HARDWARE DESIGN PROJECT

IMPLEMENTATION OF MIPS ISA 32-BIT PROCESSOR

Under the supervision of

Dr. Howida Abd AlLatif

TABLE OF CONTENTS

TEAM MEMBERS

NO.	Name	Tasks completed
1	Basmala Muhammed Abd El-Hakim	Implemented Data memory, Contributed in Instruction memory, made the presentation
2	Fayza Mahmoud Bakry	Implemented Instruction memory, mux, and ALU
3	Menna Tawfiq Nasr El-Dein	Implemented the Control unit and assembly, and made the report
4	Sondos Said El-Sayed	Implemented Registers file, Sign extend, PC, PC peripherals and schematic, assembly

1. ASSIGNMENT AIM

The aim of this project is to design and implement a single cycle processor adhering to the MIPS (Microprocessor without Interlocked Pipeline Stages) Instruction Set Architecture (ISA) for 32-bit processing. The project will involve creating a VHDL (VHSIC Hardware Description Language) implementation of the processor, focusing on achieving accurate instruction execution within a single clock cycle.

The primary objectives of this assignment are as follows:

1. **Understanding MIPS ISA:** Gain a comprehensive understanding of the MIPS ISA, including its instruction formats, data types, addressing modes, and control flow mechanisms. This includes familiarity with MIPS assembly language programming.
2. **Processor Design:** Design a single cycle processor capable of executing instructions from the MIPS ISA. This involves designing the data path, control unit, instruction decoding, register file, memory interface, and arithmetic logic unit (ALU) components.
3. **Instruction Execution:** Implement the necessary logic to ensure correct execution of MIPS instructions within a single clock cycle. This requires careful consideration of instruction decoding, operand fetching, execution, and result writing back to registers or memory.

2. PROBLEM SOLUTION

Implementing a single-cycle processor that adheres to the MIPS ISA 32-bit processor presents several challenges and opportunities. The core challenge lies in designing a processor architecture capable of executing instructions within a single clock cycle while conforming to the MIPS instruction set architecture. This entails designing an efficient control unit, data path, and memory hierarchy to ensure timely execution of instructions.

Our solution involves breaking down the problem into manageable components and designing each component with careful consideration of its functionality and interaction with other parts of the processor. We adopt a modular design approach to facilitate ease of testing, debugging, and scalability. Additionally, we leverage VHDL as the hardware description language to model and simulate our processor design.

2.1 INPUTS AND OUTPUTS:

The inputs and outputs of our single-cycle processor are defined by the MIPS ISA 32-bit architecture and encompass various signals and data paths within the processor.

Inputs:

- **Instruction Memory Input:** Fetches 32-bit instructions from memory based on the program counter (PC) value.
- **Control Signals:** Signals from the control unit indicating the operation to be performed by the processor (e.g., read, write, ALU operation).
- **Data Memory Input:** Data inputs for memory read and write operations.
- **Register File Inputs:** Read data signals for accessing register contents.
- **Immediate Values:** Immediate values extracted from instructions for arithmetic, logical, and control operations.
- **Branch Targets:** Addresses for branch instructions to facilitate control flow.
- **Clock Signal:** Synchronizes the operation of various components within the processor.

Outputs:

- **Register File Outputs:** Data outputs from the register file.
- **ALU Result:** Result of arithmetic or logical operations performed by the ALU.
- **Memory Data Output:** Data read from memory.
- **Write Data Address:** Address for writing data to memory.

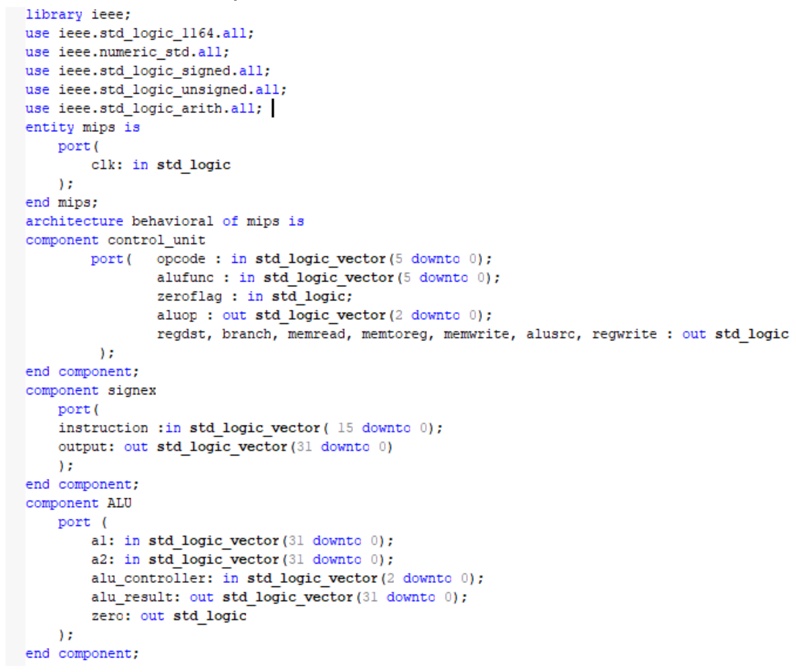
- **Branch and Jump Control Signals:** Signals indicating whether to branch or jump based on specific conditions.
- **Program Counter (PC):** Address of the next instruction to be fetched from the instruction memory.
- **Control Signals:** Signals indicating the control signals for various components within the processor.

These inputs and outputs collectively enable the processor to execute instructions according to the MIPS ISA, perform arithmetic and logical operations, manage control flow, and interact with memory. By carefully defining and managing these signals and data paths, we ensure the proper functioning and compliance of our single-cycle processor with the MIPS architecture.

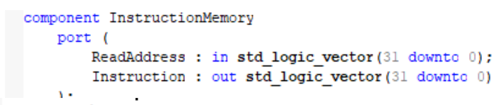
3. IMPLEMENTATION

Code on our GitHub repository MIPS

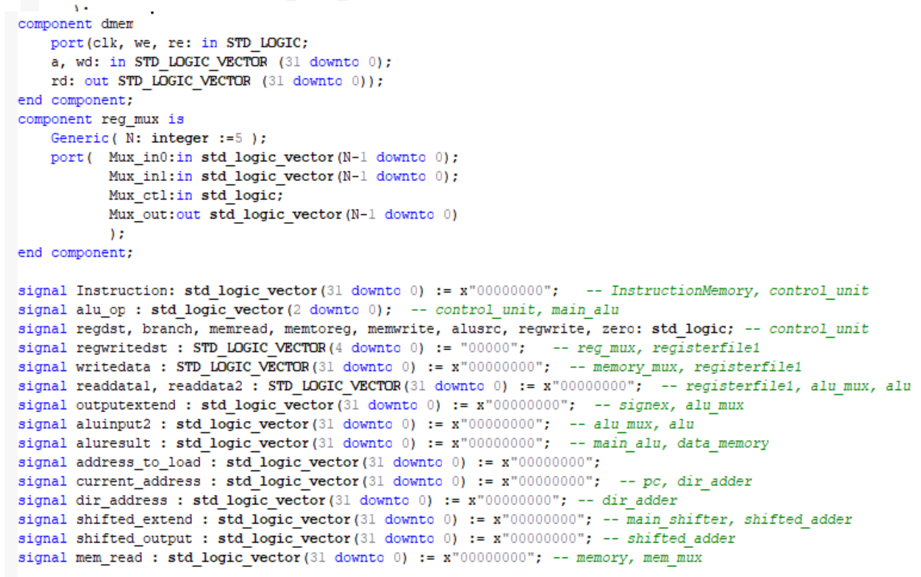
- Assembly:



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity mips is
    port(
        clk: in std_logic
    );
end mips;
architecture behavioral of mips is
component control_unit
        port(   opcode : in std_logic_vector(5 downto 0);
                alufunc : in std_logic_vector(5 downto 0);
                zeroflag : in std_logic;
                aluop : out std_logic_vector(2 downto 0);
                regdst, branch, memread, memtoreg, memwrite, alusrc, regwrite : out std_logic
         );
end component;
component signex
    port(
    instruction :in std_logic_vector( 15 downto 0);
    output: out std_logic_vector(31 downto 0)
    );
end component;
component ALU
    port (
        a1: in std_logic_vector(31 downto 0);
        a2: in std_logic_vector(31 downto 0);
        alu_controller: in std_logic_vector(2 downto 0);
        alu_result: out std_logic_vector(31 downto 0);
        zero: out std_logic
    );
end component;
```



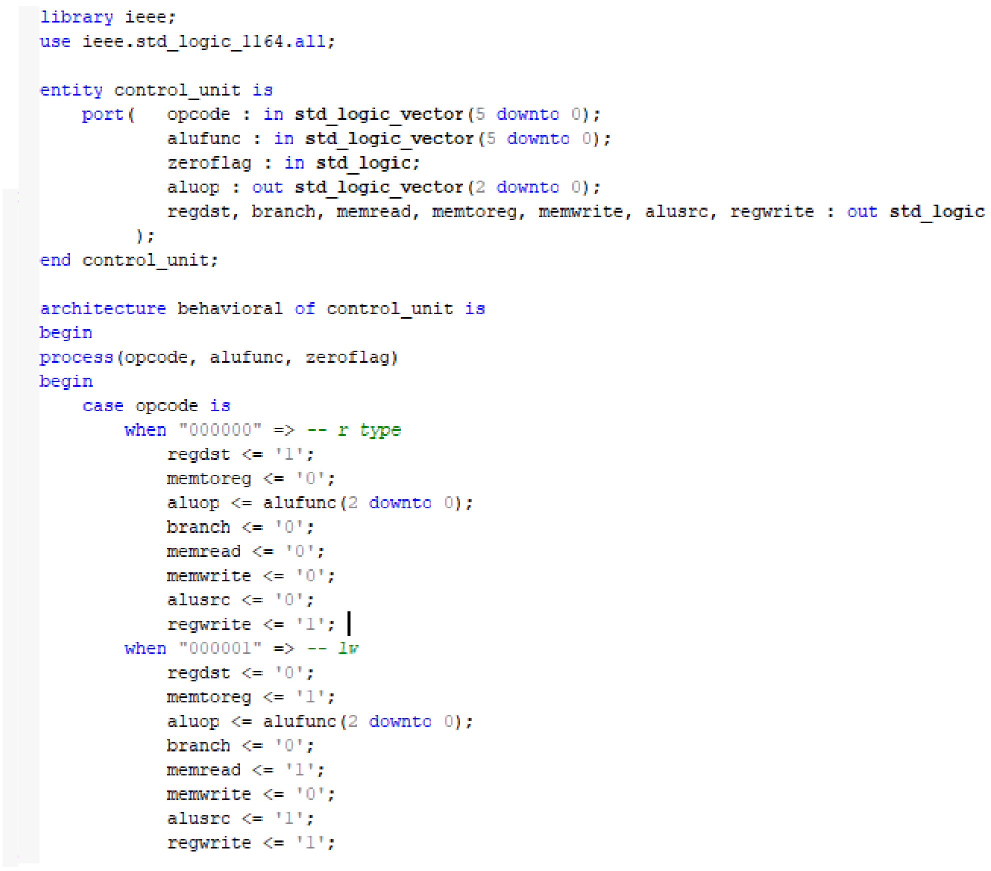
```vhdl
component InstructionMemory
    port (
        ReadAddress : in std_logic_vector(31 downto 0);
        Instruction : out std_logic_vector(31 downto 0)
    );
```



```vhdl
component dmem
    port(clk, we, re: in STD_LOGIC;
    a, wd: in STD_LOGIC_VECTOR (31 downto 0);
    rd: out STD_LOGIC_VECTOR (31 downto 0));
end component;
component reg_mux is
    Generic( N: integer :=5 );
    port(  Mux_in0:in std_logic_vector(N-1 downto 0);
           Mux_in1:in std_logic_vector(N-1 downto 0);
           Mux_ctl:in std_logic;
           Mux_out:out std_logic_vector(N-1 downto 0)
           );
end component;

signal Instruction: std_logic_vector(31 downto 0) := x"00000000";   -- InstructionMemory, control_unit
signal alu_op : std_logic_vector(2 downto 0);  -- control_unit, main_alu
signal regdst, branch, memread, memtoreg, memwrite, alusrc, regwrite, zero: std_logic; -- control_unit
signal regwritedst : STD_LOGIC_VECTOR(4 downto 0) := "00000";   -- reg_mux, registerfile1
signal writedata : STD_LOGIC_VECTOR(31 downto 0) := x"00000000";  -- memory_mux, registerfile1
signal readdata1, readdata2 : STD_LOGIC_VECTOR(31 downto 0) := x"00000000";  -- registerfile1, alu_mux, alu
signal outputextend : std_logic_vector(31 downto 0) := x"00000000";  -- signex, alu_mux
signal aluinput2 : std_logic_vector(31 downto 0) := x"00000000";  -- alu_mux, alu
signal aluresult : std_logic_vector(31 downto 0) := x"00000000";  -- main_alu, data_memory
signal address_to_load : std_logic_vector(31 downto 0) := x"00000000";
signal current_address : std_logic_vector(31 downto 0) := x"00000000";  -- pc, dir_adder
signal dir_address : std_logic_vector(31 downto 0) := x"00000000"; -- dir_adder
signal shifted_extend : std_logic_vector(31 downto 0) := x"00000000"; -- main_shifter, shifted_adder
signal shifted_output : std_logic_vector(31 downto 0) := x"00000000"; -- shifted_adder
signal mem_read : std_logic_vector(31 downto 0) := x"00000000"; -- memory, mem_mux
```

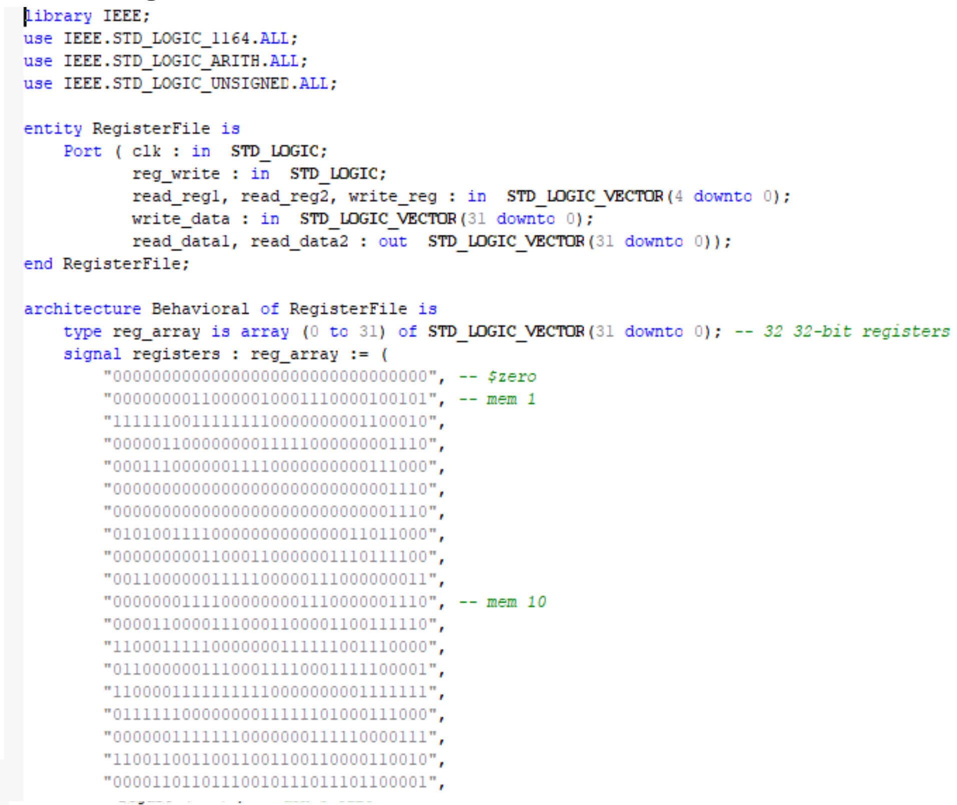
```vhdl
end component;
```

- Control unit:

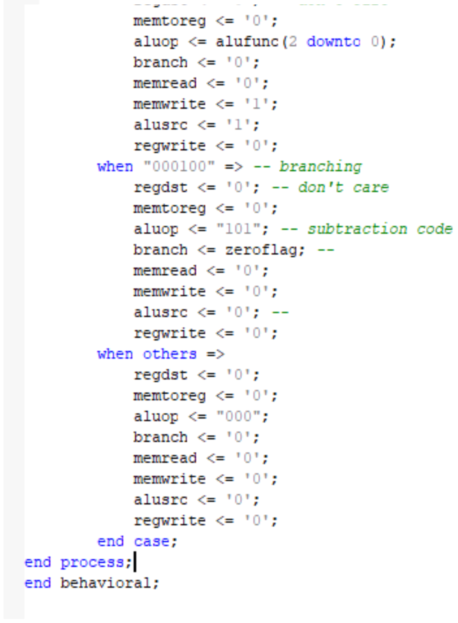


```
library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
    port(   opcode : in std_logic_vector(5 downto 0);
            alufunc : in std_logic_vector(5 downto 0);
            zeroflag : in std_logic;
            aluop : out std_logic_vector(2 downto 0);
            regdst, branch, memread, memtoreg, memwrite, alusrc, regwrite : out std_logic
        );
end control_unit;

architecture behavioral of control_unit is
begin
process(opcode, alufunc, zeroflag)
begin
    case opcode is
        when "000000" => -- r type
            regdst <= '1';
            memtoreg <= '0';
            aluop <= alufunc(2 downto 0);
            branch <= '0';
            memread <= '0';
            memwrite <= '0';
            alusrc <= '0';
            regwrite <= '1';
        when "000001" => -- lw
            regdst <= '0';
            memtoreg <= '1';
            aluop <= alufunc(2 downto 0);
            branch <= '0';
            memread <= '1';
            memwrite <= '0';
            alusrc <= '1';
            regwrite <= '1';
```

- Register file:

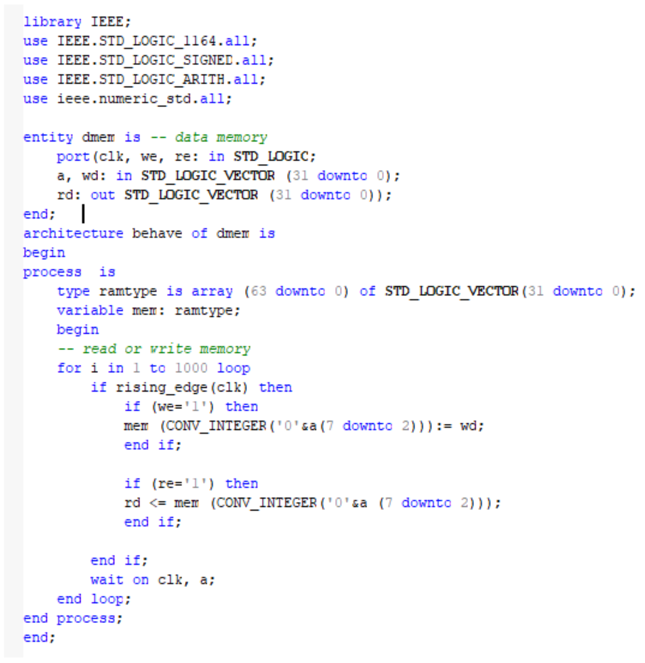


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RegisterFile is
    Port ( clk : in  STD_LOGIC;
           reg_write : in  STD_LOGIC;
           read_reg1, read_reg2, write_reg : in  STD_LOGIC_VECTOR(4 downto 0);
           write_data : in  STD_LOGIC_VECTOR(31 downto 0);
           read_data1, read_data2 : out  STD_LOGIC_VECTOR(31 downto 0));
end RegisterFile;

architecture Behavioral of RegisterFile is
    type reg_array is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0); -- 32 32-bit registers
    signal registers : reg_array := (
        "00000000000000000000000000000000", -- $zero
        "00000000110000010001110000100101", -- mem 1
        "11111001111111110000000001100010",
        "00000110000000001111100000001110",
        "00011100000011110000000000111000",
        "00000000000000000000000000001110",
        "00000000000000000000000000001110",
        "01010011110000000000000011011000",
        "00000000011000110000001110111100",
        "00110000000111110000011100000011",
        "00000001111000000000111000001110", -- mem 10
        "00001100001110001100001100111110",
        "11000111110000000111111001110000",
        "01100000011100011110001111100001",
        "11000011111111110000000001111111",
        "01111110000000011111101000111000",
        "00000011111110000000111110000111",
        "11001100110011001100110000110010",
        "00001101101110010111011101100001",
```

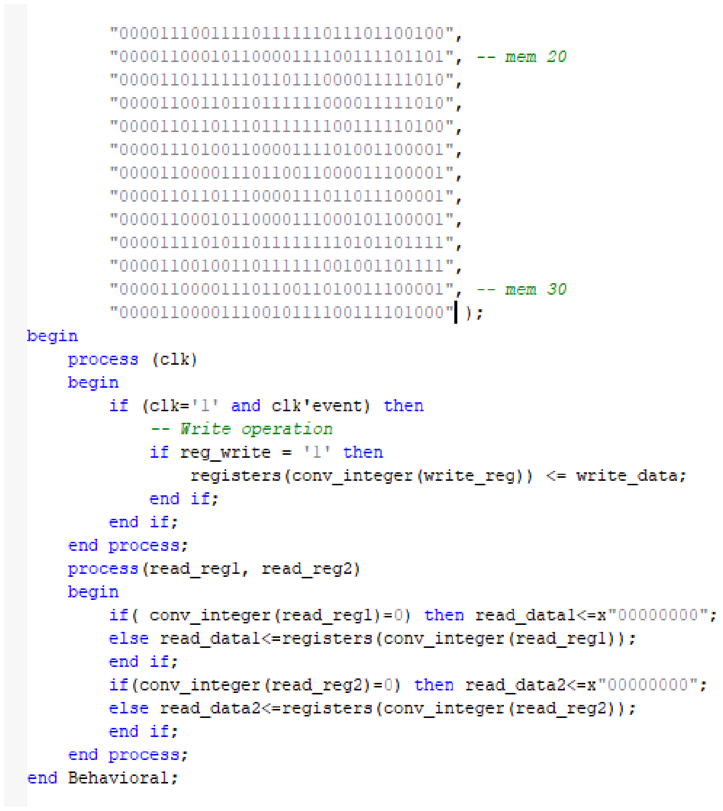


```
            memtoreg <= '0';
            aluop <= alufunc(2 downto 0);
            branch <= '0';
            memread <= '0';
            memwrite <= '1';
            alusrc <= '1';
            regwrite <= '0';
        when "000100" => -- branching
            regdst <= '0'; -- don't care
            memtoreg <= '0';
            aluop <= "101"; -- subtraction code
            branch <= zeroflag; --
            memread <= '0';
            memwrite <= '0';
            alusrc <= '0'; --
            regwrite <= '0';
        when others =>
            regdst <= '0';
            memtoreg <= '0';
            aluop <= "000";
            branch <= '0';
            memread <= '0';
            memwrite <= '0';
            alusrc <= '0';
            regwrite <= '0';
        end case;
end process;
end behavioral;
```

- Data memory:

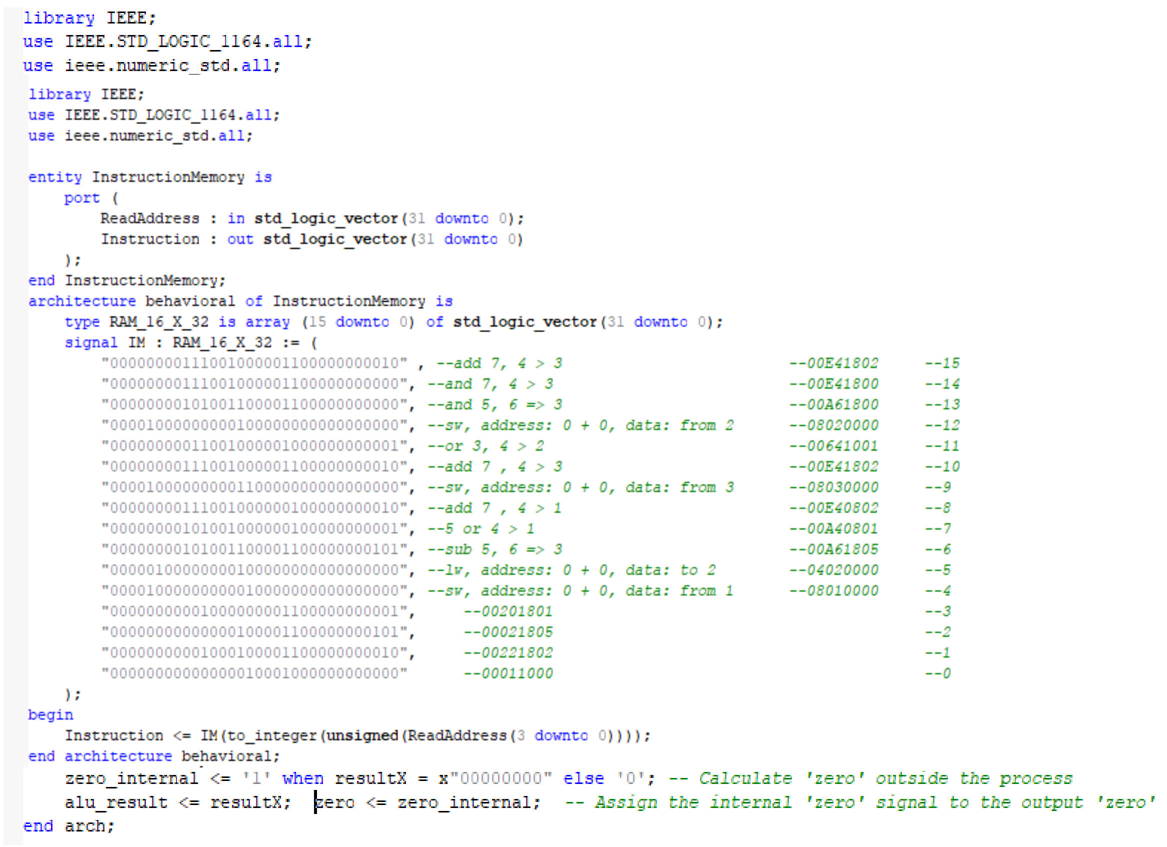


```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
use ieee.numeric_std.all;

entity dmem is -- data memory
    port(clk, we, re: in STD_LOGIC;
    a, wd: in STD_LOGIC_VECTOR (31 downto 0);
    rd: out STD_LOGIC_VECTOR (31 downto 0));
end;
architecture behave of dmem is
begin
process  is
    type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
    begin
    -- read or write memory
    for i in 1 to 1000 loop
        if rising_edge(clk) then
            if (we='1') then
            mem (CONV_INTEGER('0'&a(7 downto 2))):= wd;
            end if;

            if (re='1') then
            rd <= mem (CONV_INTEGER('0'&a (7 downto 2)));
            end if;

        end if;
        wait on clk, a;
    end loop;
end process;
end;
```



```
        "00001110011110111111011101100100",
        "00001100010110000111100111101101", -- mem 20
        "00001101111110110111000011111010",
        "00001100110110111111000011111010",
        "00001101101110111111100111110100",
        "00001110100110000111101001100001",
        "00001100001110110011000011100001",
        "00001101101110000111011011100001",
        "00001100010110000111000101100001",
        "00001111010110111111110101101111",
        "00001100100110111111001001101111",
        "00001100001110110011010011100001", -- mem 30
        "00001100001110010111100111101000" );
begin
    process (clk)
    begin
        if (clk='1' and clk'event) then
            -- Write operation
            if reg_write = '1' then
                registers(conv_integer(write_reg)) <= write_data;
            end if;
        end if;
    end process;
    process(read_reg1, read_reg2)
    begin
        if( conv_integer(read_reg1)=0) then read_data1<=x"00000000";
        else read_data1<=registers(conv_integer(read_reg1));
        end if;
        if(conv_integer(read_reg2)=0) then read_data2<=x"00000000";
        else read_data2<=registers(conv_integer(read_reg2));
        end if;
    end process;
end Behavioral;
```

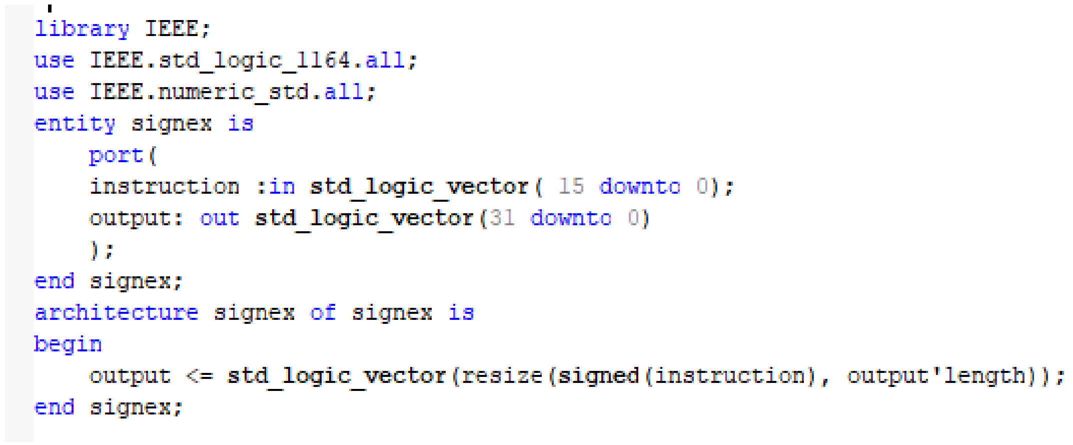
- ALU:



```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity InstructionMemory is
    port (
        ReadAddress : in std_logic_vector(31 downto 0);
        Instruction : out std_logic_vector(31 downto 0)
    );
end InstructionMemory;
architecture behavioral of InstructionMemory is
    type RAM_16_X_32 is array (15 downto 0) of std_logic_vector(31 downto 0);
    signal IM : RAM_16_X_32 := (
        "00000000111001000001100000000010" , --add 7, 4 > 3                        --00E41802     --15
        "00000000111001000001100000000000", --and 7, 4 > 3                         --00E41800     --14
        "00000000101001100001100000000000", --and 5, 6 => 3                        --00A61800     --13
        "00001000000000100000000000000000", --sw, address: 0 + 0, data: from 2    --08020000     --12
        "00000000011001000001000000000001", --or 3, 4 > 2                          --00641001     --11
        "00000000111001000001100000000010", --add 7 , 4 > 3                        --00E41802     --10
        "00001000000000110000000000000000", --sw, address: 0 + 0, data: from 3    --08030000     --9
        "00000000111001000000100000000010", --add 7 , 4 > 1                        --00E40802     --8
        "00000000101001000000100000000001", --5 or 4 > 1                           --00A40801     --7
        "00000000101001100001100000000101", --sub 5, 6 => 3                        --00A61805     --6
        "00001000000000100000000000000000", --lw, address: 0 + 0, data: to 2      --04020000     --5
        "00001000000000010000000000000000", --sw, address: 0 + 0, data: from 1    --08010000     --4
        "00000000001000000001100000000001",     --00201801                                      --3
        "00000000000000100001100000000101",     --00021805                                      --2
        "00000000001000100001100000000010",     --00221802                                      --1
        "00000000000000010001000000000000"      --00011000                                      --0
    );
begin
    Instruction <= IM(to_integer(unsigned(ReadAddress(3 downto 0))));
end architecture behavioral;
    zero_internal <= '1' when resultX = x"00000000" else '0'; -- Calculate 'zero' outside the process
    alu_result <= resultX;  zero <= zero_internal;  -- Assign the internal 'zero' signal to the output 'zero'
end arch;
```

- Instruction memory:

- Sign extend:



```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity signex is
    port(
    instruction :in std_logic_vector( 15 downto 0);
    output: out std_logic_vector(31 downto 0)
    );
end signex;
architecture signex of signex is
begin
    output <= std_logic_vector(resize(signed(instruction), output'length));
end signex;
```

- MUX:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Mux is
    Generic( N: integer :=32 );
    port(  Mux_in0:in std_logic_vector(N-1 downto 0);
           Mux_in1:in std_logic_vector(N-1 downto 0);
           Mux_ctl:in std_logic;
           Mux_out:out std_logic_vector(N-1 downto 0)
           );
end Mux;
architecture behavioral of Mux is
begin
    Mux_out<=Mux_in0 when Mux_ctl='0' else
        Mux_in1;
end behavioral;
```

- Adder:

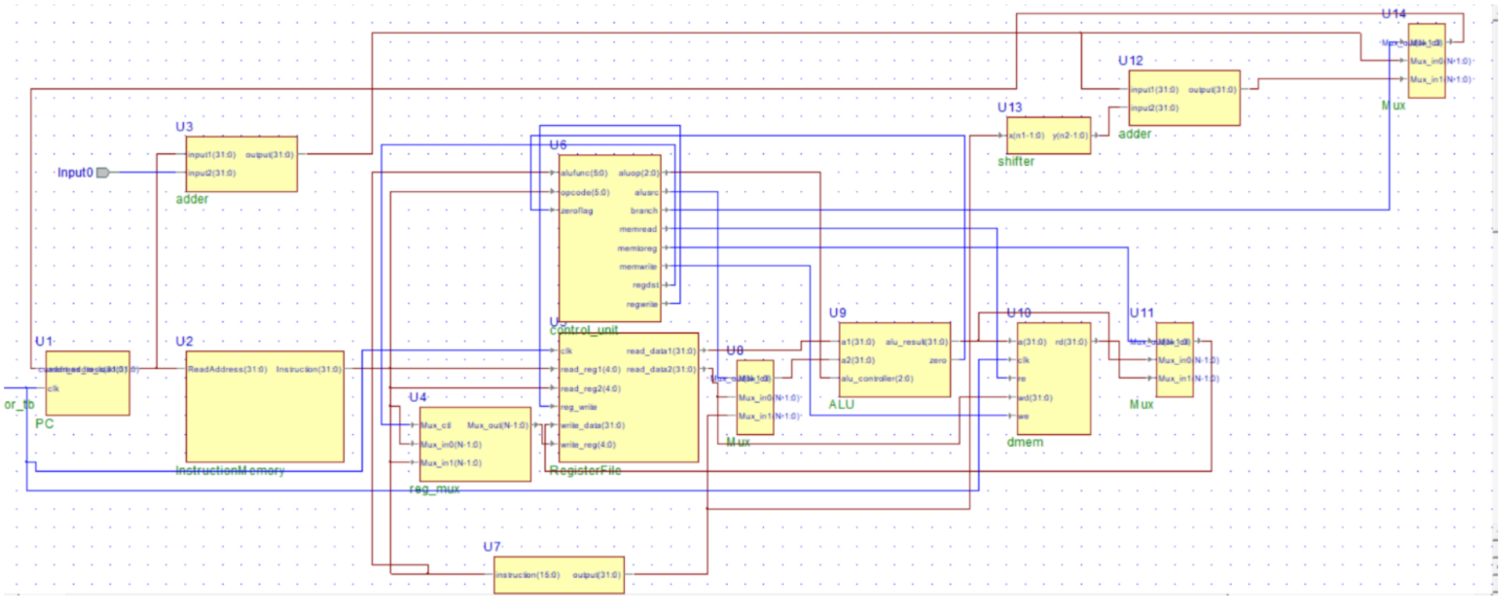
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity adder is
    port (
        input1,input2: in std_logic_vector(31 downto 0);
        output: out std_logic_vector(31 downto 0)
    );
end adder;
architecture adder of adder is
begin
     output<=input1+input2;
end adder;
```

- Shifter:

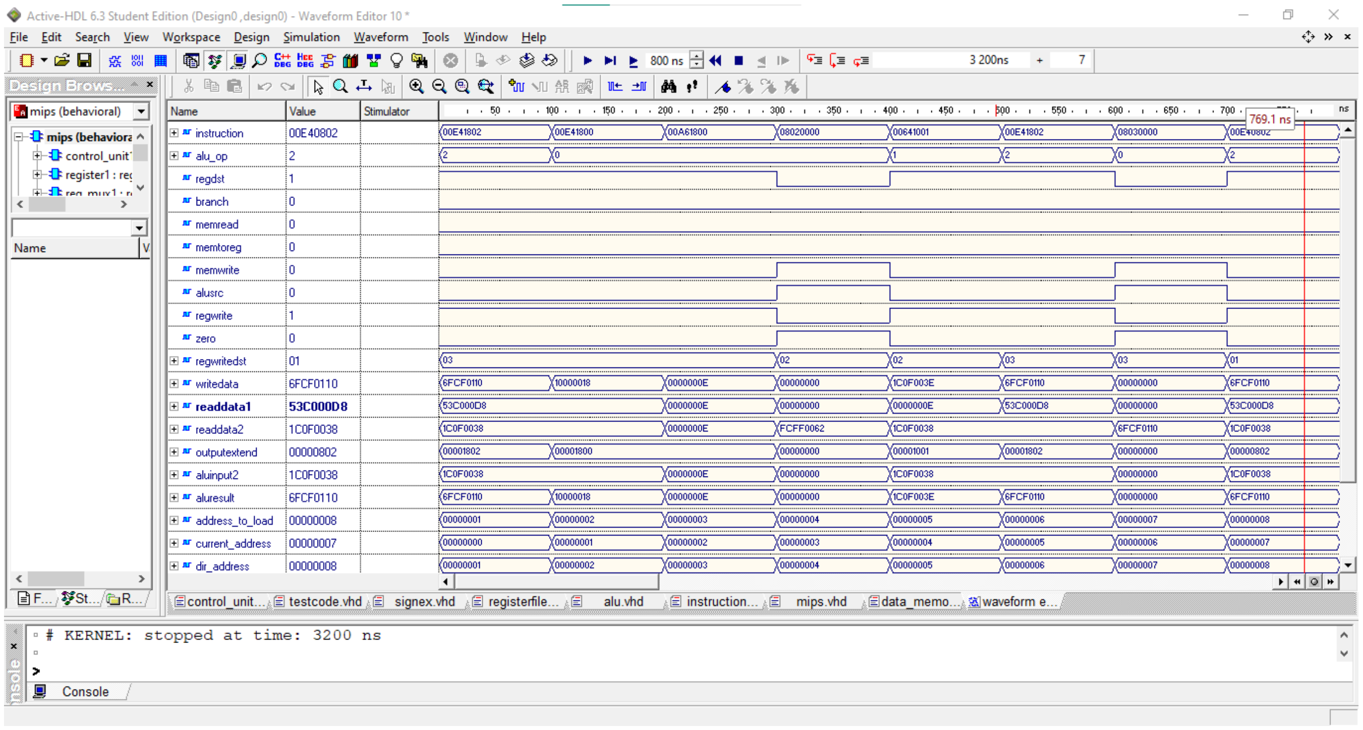
```
librarv IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Clock_Generator_tb is
end Clock_Generator_tb;

architecture Behavioral of Clock_Generator_tb is
    signal clk_out : STD_LOGIC;
    signal current_address_tb : STD_LOGIC_VECTOR (31 downto 0) := (others => '0'); -- Initialize current_address
    component mips is
        Port (
            clk_out : out STD_LOGIC
        );
    end component;
begin
    uut: mips
        Port map (
            clk_out => clk_out
        );
    tb_process : process
    begin
        wait for 100 ns;
        assert false report "Simulation Finished" severity note;
        wait;
    end process;
end Behavioral;
```

- Testbench:

3.1. SCHEMATIC DIAGRAM

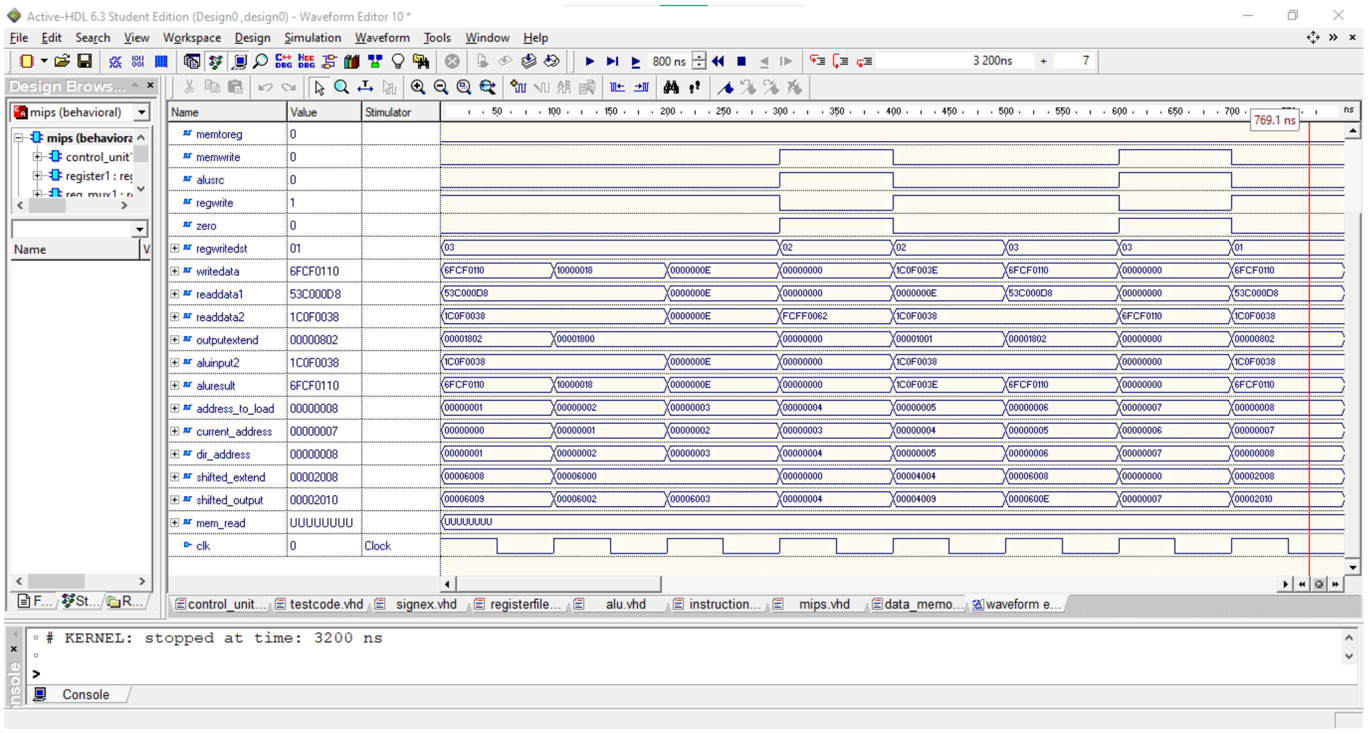


3.2. SIMULATION


Active-HDL 6.3 Student Edition (Design0 ,design0) - Waveform Editor 10 *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
800 ns
3 200ns
769.1 ns

Name	Value	Stimulator
instruction	00E40802	
alu_op	2	
regdst	1	
branch	0	
memread	0	
memtoreg	0	
memwrite	0	
alusrc	0	
regwrite	1	
zero	0	
regwritedst	01	
writedata	6FCF0110	
readdata1	53C000D8	
readdata2	1C0F0038	
outputextend	00000802	
aluinput2	1C0F0038	
aluresult	6FCF0110	
address_to_load	00000008	
current_address	00000007	
dir_address	00000008	

```
# KERNEL: stopped at time: 3200 ns
>
```


Active-HDL 6.3 Student Edition (Design0 ,design0) - Waveform Editor 10 *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
800 ns
3 200ns
769.1 ns

Name	Value	Stimulator
memtoreg	0	
memwrite	0	
alusrc	0	
regwrite	1	
zero	0	
regwritedst	01	
writedata	6FCF0110	
readdata1	53C000D8	
readdata2	1C0F0038	
outputextend	00000802	
aluinput2	1C0F0038	
aluresult	6FCF0110	
address_to_load	00000008	
current_address	00000007	
dir_address	00000008	
shifted_extend	00002008	
shifted_output	00002010	
mem_read	UUUUUUUU	
clk	0	Clock

```
# KERNEL: stopped at time: 3200 ns
>
```

4. REFERENCES

The following references were instrumental in guiding the research, design, and implementation of the MIPS ISA 32-bit processor using VHDL.

- Nelson, V.P., Carroll, B.D., Nagle, H.T., & Irwin, J.D. (2020). ***Digital Logic Circuit Analysis and Design (2nd ed.).*** Pearson Education, Inc.

- Brown, S., & Vranesic, Z. (2014). ***Fundamentals of Digital Logic with Verilog Design (3rd ed.).*** McGraw-Hill Education.

- Harris, D., & Harris, S. (2012). ***Digital Design and Computer Architecture (2nd ed.).*** Morgan Kaufmann.