DELHI TECHNOLOGICAL UNIVERSITY

SUBJECT- Digital Design II (EC204)

SUBMITTED To – Dr. Poornima Mittal

SUBMITTED BY- Abhay Lakhotra & Anshul

(2K19/EC/006) (2K19/EC/022)

DIGITAL DESIGN INNOVATIVE PROJECT REPORT

DESIGN OF 16 BIT MIPS PROCESSOR IN VHDL

ABSTRACT

Firstly, we will use the instruction set and architecture design for the MIPS processor provided. Based on the provided instruction set, we will design and implement the data path and control unit. After completing the design for the MIPS processor, we will write the code for the whole design of the MIPS processor. After that we will verify the code by doing simulations on ModelSim and create a testbench in order to verify the functional correctness of HDL model.

INTRODUCTION

MIPS(***Microprocessor without Interlocked Pipelined Stages***) is a popular *reduced instruction set architecture (RISC) processor*. It is a 16-bit processor (it can operate on 16 bits of data at a time). The MIPS processor structured depends on the RISC processor. It has 16 registers. The processor utilizes the 5 pipeline stages, which are Instruction Fetch (IF), Instruction Decoder (ID), Execute (EX), Memory Access (MEM) and Write Back (WB). The pipelined MIPS processor plays out every one of the stages in various clock cycles.

RISC Instruction Set will consist of fewer number of instructions, simpler instructions and large number of registers mostly general purpose registers. The result is that it's much easier to implement these kind of processors in hardware.

Instruction execution cycle is divide into 5 steps :

1. **IF** : Instruction Fetch
2. **ID** : Instruction Decode / Register Fetch
3. **EX** : Execution/ Effective Address Calculation

4. **MEM** : Memory Access/ Branch Completion
5. **WB** : Register Write-back

The ***Instruction Fetch stage*** is where a program counter will pull the next instruction from the correct location in program memory. In addition the program counter was updated with either the next instruction location sequentially, or the instruction location as determined by a branch.

- Here the instruction pointed to by *PC* is fetched from memory, and also the next value of *PC* is computed.
- Every MIPS instruction is of 16 bits.
- Every memory word is of 16 bits and has a unique address.
- For a branch instruction, new value of PC may be the target address. So, PC is not updated in this stage; new value is stored in a register *NPC*.

IF :- IR <- Mem [PC];

NPC <- PC + 1;

The ***Instruction Decode stage*** is where the control unit determines what values the control lines must be set to depending on the instruction. In addition, hazard detection is implemented in this stage, and all necessary values are fetched from the register banks.

- The instruction already fetched in *IR* is decoded.
- Decoding is done in parallel with reading the register operands rs and rt.
- In a similar way, the immediate data are sign-extended.

ID :- A <- Reg[rs];

B <- Reg[rt];

Here, A and B are temporary registers which will be required later.

The ***Execute stage*** is where the instruction is actually sent to the ALU and executed. If necessary, branch locations are calculated in this stage as well. Additionally, this is the stage where the forwarding unit will determine whether the output of the ALU or the memory unit should be forwarded to the ALU's inputs.

The **Memory Access stage** is where, if necessary, system memory is accessed for data. Also, if a write to data memory is required by the instruction it is done in this stage. In order to avoid additional complications it is assumed that a single read or write is accomplished within a single CPU clock cycle.

Finally, the ***Write Back stage*** is where any calculated values are written back to their proper registers. The write back to the register bank occurs during the first half of the cycle in order to avoid structural and data hazards if this was not the case.

IMPLEMENTATION

- The instruction fetch stage has multiple responsibilities in that it must properly update the CPU's program counter in the normal case as well as the branch instruction case. The instruction fetch stage is also responsible for reading the instruction memory and sending the current instruction to the next stage in the pipeline, or a stall if a branch has been detected in order to avoid incorrect execution. The instruction fetch stage is composed of three components: instruction memory, program counter, and the instruction address adder. The instruction memory also takes inputs from the outside world that allow the loading of instruction memory for later execution.

- The instruction memory unit was designed to model a small amount of cache and therefore was made to be accessed within a single CPU cycle. The instruction memory was sized at 1k bits and could therefore at maximum contain 32 separate instructions.

- The Decode Stage is the stage of the CPU's pipeline where the fetched instruction is decoded, and values are fetched from the register bank. It is responsible for mapping the different sections of the instruction into their proper representations (based on R or I type instructions). The Decode stage consists of the Control unit, the Hazard Detection Unit, the Sign Extender, and the Register bank, and is responsible for connecting all of these components together. It splits the instruction into its various parts and feeds

them to the corresponding components. Regisers Rs and Rt are fed to the register bank, the immediate section is fed to the sign extender, and the ALU opcode and function codes are sent to the control unit. The outputs of these corresponding components are then clocked and stored for the next stage.

- The execute stage is responsible for taking the data and actually performing the specified operation on it. The execute stage consists of an ALU, a Determine Branch unit, and a Forwarding Unit. The execute stage connects these components together so that the ALU will process the data properly, given inputs chosen by the forwarding unit, and will notify the decode stage if a branch is indeed to be taken.

- The ALU is responsible for performing the actual calculations specified by the instruction. It takes two 16-bit inputs and some control signals, and gives a single 16-bit output along with some information about the output – whether it is zero or negative. The forwarding unit is responsible for choosing what input is to be fed into the ALU. It takes the input from the decode stage, the value that the alu has fed to the Memory stage, and the value that the Alu has fed to the write back stage, as well as the register numbers corresponding to all of these, and determines if any conflicts exist. It will choose which of these values must be sent to the ALU.

- The Memory stage is responsible for taking the output of the alu and forwarding it to the proper memory location if the instruction is a store. The memory stage contains one component: the data_memory object. It connects the data memory to a register bank for the write back stage to read, and also forwards on information about the current write back register. This register's number and calculated value are fed back to the forwarding unit in the execute stage to allow it to determine which value to pass to the ALU.

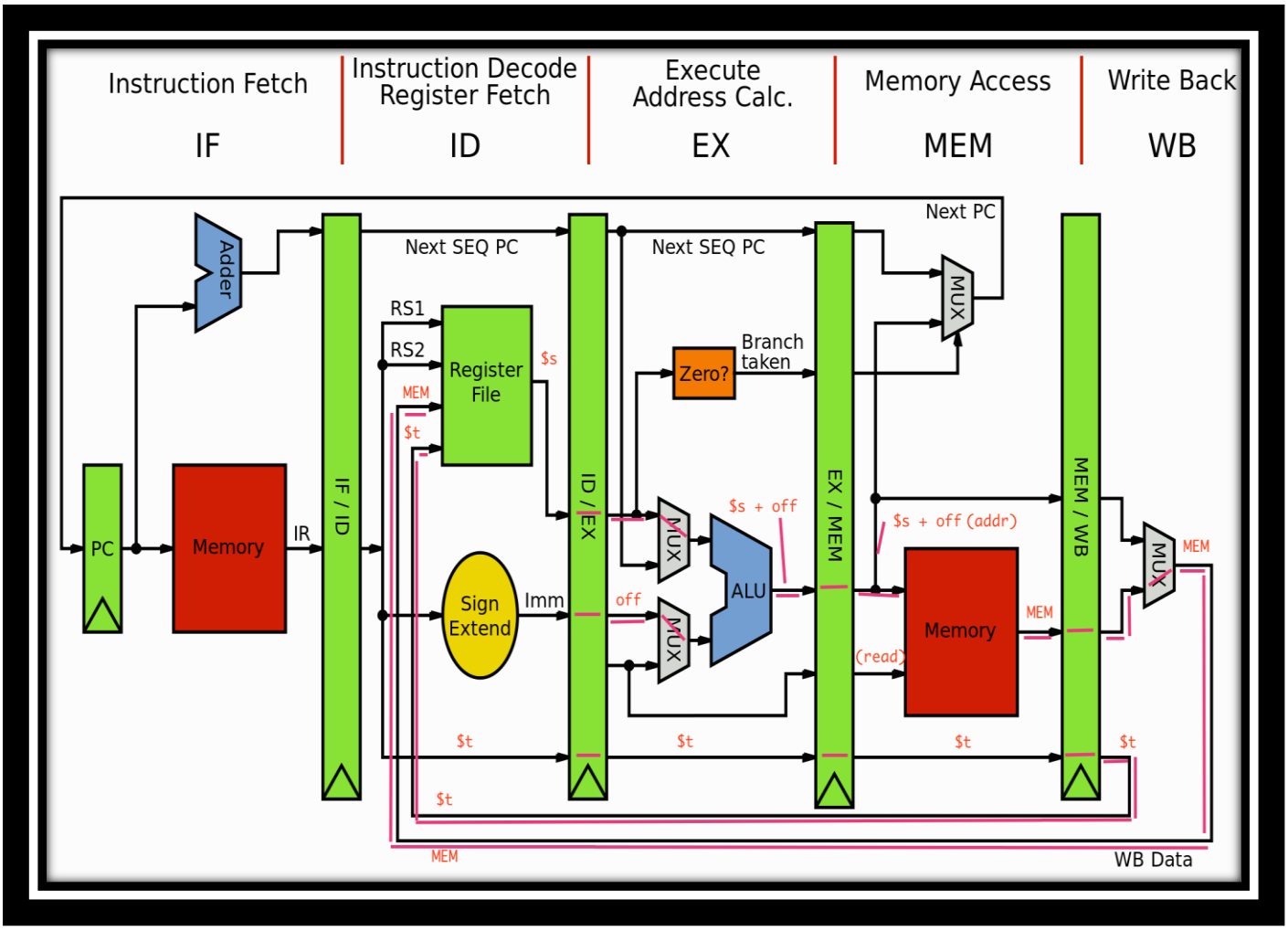
- The writeback stage is responsible for writing the calculated value back to the proper register. It has input control lines that tell it whether this instruction writes back or not, and whether it writes back ALU output or Data memory

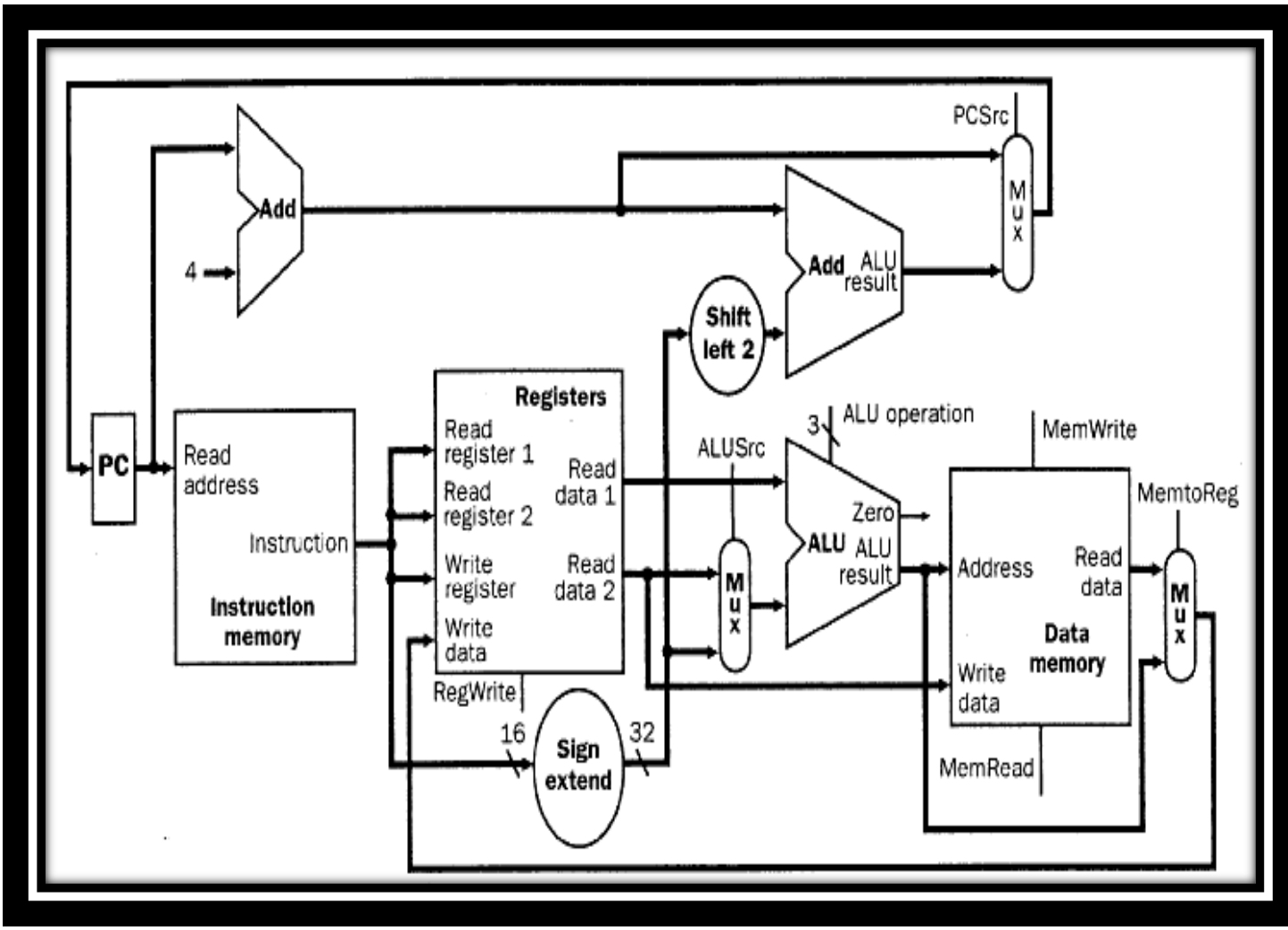
output. It then chooses one of these outputs and feeds it to the register bank based on these control lines.

For Example, ***ADD R1, R2 & R3***

PC contains the address of the instruction , it is fetched. So, the instruction is loaded in IR. Two operands are prefetched A & B (two source operands). For example, A &B selected, they are added. We store the result in ALUOut. For memory, it does nothing , ALUOut is simply forwarded and for WB this ALUOut will be written back to the register and this IR will contain which register number. This is the simple datapath we have shown.

DATAPATH


Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
MUX
RS1
RS2
Register
File
$s
MEM
$t
Zero?
Branch
taken
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
$s + off
$s + off (addr)
MUX
ALU
Sign
Extend
Imm
off
MUX
MUX
MEM
Memory
(read)
MEM
$t
$t
$t
$t
$t
MEM
WB Data



CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;
 entity MIPS_VHDL is
port (
clk,reset: in std_logic;
pc_out, alu_result: out std_logic_vector(15 downto 0)
```

```
);
end MIPS_VHDL;
 architecture Behavioral of MIPS_VHDL is
signal pc_current: std_logic_vector(15 downto 0);
signal pc_next,pc2: std_logic_vector(15 downto 0);
signal instr: std_logic_vector(15 downto 0);
signal reg_dst,mem_to_reg,alu_op: std_logic_vector(1 downto 0);
signal jump,branch,mem_read,mem_write,alu_src,reg_write: std_logic;
signal reg_write_dest: std_logic_vector(2 downto 0);
signal reg_write_data: std_logic_vector(15 downto 0);
signal reg_read_addr_1: std_logic_vector(2 downto 0);
signal reg_read_data_1: std_logic_vector(15 downto 0);
signal reg_read_addr_2: std_logic_vector(2 downto 0);
signal reg_read_data_2: std_logic_vector(15 downto 0);
signal sign_ext_im,read_data2,zero_ext_im,imm_ext: std_logic_vector(15 downto 0);
signal JRControl: std_logic;
signal ALU_Control: std_logic_vector(2 downto 0);
signal ALU_out: std_logic_vector(15 downto 0);
signal zero_flag: std_logic;
signal im_shift_1, PC_j, PC_beq, PC_4beq,PC_4beqj,PC_jr: std_logic_vector(15 downto 0);
signal beq_control: std_logic;
signal jump_shift_1: std_logic_vector(14 downto 0);
signal mem_read_data: std_logic_vector(15 downto 0);
signal no_sign_ext: std_logic_vector(15 downto 0);
signal sign_or_zero: std_logic;
signal tmp1: std_logic_vector(8 downto 0);
begin
```

```
-- PC of the MIPS Processor
process(clk,reset)
begin
if(reset='1') then
  pc_current <= x"0000";
elsif(rising_edge(clk)) then
  pc_current <= pc_next;
end if;
end process;
-- PC + 2
  pc2 <= pc_current + x"0002";

-- Instruction memory
Instruction_Memory: entity work.Instruction_Memory_VHDL
        port map
       (
        pc=> pc_current,
        instruction => instr
        );
-- jump shift left 1
jump_shift_1 <= instr(13 downto 0) & '0';

-- control unit
control: entity work.control_unit_VHDL
   port map
   (reset => reset,
    opcode => instr(15 downto 13),
    reg_dst => reg_dst,
```

```
    mem_to_reg => mem_to_reg,
    alu_op => alu_op,
    jump => jump,
    branch => branch,
    mem_read => mem_read,
    mem_write => mem_write,
    alu_src => alu_src,
    reg_write => reg_write,
    sign_or_zero => sign_or_zero
    );

-- multiplexer regdest
  reg_write_dest <= "111" when  reg_dst= "10"
  else instr(6 downto 4) when  reg_dst= "01"
  else instr(9 downto 7);

-- register file
reg_read_addr_1 <= instr(12 downto 10);
reg_read_addr_2 <= instr(9 downto 7);
register_file: entity work.register_file_VHDL
port map
(
clk => clk,
rst => reset,
reg_write_en => reg_write,
reg_write_dest => reg_write_dest,
reg_write_data => reg_write_data,
reg_read_addr_1 => reg_read_addr_1,
```

```
reg_read_data_1 => reg_read_data_1,
reg_read_addr_2 => reg_read_addr_2,
reg_read_data_2 => reg_read_data_2
);

-- sign extend
tmp1 <= (others => instr(6));
sign_ext_im <=  tmp1 & instr(6 downto 0);
zero_ext_im <= "000000000"& instr(6 downto 0);
imm_ext <= sign_ext_im when sign_or_zero='1' else zero_ext_im;

-- control unit
JRControl <= '1' when ((alu_op="00") and (instr(3 downto 0)="1000")) else '0';

-- ALU control unit
ALUControl: entity work.ALU_Control_VHDL port map
  (
   ALUOp => alu_op,
   ALU_Funct => instr(2 downto 0),
   ALU_Control => ALU_Control
   );

-- multiplexer
read_data2 <= imm_ext when alu_src='1' else reg_read_data_2;

-- ALU unit
alu: entity work.ALU_VHDL port map
  (
```

```
   a => reg_read_data_1,
   b => read_data2,
   alu_control => ALU_Control,
   alu_result => ALU_out,
   zero => zero_flag
   );
-- immediate shift 1
im_shift_1 <= imm_ext(14 downto 0) & '0';
no_sign_ext <= (not im_shift_1) + x"0001";
-- PC beq add
PC_beq <= (pc2 - no_sign_ext) when im_shift_1(15) = '1' else (pc2 +im_shift_1);
-- beq control
   beq_control <= branch and zero_flag;
-- PC_beq
   PC_4beq <= PC_beq when beq_control='1' else pc2;
-- PC_j
PC_j <= pc2(15) & jump_shift_1;
-- PC_4beqj
PC_4beqj <= PC_j when jump = '1' else  PC_4beq;
-- PC_jr
PC_jr <= reg_read_data_1;
-- PC_next
pc_next <= PC_jr when (JRControl='1') else PC_4beqj;

-- data memory
data_memory: entity work.Data_Memory_VHDL port map
  (
  clk => clk,
```

```
  mem_access_addr => ALU_out,
  mem_write_data => reg_read_data_2,
  mem_write_en => mem_write,
  mem_read => mem_read,
  mem_read_data => mem_read_data
  );

-- write back
reg_write_data <= pc2 when (mem_to_reg = "10")
else mem_read_data when (mem_to_reg = "01")
else ALU_out;

-- output
pc_out <= pc_current;
alu_result <= ALU_out;

end Behavioral;
```

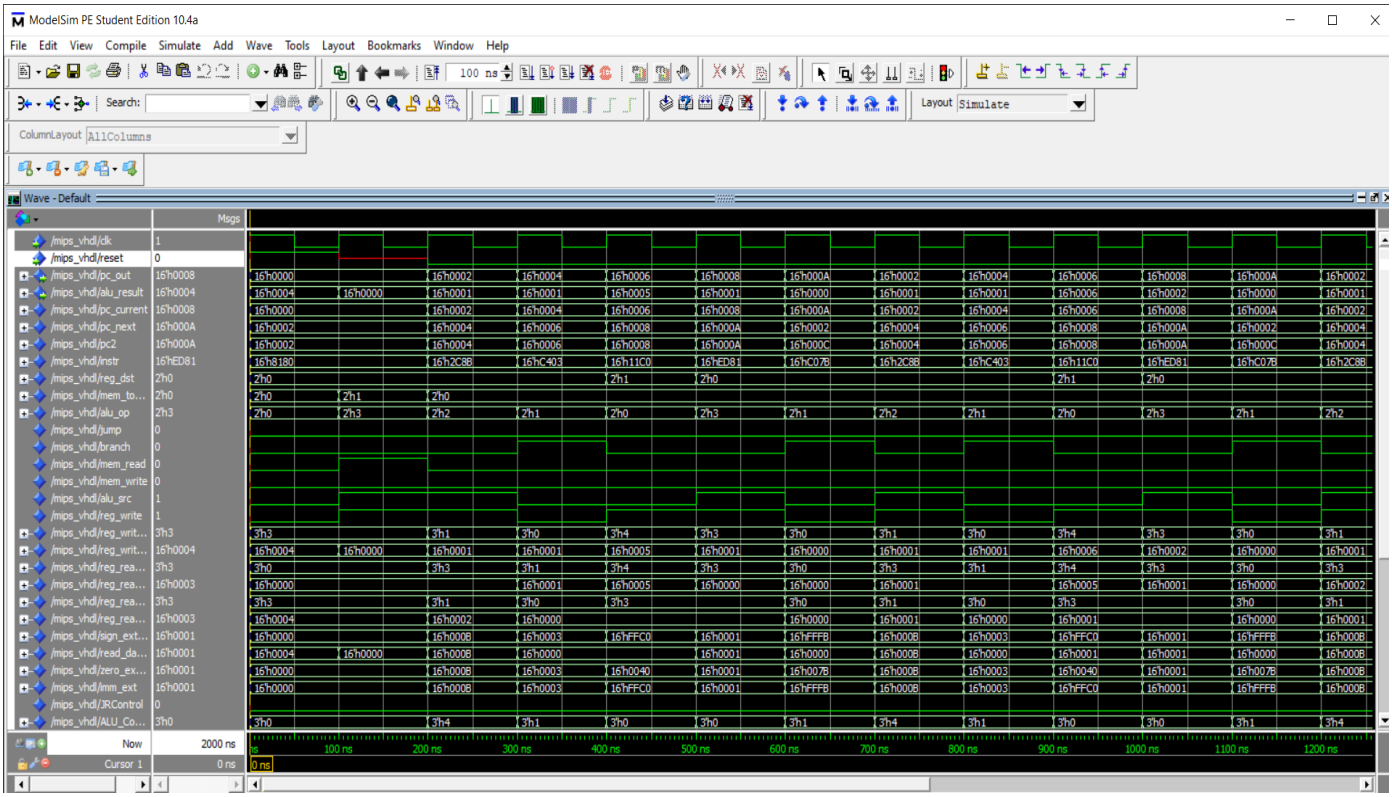
TESTBENCH CODE

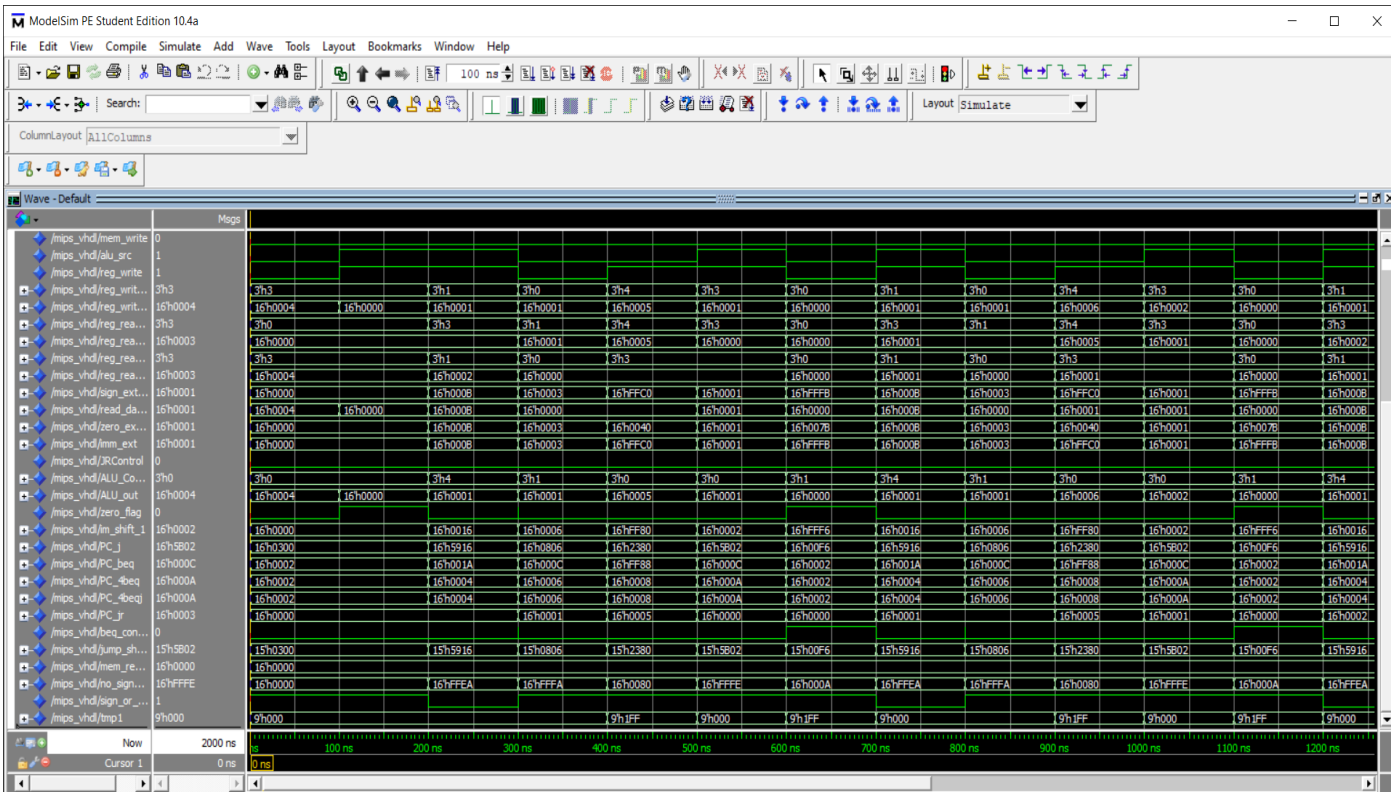
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_MIPS_VHDL IS
END tb_MIPS_VHDL;

ARCHITECTURE behavior OF tb_MIPS_VHDL IS
    -- Component Declaration for the single-cycle MIPS Processor in VHDL
    COMPONENT MIPS_VHDL
```

```
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         pc_out : OUT  std_logic_vector(15 downto 0);
         alu_result : OUT  std_logic_vector(15 downto 0)
        );
    END COMPONENT;
   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   --Outputs
   signal pc_out : std_logic_vector(15 downto 0);
   signal alu_result : std_logic_vector(15 downto 0);
   -- Clock period definitions
   constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the for the single-cycle MIPS Processor in VHDL
   uut: MIPS_VHDL PORT MAP (
          clk => clk,
          reset => reset,
          pc_out => pc_out,
          alu_result => alu_result
        );

   -- Clock process definitions
   clk_process :process
   begin
  clk <= '0';
```

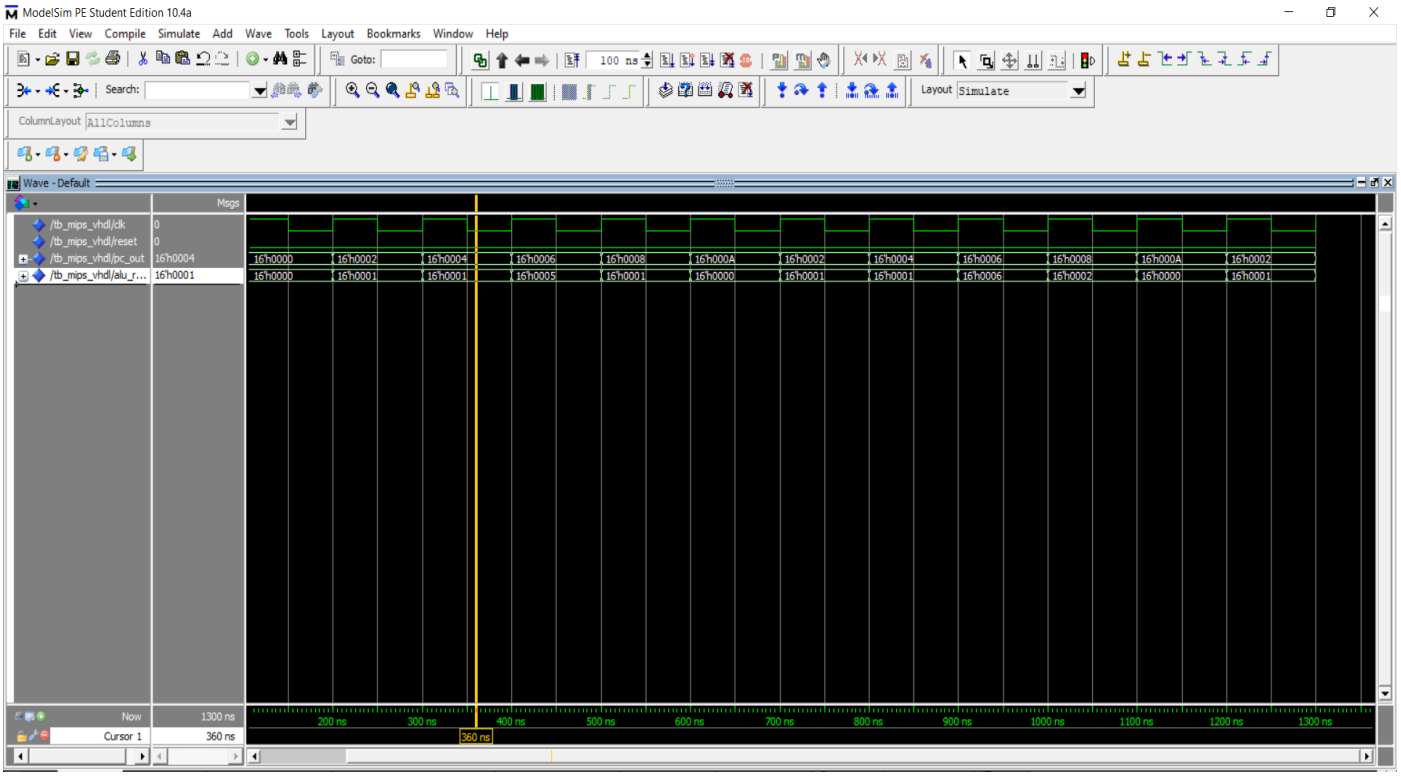
```
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
   end process;
   -- Stimulus process
   stim_proc: process
   begin
      reset <= '1';
      wait for clk_period*10;
  reset <= '0';
      -- insert stimulus here
      wait;
   end process;

END;
```

OUTPUTS


ModelSim PE Student Edition 10.4a
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/mips_vhdl/clk
/mips_vhdl/reset
/mips_vhdl/pc_out
/mips_vhdl/alu_result
/mips_vhdl/pc_current
/mips_vhdl/pc_next
/mips_vhdl/pc2
/mips_vhdl/instr
/mips_vhdl/reg_dst
/mips_vhdl/mem_to...
/mips_vhdl/alu_op
/mips_vhdl/jump
/mips_vhdl/branch
/mips_vhdl/mem_read
/mips_vhdl/mem_write
/mips_vhdl/alu_src
/mips_vhdl/reg_write
/mips_vhdl/reg_writ...
/mips_vhdl/reg_writ...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/sign_ext...
/mips_vhdl/read_da...
/mips_vhdl/zero_ex...
/mips_vhdl/mm_ext
/mips_vhdl/JRControl
/mips_vhdl/ALU_Co...
Now
Cursor 1


ModelSim PE Student Edition 10.4a
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/mips_vhdl/mem_write
/mips_vhdl/alu_src
/mips_vhdl/reg_write
/mips_vhdl/reg_writ...
/mips_vhdl/reg_writ...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/reg_rea...
/mips_vhdl/sign_ext...
/mips_vhdl/read_da...
/mips_vhdl/zero_ex...
/mips_vhdl/mm_ext
/mips_vhdl/JRControl
/mips_vhdl/ALU_Co...
/mips_vhdl/ALU_out
/mips_vhdl/zero_flag
/mips_vhdl/im_shift_1
/mips_vhdl/PC_j
/mips_vhdl/PC_beq
/mips_vhdl/PC_4beq
/mips_vhdl/PC_4beq
/mips_vhdl/PC_jr
/mips_vhdl/beq_con...
/mips_vhdl/jump_sh...
/mips_vhdl/mem_re...
/mips_vhdl/no_sign...
/mips_vhdl/sign_or_...
/mips_vhdl/tmp1
Now
Cursor 1

TESTBENCH OUTPUT



CONCLUSION

From the above output waveforms, we can see that the outputs (pc_out and alu_result) are showing same waveforms in both the testbench and code output. Hence, it verifies the design of 16 bit MIPS processor using behavioral modeling in VHDL.

FUTURE EXTENSION

We can modify the instruction memory to simulate all the instructions in the instruction set architecture, and then check simulation waveform and memory to see if the processor works correctly as designed.

REFERENCES

1. https://www.rroij.com/open-access/design-of-bit-risc-cpu-based-on-mips-20-24.pdf
2. https://www.researchgate.net/publication/312581426_VHDL_Design_and_Simulation_of_a_32_Bit_MIPS_RISC_Processor
3. https://www.youtube.com/watch?v=NCrlyaXMAn8&list=PLJ5C_6qdAvBELELTSPgzYkQg3HgclQh-5
4. https://www.youtube.com/watch?v=8xS1D8xYUoE&t=202s
5. https://en.wikipedia.org/wiki/MIPS_architecture