Technical Project Report

Logic Design and Computer Organization – CSE221

1 Table of Contents

2 Introduction

The primary objective of this project was to enhance a MIPS CPU to support not only R-type instructions but also, I-type (e.g., lw, sw, beq) and J-type instructions. This required designing and integrating a control module, modifying the Datapath, and connecting instruction and data memory modules. MIPS is a simple, yet powerful instruction set architecture that demonstrates key concepts in computer architecture. By implementing this project, we aimed to understand the design, implementation, and simulation of a functional MIPS processor capable of executing a simple program.

3 Project Description

3.1 Requirements

The project involved several tasks: implementing the control module to generate control signals, connecting the control module to the datapath, integrating memory modules, writing a program to test the CPU, and simulating its execution to verify correctness.

3.2 Scope

The focus was on creating a single-cycle MIPS CPU capable of handling R, I, and J instruction types. This implementation does not include advanced features like pipelining or branch prediction but lays the groundwork for future enhancements.

4 Design and Implementation

4.1 Control Module

The control module is crucial in generating control signals based on the opcode and function code of the instruction. For this project, we implemented logic to generate signals such as RegWrite, ALUSrc, MemRead, MemWrite, and Branch. The control unit was designed to decode the instruction and activate the appropriate datapath components for R, I, or J instructions. For example:

- R-type instructions activated RegWrite and selected the ALU operation using the ALUOp signal.
- I-type instructions like lw and sw used ALUSrc to select the immediate value.
- The beq instruction utilized the Branch signal to control the program counter.

4.2 Datapath

The datapath is the core of the CPU, responsible for executing instructions. It was designed to include components such as the register file, ALU, multiplexer, and program counter. Modifications were made to:

- Handle immediate values for I-type instructions by adding a sign extender.
- Add logic for branching (beq) using a comparator and a mechanism to calculate the target address.
- Support J-type instructions by incorporating a jump mechanism to update the program counter directly.

4.3 Memory Modules

The CPU was connected to two memory modules:

- Instruction Memory: This stored the program to be executed. It was initialized with a set of instructions in assembly format to test the functionality of R, I, and J instructions.
- Data Memory: This was used to store and retrieve data during the execution of lw and sw instructions. The memory was designed to interface seamlessly with the datapath and control module.

4.4 Integration of MIPS Module

The control module and datapath were integrated to form the MIPS CPU. The instruction memory was connected to fetch instructions, and the data memory interacted with the CPU during load and store operations. This integration ensured smooth communication among all components, enabling the execution of a complete instruction cycle.

4.5 Program Execution

A simple program was written and loaded into the instruction memory. This program included R-type operations like addition and subtraction, I-type instructions like `lw`, `sw`, and `beq`, and a J-type jump instruction. The CPU was tested to ensure correct execution, with intermediate and final results verifying its functionality.

5 Simulation and Testing

5.1 Simulation Environment

The CPU was implemented and simulated using [your simulation tool, e.g., ModelSim or Quartus]. The simulation environment was configured to load the program into memory and execute it step-by-step.

5.2 Test Cases

Several test cases were designed, including:

1. R-type: Verifying operations like addition and subtraction.
2. I-type: Testing lw and sw for memory access and beq for conditional branching.
3. J-type: Testing jump instructions.

5.3 Results

The simulation outputs, including waveforms, confirmed the correct execution of all instruction types. For example:

- Register values updated as expected for arithmetic instructions.
- Memory contents reflected the correct behavior for lw and sw.
- The program counter updated correctly for branch and jump instructions.

5.4 Debugging

Initial testing revealed issues with branch target calculation and data memory access, which were resolved by refining the control signals and datapath connections.

6 Analysis and Discussion

The implemented CPU performed well in executing the given program. The single-cycle design ensured simplicity but limited performance due to the lack of pipelining. Challenges included ensuring synchronization

among modules and debugging logical errors in control signal generation. Through this project, we gained insights into CPU design, the role of control signals, and the intricacies of handling different instruction types.

7 Conclusion

This project successfully enhanced the MIPS CPU to handle R, I, and J instruction types. By integrating the control module, datapath, and memory modules, we created a functional CPU capable of executing a test program. Future improvements could include implementing a pipelined architecture, adding support for floating-point instructions, and optimizing memory access.

8 Appendices

8.1 Block Diagrams:

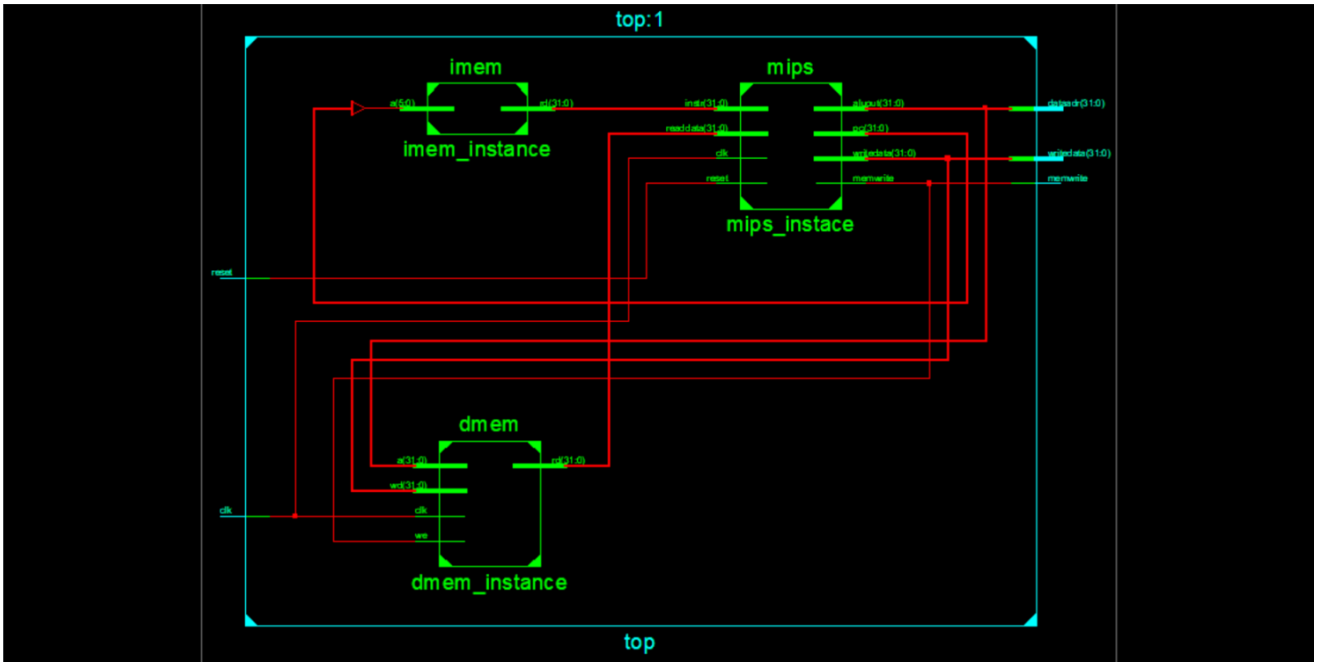


Figure 1 Top module Diagram

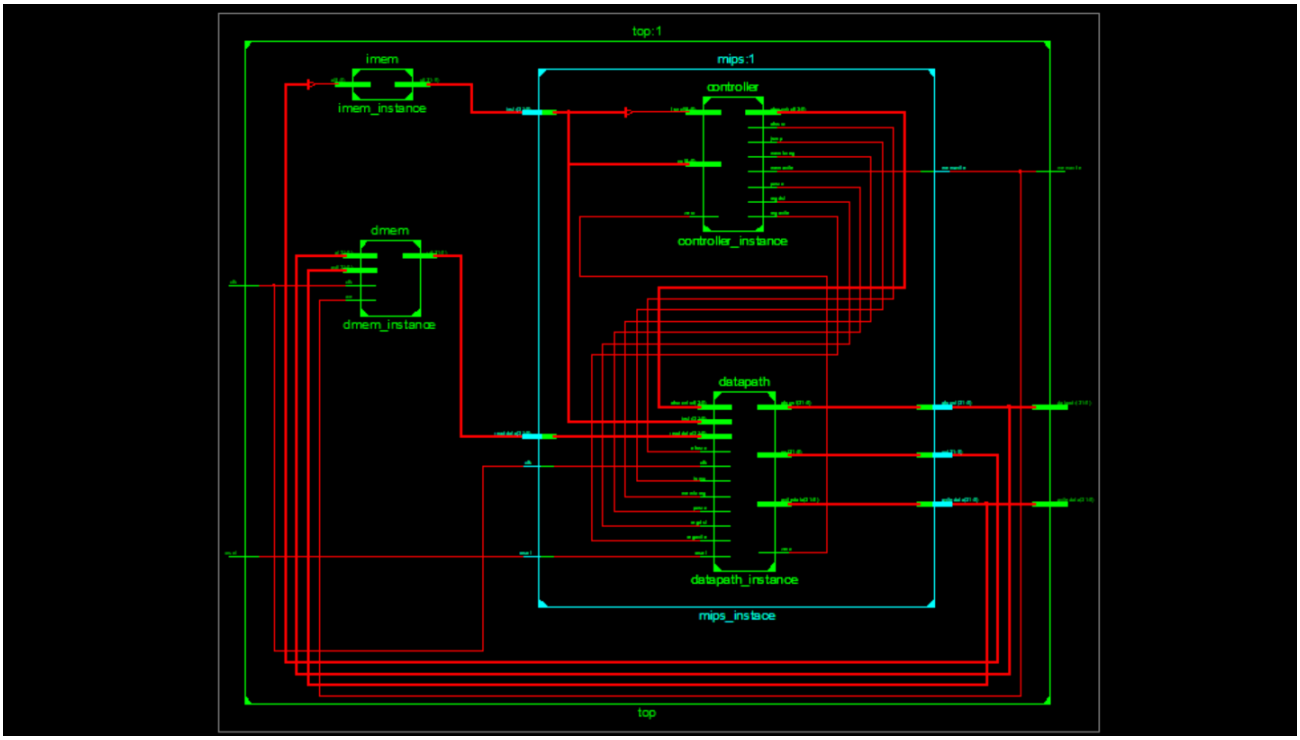


Figure 2 Depiction of the MIPS module

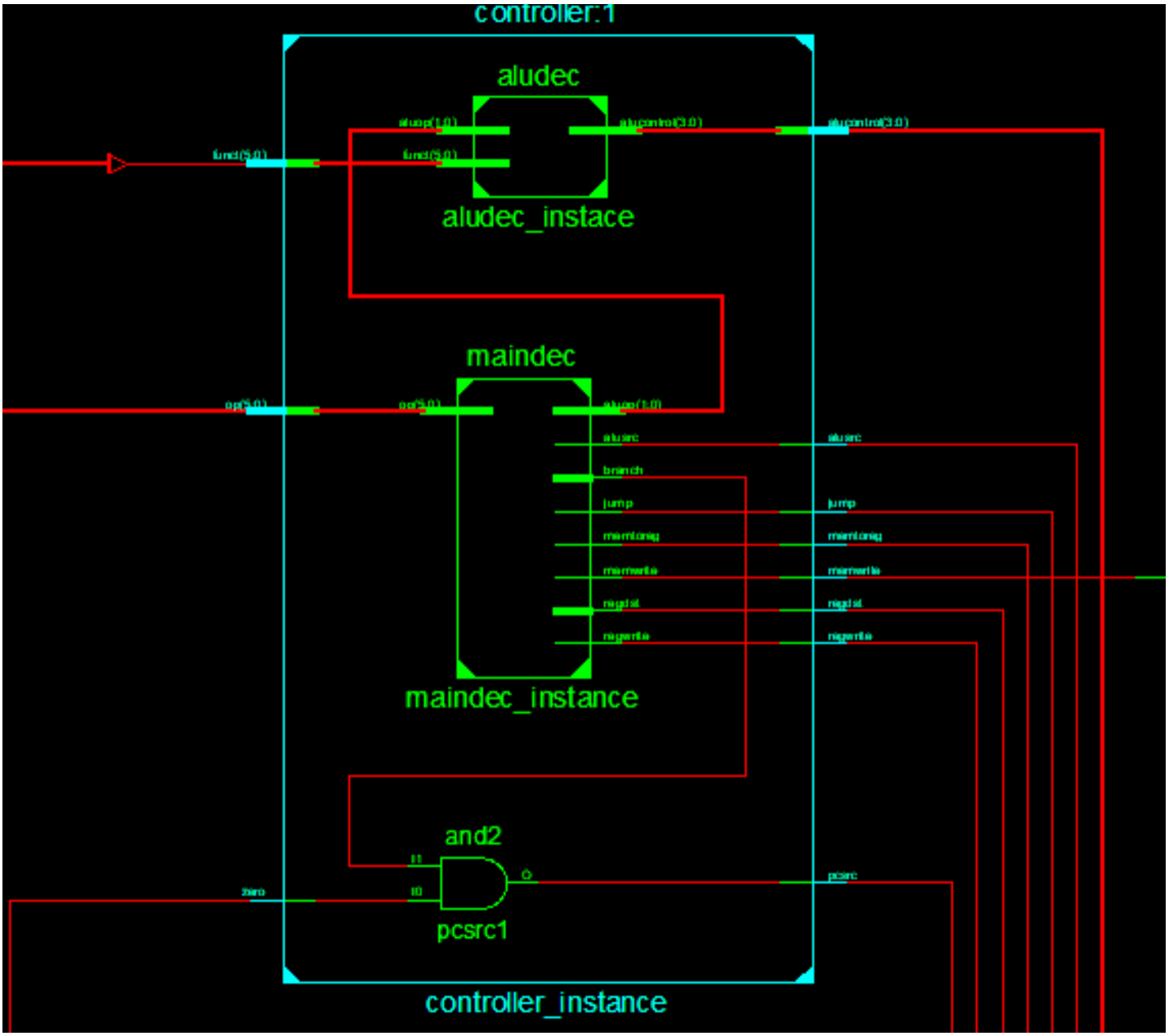


Figure 3 Controller Diagram

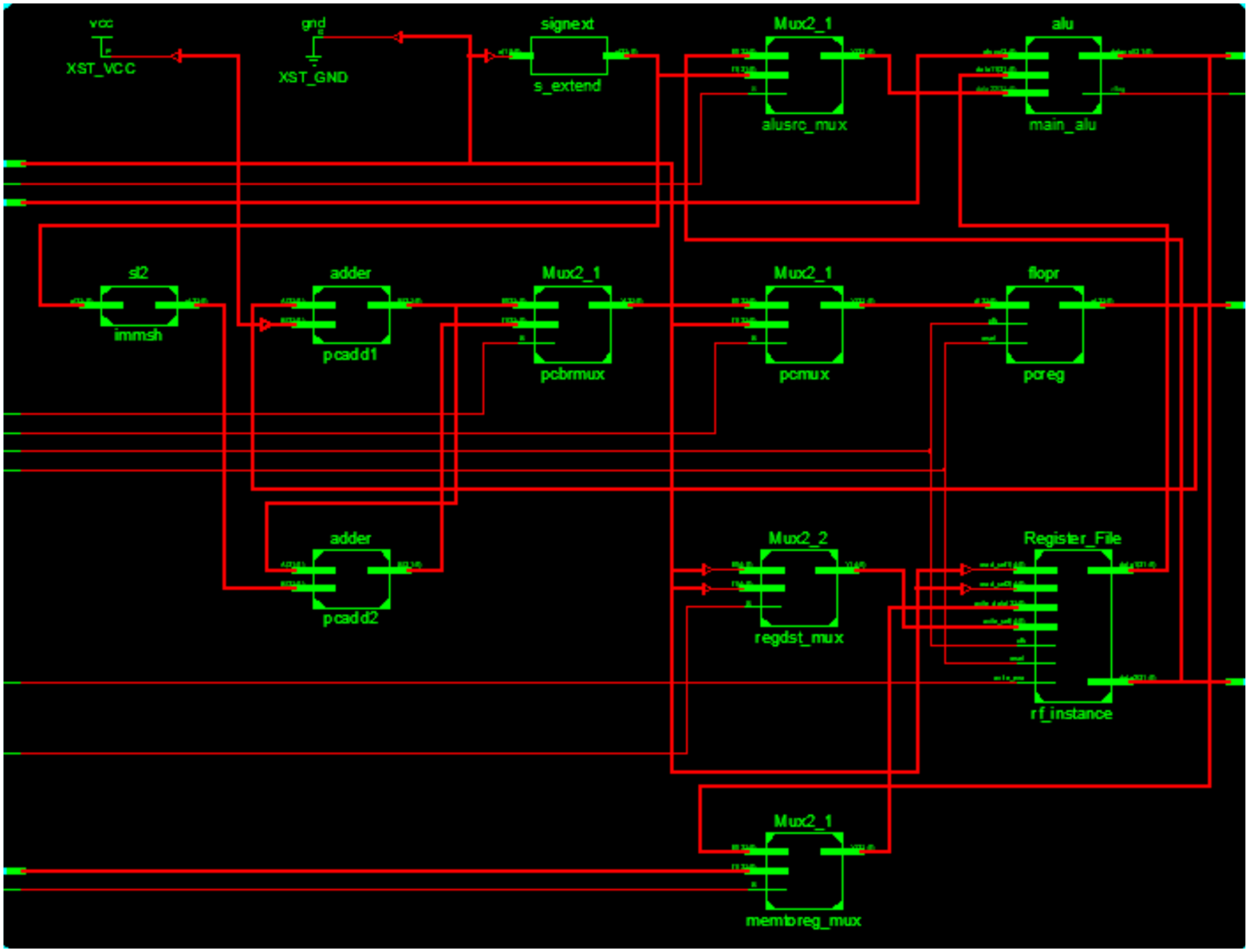
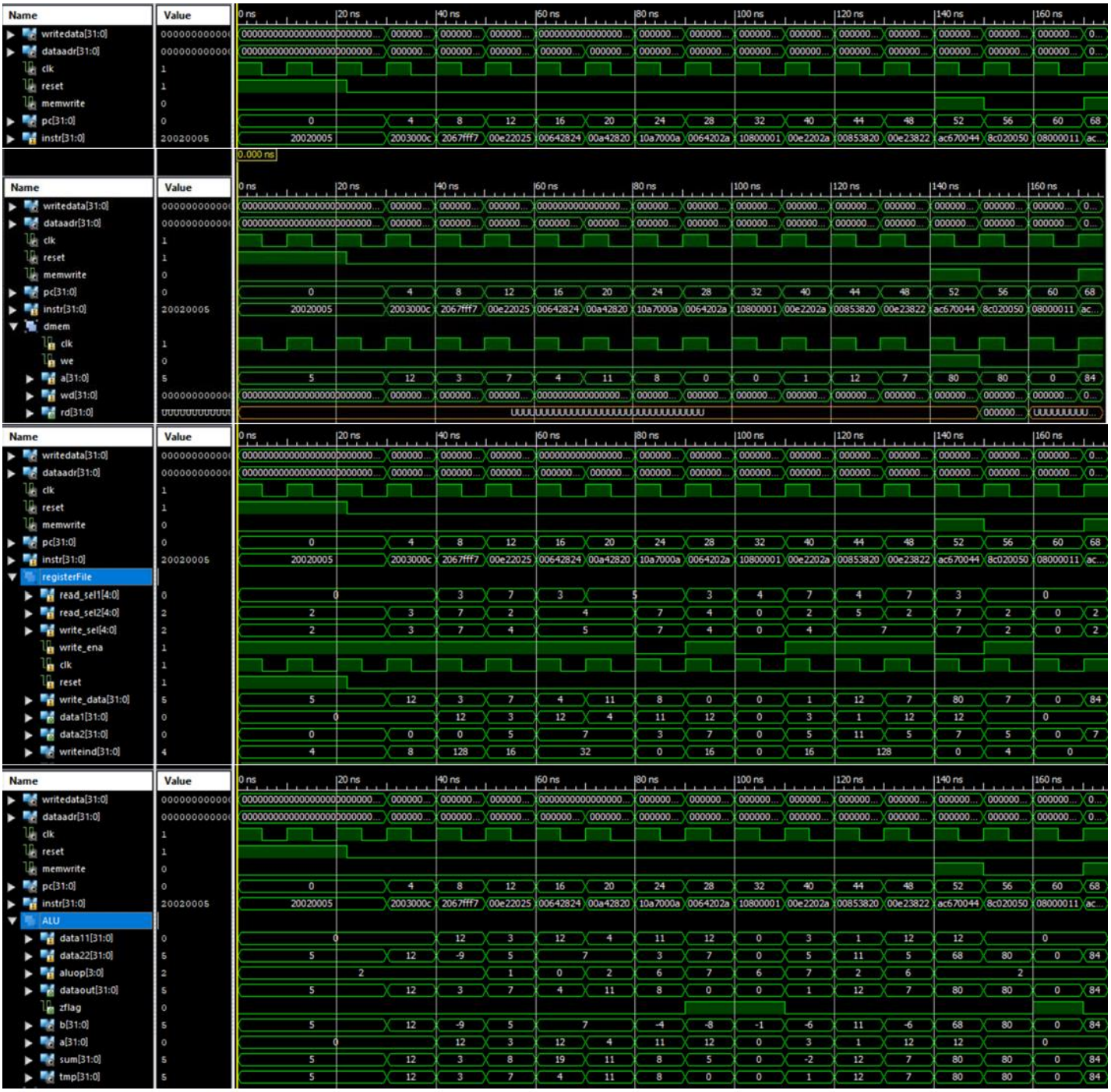


Figure 4 Datapath Diagram

8.2 Simulation Outputs:




```
** Failure:NO ERRORS: Simulation succeeded
User(VHDL) Code Called Simulation Stop
In process testbench.vhd:40

INFO: Simulator is stopped.
ISim>
```

9 Contribution

ID	Name	Contribution	Percentage
23P0279	Ibrahim Ahmed Abdelfattah	- MIPS - Top Module - Report - Datapath	20%
2300535	Mohamed Tarek Elsayed	- Datapath - Top Module - MIPS - Controller	20%
23P0314	Omar Ahmed Shabaan	- Datapath - Top Module - MIPS - Report	20%
23P0266	John George	- MIPS - Top Module - Datapath	20%
23P0192	Peter Maged	- Controller - ALU, Decoder - Main Decoder	20%

While the table above outlines individual contributions, it's important to emphasize that this project was a truly collaborative effort. Each team member actively participated in design discussions, code reviews, and troubleshooting sessions. We believe that the success of this project is a testament to our shared commitment to teamwork, open communication, and the equitable distribution of workload. We are proud of the collective effort we put forth and the knowledge we gained through this collaborative experience.

A. Source Code:

```
use IEEE.STD_LOGIC_1164.all;
use work.top_package.ALL;
entity top is -- top-level design for testing
port(clk, reset: in STD_LOGIC;
     writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
     memwrite: out STD_LOGIC);
end top;

architecture Behavioral of top is
      signal memwritet: STD_LOGIC;
      signal pc, instr, readdata, dataadrt, writedatat: STD_LOGIC_VECTOR(31 downto 0);
begin

      mips_instace: Mips
                port map(
                           clk => clk,
                           reset => reset,
                           pc => pc,
                           instr => instr,
                           memwrite => memwritet,
                           aluout => dataadrt ,
                           writedata => writedatat,
                           readdata =>  readdata
                );

      dmem_instance: dmem
                port map(
                           clk => clk,
                           we => memwritet,
                           a =>  dataadrt,
                           wd => writedatat,
                           rd => readdata
                );
      imem_instance: imem
                port map(
                           a  => pc(7 downto 2),
                           rd => instr
                );

      memwrite <= memwritet;
      dataadr <= dataadrt;
      writedata <= writedatat;

end Behavioral;

```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.mips_package.ALL;
entity mips is -- single cycle MIPS processor
```

```
 5.   port (
 6.     clk, reset: in STD_LOGIC;
 7.     pc: buffer STD_LOGIC_VECTOR(31 downto 0);
 8.     instr: in STD_LOGIC_VECTOR(31 downto 0);
 9.     memwrite: out STD_LOGIC;
10.     aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
11.     readdata: in STD_LOGIC_VECTOR(31 downto 0)
12.   );
13. end entity;
14.
15. architecture Behavioral of mips is
16.       signal memtoreg, alusrc, regdst, regwrite, jump, pcsrc: STD_LOGIC;
17.       signal zero: STD_LOGIC;
18.       signal alucontrol: STD_LOGIC_VECTOR(3 downto 0);
19.       signal alu_out, write_data: STD_LOGIC_VECTOR(31 downto 0);
20. begin
21.       datapath_instance: datapath
22.                 port map(
23.                       clk => clk,
24.                                       reset => reset,
25.                                       memtoreg => memtoreg,
26.                                       pcsrc => pcsrc,
27.                                       alusrc => alusrc,
28.                                       regdst => regdst,
29.                                       regwrite => regwrite,
30.                                       jump => jump,
31.                                       alucontrol => alucontrol,
32.                                       zero => zero,
33.                                       pc => pc,
34.                                       instr => instr,
35.                                       writedata => writedata,
36.                                       aluout => aluout,
37.                                       readdata => readdata
38.                 );
39.    controller_instance: controller
40.                 port map(
41.                                        op => instr(31 downto 26),
42.                                        funct => instr(5 downto 0),
43.                                        zero => zero,
44.                                        memtoreg => memtoreg,
45.                                        memwrite => memwrite,
46.                                        pcsrc => pcsrc,
47.                                        alusrc => alusrc,
48.                                        regdst => regdst,
49.                                        regwrite => regwrite,
50.                                        jump => jump,
51.                                        alucontrol => alucontrol
52.                 );
53.
54. end Behavioral;
55.
```

```
 1. library IEEE;
 2. use IEEE.STD_LOGIC_1164.ALL;
 3. use work.data_path_package.ALL;
 4.
 5. entity datapath is
```

```
    -- MIPS datapath
    port (
        clk, reset      : in  STD_LOGIC;
        memtoreg, pcsrc : in  STD_LOGIC;
        alusrc, regdst  : in  STD_LOGIC;
        regwrite, jump  : in  STD_LOGIC;
        alucontrol      : in  STD_LOGIC_VECTOR(3 downto 0);
        zero            : out STD_LOGIC;
        pc              : buffer STD_LOGIC_VECTOR(31 downto 0);
        instr           : in  STD_LOGIC_VECTOR(31 downto 0);
        writedata       : buffer STD_LOGIC_VECTOR(31 downto 0);
        aluout          : buffer STD_LOGIC_VECTOR(31 downto 0);
        readdata        : in  STD_LOGIC_VECTOR(31 downto 0)
    );
end datapath;

architecture Behavioral of datapath is
    -- Signal declarations
      signal pc_internal : STD_LOGIC_VECTOR(31 downto 0);
    signal writereg     : STD_LOGIC_VECTOR(4 downto 0);
    signal pcjump, pcnext, pcnextbr, pcplus4, pcbranch : STD_LOGIC_VECTOR(31 downto 0);
    signal signimm, signimmsh : STD_LOGIC_VECTOR(31 downto 0);
   -- signal srca, srcb, result : STD_LOGIC_VECTOR(31 downto 0);
    signal regdst_out_signal  : STD_LOGIC_VECTOR(4 downto 0);
    signal memtoreg_out_signal: STD_LOGIC_VECTOR(31 downto 0);
    signal alu_data1_out      : STD_LOGIC_VECTOR(31 downto 0);
    signal alu_data2_out      : STD_LOGIC_VECTOR(31 downto 0);
    signal alusrc_mux_out     : STD_LOGIC_VECTOR(31 downto 0);
      signal main_aluout_signal : STD_LOGIC_VECTOR(31 downto 0);
    signal alu_z_flag         : STD_LOGIC;
begin
    -- MUX and adder logic for the program counter (PC)
    pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";

    -- Register for storing the program counter (PC)
    pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc_internal);

    -- Adder to calculate the next PC (PC + 4)
    pcadd1: adder port map(pc_internal, X"00000004", pcplus4);

    -- Shift left operation for the immediate value (sign-extended)
    immsh: sl2 port map(signimm, signimmsh);

    -- Adder for calculating the branch address (PC + branch offset)
    pcadd2: adder port map(pcplus4, signimmsh, pcbranch);

    -- MUX to select between the PC + 4 or the branch target
    pcbrmux: Mux2 generic map(32) port map(pcplus4, pcbranch, pcsrc, pcnextbr);

    -- MUX to select between the branch target or the jump target
    pcmux: Mux2 generic map(32) port map(pcnextbr, pcjump, jump, pcnext);

    regdst_mux: Mux2 generic map(5)
        port map (
            I0 => instr(20 downto 16),
            I1 => instr(15 downto 11),
            S  => regdst,
            Y  => regdst_out_signal
        );
```

```
65.     rf_instance: Register_File
66.         port map (
67.             read_sel1  => instr(25 downto 21),
68.             read_sel2  => instr(20 downto 16),
69.             write_sel  => regdst_out_signal,
70.             write_ena  => regwrite,
71.             clk        => clk,
72.                                   reset      => reset,
73.             write_data => memtoreg_out_signal,
74.             data1      => alu_data1_out,
75.             data2      => alu_data2_out
76.         );
77.     s_extend: signext
78.         port map (
79.             a => instr(15 downto 0),
80.             y => signimm
81.         );
82.     alusrc_mux: Mux2 generic map(32)
83.         port map (
84.             I0 => alu_data2_out,
85.             I1 => signimm,--sign_extend_out,
86.             S  => alusrc,
87.             Y  => alusrc_mux_out
88.         );
89.     main_alu: alu
90.         port map (
91.             data11  => alu_data1_out,
92.             data22  => alusrc_mux_out,
93.             aluop   => alucontrol,
94.             dataout => main_aluout_signal,
95.             zflag   => alu_z_flag
96.         );
97.
98.     memtoreg_mux: Mux2 generic map(32)
99.         port map (
100.            I0 => main_aluout_signal,
101.            I1 => readdata,
102.            S  => memtoreg,
103.            Y  => memtoreg_out_signal
104.        );
105.
106.                        pc<= pc_internal;
107.                        zero <= alu_z_flag;
108.                        aluout<= main_aluout_signal;
109.                        writedata <= alu_data2_out;
110.
111. end Behavioral;
112.
```

```
1. library IEEE;
2. use IEEE.STD_LOGIC_1164.ALL;
3. use work.controller_package.ALL;
4.
5. entity controller is
6.  port (
7.     op, funct: in STD_LOGIC_VECTOR(5 downto 0);
8.     zero: in STD_LOGIC;
```

```
 9.     memtoreg, memwrite: out STD_LOGIC;
10.     pcsrc, alusrc: out STD_LOGIC;
11.     regdst, regwrite: out STD_LOGIC;
12.     jump: out STD_LOGIC;
13.     alucontrol: out STD_LOGIC_VECTOR(3 downto 0)
14.   );
15. end controller;
16.
17. architecture Behavioral of controller is
18.       signal aluop: STD_LOGIC_VECTOR(1 downto 0);
19.       signal branch: STD_LOGIC;
20. begin
21.       maindec_instance: maindec
22.                 port map(
23.                    op => op,
24.                           memtoreg => memtoreg ,
25.                           memwrite => memwrite,
26.                           branch => branch ,
27.                           alusrc => alusrc,
28.                           regdst => regdst,
29.                           regwrite => regwrite,
30.                           jump => jump,
31.                           aluop => aluop
32.                 );
33.       aludec_instace: aludec
34.                 port map(
35.                    funct => funct,
36.                           aluop => aluop,
37.                           alucontrol => alucontrol
38.                 );
39.
40.       pcsrc <= zero and branch;
41.
42. end Behavioral;
43.
```

```
 1. library IEEE;
 2. use IEEE.STD_LOGIC_1164.ALL;
 3. entity maindec is
 4.   port (
 5.     op: in STD_LOGIC_VECTOR(5 downto 0);
 6.     memtoreg, memwrite: out STD_LOGIC;
 7.     branch, alusrc: out STD_LOGIC;
 8.     regdst, regwrite: out STD_LOGIC;
 9.     jump: out STD_LOGIC;
10.     aluop: out STD_LOGIC_VECTOR(1 downto 0)
11.   );
12. end entity;
13. architecture maindecLogic of maindec is
14.   signal controls: STD_LOGIC_VECTOR(8 downto 0);
15. begin
16.
17.   process (op) is
18.   begin
19.     case op is
20.       when "000000" => controls <= "110000010"; -- RTYPE
21.       when "100011" => controls <= "101001000"; -- LW
```

```
22.       when "101011" => controls <= "001010000"; -- SW
23.       when "000100" => controls <= "000100001"; -- BEQ
24.       when "001000" => controls <= "101000000"; -- ADDI
25.       when "000010" => controls <= "000000100"; -- J
26.       when others => controls <= "---------"; -- illegal op
27.     end case;
28.   end process;
29.
30.   (regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump,
31.    aluop(1), aluop(0)) <= controls;
32.
33. end architecture;
34.
```

```
 1. library IEEE;
 2. use IEEE.STD_LOGIC_1164.all;
 3. -- ALU control decoder
 4. entity aludec is
 5.   port (
 6.     funct: in STD_LOGIC_VECTOR(5 downto 0);
 7.     aluop: in STD_LOGIC_VECTOR(1 downto 0);
 8.     alucontrol: out STD_LOGIC_VECTOR(3 downto 0)
 9.   );
10. end entity;
11. architecture behave of aludec is
12. begin
13.
14.   process (aluop, funct) is
15.   begin
16.     case aluop is
17.       when "00" => alucontrol <= "0010"; -- add (for lw/sw/addi)
18.       when "01" => alucontrol <= "0110"; -- sub (for beq)
19.       when others =>
20.         case funct is -- R-type instructions
21.           when "100000" => alucontrol <= "0010"; -- add
22.           when "100010" => alucontrol <= "0110"; -- sub
23.           when "100100" => alucontrol <= "0000"; -- and
24.           when "100101" => alucontrol <= "0001"; -- or
25.           when "101010" => alucontrol <= "0111"; -- slt
26.           when others => alucontrol <= "----"; -- ???
27.         end case;
28.     end case;
29.   end process;
30.
31. end architecture;
32.
```

```
 1. library IEEE;
 2. use IEEE.STD_LOGIC_1164.ALL;
 3. use IEEE.STD_LOGIC_SIGNED.ALL;
 4. use IEEE.STD_LOGIC_ARITH.ALL;
 5. use IEEE.NUMERIC_STD.ALL;
 6.
 7. entity dmem is -- Data memory
 8.     port (
 9.         clk, we : in  STD_LOGIC;
```

```
10.         a, wd   : in  STD_LOGIC_VECTOR(31 downto 0);
11.         rd      : out STD_LOGIC_VECTOR(31 downto 0)
12.     );
13. end dmem;
14.
15. architecture behave of dmem is
16. begin
17.     process is
18.         type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
19.         variable mem : ramtype;
20.     begin
21.         -- Read or write memory
22.         for i in 1 to 1000 loop
23.             if rising_edge(clk) then
24.                 if (we = '1') then
25.                     mem(CONV_INTEGER('0' & a(7 downto 2))) := wd;
26.                 end if;
27.             end if;
28.
29.             rd <= mem(CONV_INTEGER('0' & a(7 downto 2)));
30.             wait on clk, a;
31.         end loop;
32.     end process;
33. end behave;
34.
```

```
 1. library IEEE;
 2. use IEEE.STD_LOGIC_1164.ALL;
 3. use IEEE.STD_LOGIC_SIGNED.ALL;
 4. use IEEE.STD_LOGIC_ARITH.ALL;
 5. use IEEE.NUMERIC_STD.ALL;
 6. use IEEE.STD_LOGIC_TEXTIO.ALL;
 7. library STD;
 8. use STD.TEXTIO.ALL;
 9.
10. entity imem is -- instruction memory
11.     port (
12.         a  : in  STD_LOGIC_VECTOR(5 downto 0);
13.         rd : out STD_LOGIC_VECTOR(31 downto 0)
14.     );
15. end imem;
16.
17. architecture behave of imem is
18. begin
19.     process is
20.         file mem_file : TEXT;
21.         variable L    : line;
22.         variable ch   : character;
23.         variable i, index, result : integer;
24.         type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
25.         variable mem : ramtype;
26.     begin
27.         -- Initialize memory from file
28.         for i in 0 to 63 loop
29.             mem(i) := (others => '0'); -- Set all contents to '0'
30.         end loop;
31.
```

```
32.         index := 0;
33.         FILE_OPEN(mem_file, "C:\Users\oa508\Documents\CO_project_phase2_v2\memfile.dat", READ_MODE);
34.
35.         while not endfile(mem_file) loop
36.             readline(mem_file, L);
37.             result := 0;
38.
39.             for i in 1 to 8 loop
40.                 read(L, ch);
41.                 if '0' <= ch and ch <= '9' then
42.                     result := character'pos(ch) - character'pos('0');
43.                 elsif 'a' <= ch and ch <= 'f' then
44.                     result := character'pos(ch) - character'pos('a') + 10;
45.                 else
46.                     report "Format error on line " & integer'image(index) severity error;
47.                 end if;
48.
49.                 mem(index)(35 - i * 4 downto 32 - i * 4) := std_logic_vector(to_unsigned(result, 4));
50.             end loop;
51.
52.             index := index + 1;
53.         end loop;
54.
55.         -- Read memory
56.         for i in 1 to 1000 loop
57.             rd <= mem(CONV_INTEGER(a));
58.             wait on a;
59.         end loop;
60.     end process;
61.
62. end;
63.
```