Department of Space, Government of India

Indian Institute of Space Science and Technology

Deemed to be University under Section 3 of the UGC Act, 1956

Thiruvananthapuram

Computer Organisation and OS

Project Report

DEPARTMENT OF AVIONICS

2014

Amal Krishna R
SC12B075
Bharat Devasani
SC12B078

Introduction

The VHDL model for the processor is as organised as shown in the figure. The instruction memory, data memory and register file are created as components with their architecture and entity descriptions. The main code, the MIPS entity embeds the control sequencing the instructions through the various stages of its operation. For simplicity we combined the instruction and data memory units to be a single memory and illustrate the use of the address and data buses. Later, when we use a test bench, we allow the test bench to directly write into the instruction memory in order to deposit instructions to be tested. The design is implemented on an FPGA using the LCD module.

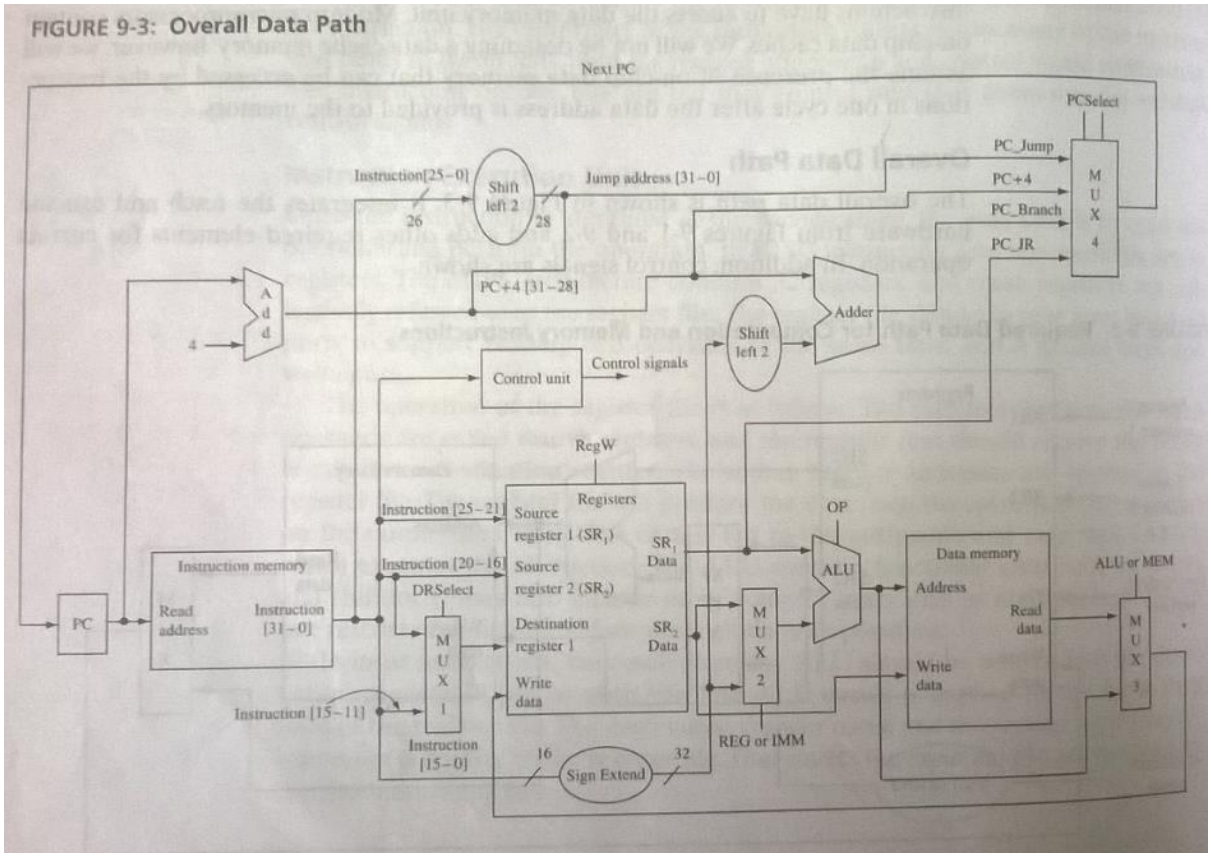
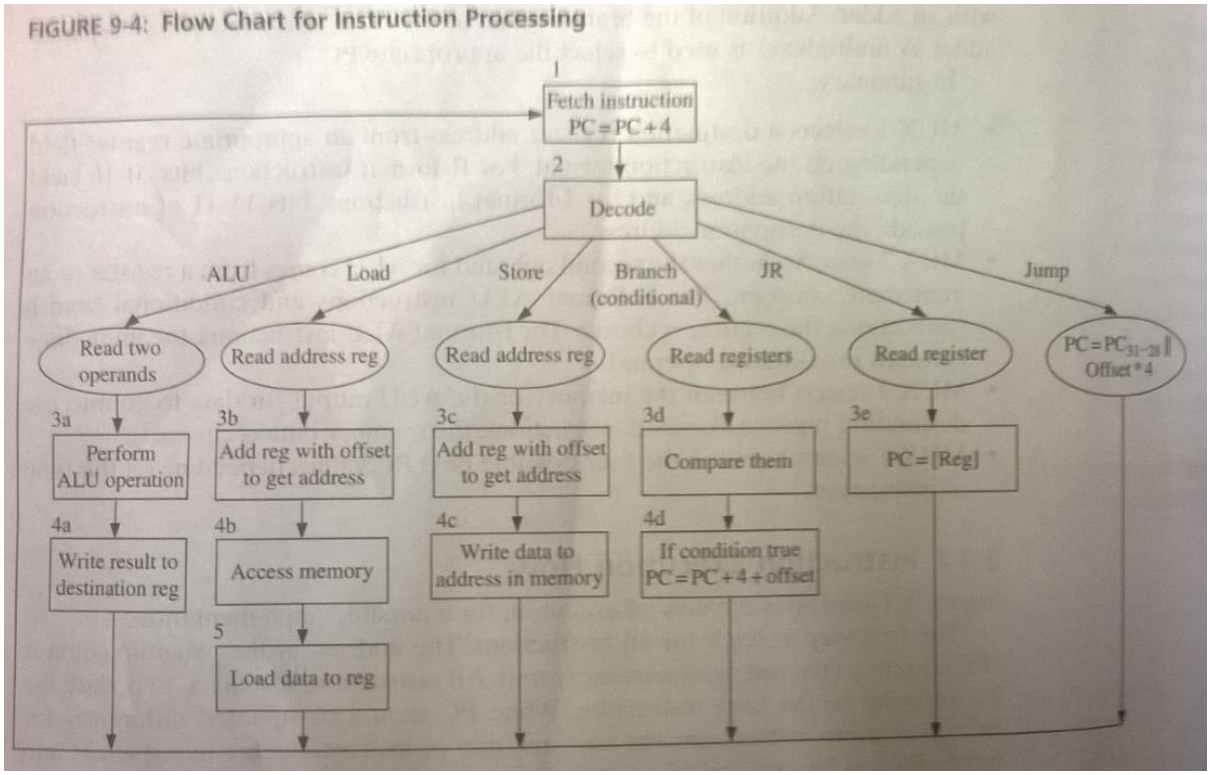


FIGURE 9-4: Flow Chart for Instruction Processing



VHDL Model for the Register File

The REG entity is used to represent the 32 MIPS registers. Each register is 32 bit long. The designation register address is DR, and the source register addresses are SR1 and SR2. Since there are 32 registers, DR, SR1 and SR2 are 5 bits each. The outputs ReadReg1 and ReadReg2 are the contents of the registers specified by SR1 and SR2. ReadReg1 is fed straight to the ALU. ReadReg2 can be used as a second ALU input, or as the input to data memory in the case of store instructions. The control signal RegW is used to control the write operation to the register file. If Regw is true, the data on lines Reg_In is written into the register pointed to by DR. Asynchronous reads is used to allow generation of distributed RAM for the register file.

VHDL Model for memory

The VHDL Model is similar to the SRAM model which has tri state input-output lines and allows easy testing with a test bench, where the test bench can write instructions into memory and the processor can drive the data bus of the memory. Although illustrated separate instruction and data memories, for continence and for illustrating the use of address and data buses, we have used a unified memory module which stores both instructions and data. The memory consists of 128 locations, each 32 bits wide, but we only use the seven lower bits since we implement only a small memory.

The address bus will be driven by the processor approximately for instruction and data access. The address input may come from the ALU that computes the address to access the data portion of the memory. The chip select (CS) and write enable (WE) signals allow the processor to control the reads and writes. When CS and WE are true, the data on Mem_Bus written to the memory location pointed to by address ADDR.

For simplicity, **the address is shown as a word in the VHDL code for the memory**. Hence, branch and jump offsets are used without multiplying by 4. In the actual MIPS processor, the memory is byte-addressable. Therefore, each instruction memory access should obtain the data found in the specified location concatenated with the next three memory locations. For example, if address = 0, the instruction register must be loaded with the contents of MEM[0], MEM[1]. MEM[2], and MEM[3]. The instructions are stored depending on the endianness of the machine. Many modern microprocessors support both big-endian and little-endian approaches.

VHDL Code for the Processor CPU

The register module that was created in the earlier section is used here. The VHDL model generally follows the flow of, implementing the fetcsh, decode, and excecute phases of an instruction. In order to increase the readability of the code, several aliases are defined. The most significant 6 bits of the instruction are denoted by the alias Opcode. The lowest 6 bits of the instruction aare denoted with the alia F_Code. The shift amount in shift instructions is denoted using NumShift, The two register source fields are aliased to SR1 and SR2. The following statements accomplish the aliasing

```
alias opcode: unsigned(5 downto 0) is Instr(31 downto 26);
alias SR1: unsigned(4 downto 0) is Instr(25 downto 21);
alias SR2: unsigned(4 downto 0) is Instr(20 downto 16);
alias F_Code: unsigned(5 downto 0) is Instr(5 downto 0);
alias NumShift: unsigned(4 downto 0) is Instr(10 downto 6);
alias ImmField: unsigned (15 downto 0) is Instr(15 downto 0);
```

For readability of the code, we also used constant declarations to associate the various opcodes. For example,

```
constant lw:   unsigned(5 downto 0) := "100011";  -- 35
constant sw:   unsigned(5 downto 0) := "101011";  -- 43
```

MIPS Processor Model Signals:

Clk	Input	Clock
Rst	Input	Synchronous reset
CS	Output	Memory chip select
WE	Output	Memory write enable
Addr	Output	Memory address
Mem_Bus	In/Out	Tristate memory bus
Op		ALU Operation select
Format		Indicates whether R,I or J format
Instr		The current instruction
Imm_Ext		Sign-extended immediate constant
PC		Current Program counter
NPC		Next Program counter
ReadReg1		Contents of first source register
ReadReg2		Contents of second source register
Reg_In		Data input to registers
ALU_InA		First operand
ALU_InB		Second Operand
ALU_Result		Output for ALU
ALUor MEM		Select signal for the Reg_In multiplier
REGorIMM		Select signal for the ALU_InB multiplier
RegW		Indicates if the destination register should be written to
FetchDorI		Select signal for the Address multiplier
Writing		Control signal for the MIPS processor output to the memory
DR		Address of destination register
State		Current state
NState		Next state

Two processes are used in the code. Since we have used separate clock cycles for the fetch operation. Decode operation, execute operation, and so on, it is necessary to save signals created during each stage for later use. The statements such as

```
OpSave <= Op;
REGorIMM_Save <= REGorIMM;
ALUorMEM_Save <= ALUorMEM;
```

Are used in the clocked process (the second process) for saving (explicit latching) of the relevant signals.

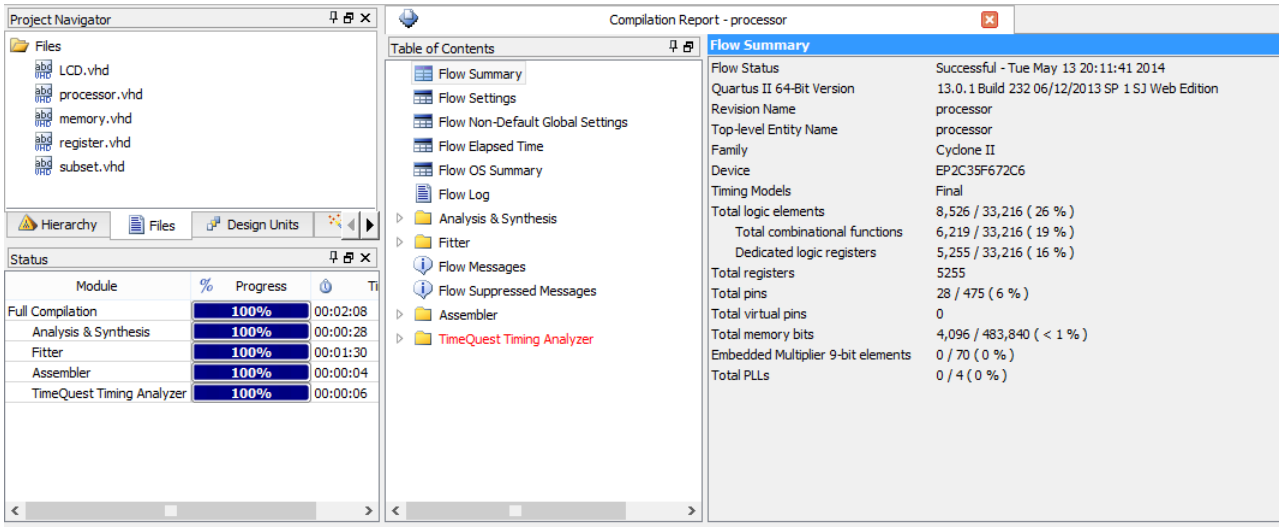
The multiplier at the input of the program counter is not explicitly coded. The various data transfers are coded behaviourally in the various states. A good synthesizer will be able to generate the multiplier to accomplish the various data transfers. Similarly, the multiplier to select the destination register address is also not explicitly coded. If the synthesis tool generates inefficient hardware for this multiplexed data transfer, we can code the multiplexer into the data path and generate control signals.

Complete MIPS

The processor module and the memory are integrated to yield the complete MIPS model. Component descriptions are created for the processor and the memory units. These components are integrated by using port-map statements. The high level entity is called Complete_MIPS. We haave also brought out the address and data buses as outputs from the high-level entity. If no outputs are shown in an entity, when the code is synthesized, it results in empty blocks. Depending on the synthesis tool, unused signals (and corresponding nets) may be deleted from the synthesized circuit.

LCD Module

The LCD module is used to display the Register value, Program counter and Memory as outputs on the LCD screen and Clock and Reset is given to push button's and Register address and address enable is given from toggle switches.

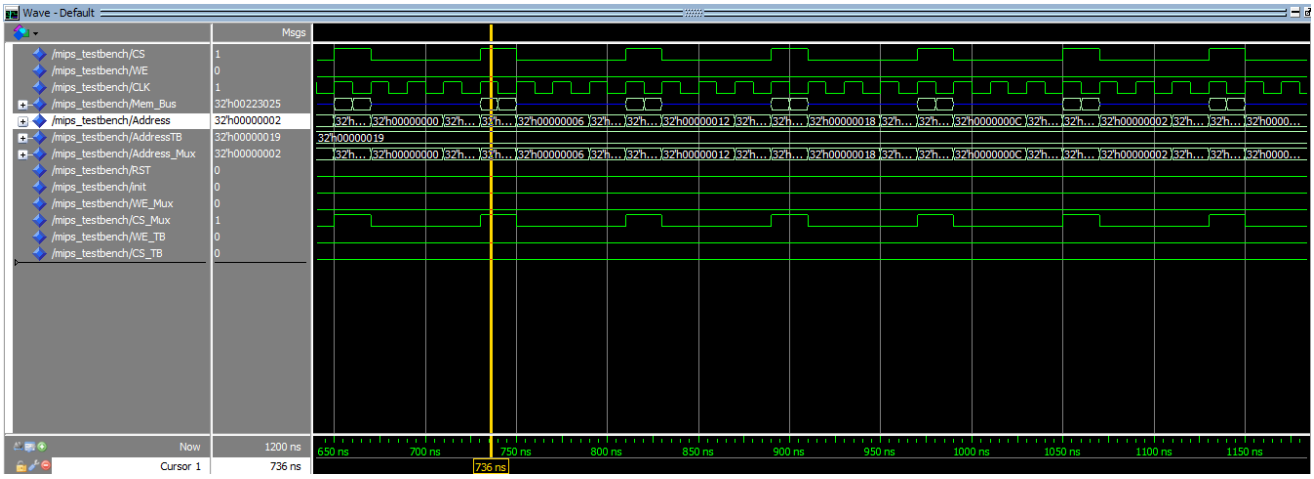


Testing the Processor module

The overall MIPS VHDL model is tested using a test bench. The test must verify the proper operation of each implemented instruction. The test bench consists of a MIPS program with the test instructions and VHDL code to load the program into memory and verify the program's output. We use a constant array of instructions that we want to write into the memory and a constant array of expected outputs to which we will compare the processor execution result.

However, note that now the memory is connected to the processor and test bench, and that means both our test bench and the processor will try to control the two signals at the same time. One way to resolve this is to put muxes at the input ports of the memory. There are a few muxes for that purpose: Address_Mux (for choosing the address), CS_Mux for choosing the CS Signal, and WE_Mux (for choosing the WE Signal). The select signal for the muxes is init. When the signal is '1', the three muxes select the address and CS and WE signals from the test bench. Otherwise, these signals from the processor module are chosen. We also assert the reset of our CPU throughout the initialization process to make sure the CPU does not run until the test bench finishes writing the instructions into the memory. When init is '0', the CPU and memory are connected for normal operation.

As the MIPS program executes, each test instruction stores its result in a different register. After all of the test instructions have been executed, the program performs a series of store instructions. Each of these instructions places the contents of a different register onto the bus as it executes. So if there are 10 instructions that we want to verify, we also have 10 store word instructions. During each store, the value on the bus is compared to the expected result for that register with an assert statement. MIPS processor, register $0 is always 0. We did not implement that in the register file. Hence we clear register $0 using an instruction. The first instruction in the test sequence does that. In normal MIPS processor code, you will not find instructions with register $10 as the destination. Essentially, writes to register $0 are ignored in MIPS.



Wave form for Test-bench

Source Codes

1. Register : register.vhd

```
architecture Behavioral of REG is
  type RAM is array (0 to 31) of unsigned(31 downto 0);
  signal Regs: RAM := (others => (others => '0'));
begin
  process(clk)
  begin
    if CLK = '1' and CLK'event then
 IF (RST = '0') THEN
 FOR i IN 0 TO 31 LOOP
 Regs(i) <= to_unsigned(i,32);
 END LOOP;
      elsif RegW = '1' and DR/=0 then
        Regs(to_integer(DR)) <= Reg_In;
      end if;
    end if;
  end process;
  ReadReg1 <= Regs(to_integer(SR1));
  ReadReg2 <= Regs(to_integer(SR2));
  reg_output_ext <= Regs(to_integer(reg_addr_ext));
end Behavioral;
```

2. Memory : memory.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Memory is
  port(CS, WE, Clk: in std_logic;
       ADDR: in unsigned(31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0);
       Addr_ext: in unsigned(6 downto 0);
       Output_ext: out unsigned(31 downto 0));

end Memory;

architecture Internal of Memory is
  type RAMtype is array (0 to 127) of unsigned(31 downto 0);
  signal RAM1: RAMtype := (0 => x"20420002",
                  1 => x"1000FFFF",

                  others => (others => '0'));
  signal output: unsigned(31 downto 0);
begin
  Mem_Bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" when CS = '0' or WE = '1'
    else output;
Output_ext <= RAM1(to_integer(Addr_ext));
  process(Clk)
  begin
    if Clk = '0' and Clk'event then
      if CS = '1' and WE = '1' then
```

```
        RAM1(to_integer(ADDR(6 downto 0))) <= Mem_Bus;
      end if;
    output <= RAM1(to_integer(ADDR(6 downto 0)));
    end if;
  end process;
end Internal;
```

3. Processor : processor.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity processor is
  port(CLK, RST, clkin: in std_logic;
       Addr_ext: in unsigned(6 downto 0);
            db :out Unsigned(7 downto 0);
        E: buffer STD_LOGIC;
            lcd_on, lcd_blon, RS, RW : out STD_LOGIC;
       reg_addr_ext: in unsigned(4 downto 0));
end processor;

architecture model of processor is
  component MIPS is
  port(CLK, RST: in std_logic;
       CS, WE: out std_logic;
      PC_out : out unsigned(31 downto 0);
       ADDR: out unsigned (31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0);
       reg_addr_ext: in unsigned(4 downto 0);
       reg_output_ext: out unsigned(31 downto 0));
  end component;
  component Memory is
  port(CS, WE, Clk: in std_logic;
       ADDR: in unsigned(31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0);
       Addr_ext: in unsigned(6 downto 0);
            Output_ext: out unsigned(31 downto 0));
  end component;

  COMPONENT lcddriver is
generic(clk_divider:integer :=50000);
port (clk,rst:in STD_LOGIC;
rs,rw:out STD_LOGIC;
E: buffer STD_LOGIC;
d1, d2, d3, d4, d5, d6, d7, d8: in UNSIGNED(7 downto 0);
X1,X2,X3,X4,X5,X6,X7,X8,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8 : in UNSIGNED(7 downto 0);
db :out Unsigned(7 downto 0);
lcd_on, lcd_blon : out STD_LOGIC);
end COMPONENT;

  signal CS, WE: std_logic;
  signal A_Out, D_Out: unsigned(31 downto 0);
  signal ADDR, Mem_Bus: unsigned(31 downto 0);
  signal Output_ext, reg_output_ext, PC_out:  unsigned(31 downto 0);
  SIGNAL D1,x1,y1 : unsigned(7 DOWNTO 0);
SIGNAL D2,x2,y2 : unsigned(7 DOWNTO 0);
```

```
SIGNAL D3,x3,y3 : UNSIGNED(7 DOWNTO 0);
SIGNAL D4,x4,y4 : UNSIGNED(7 DOWNTO 0);
SIGNAL D5,x5,y5 : UNSIGNED(7 DOWNTO 0);
SIGNAL D6,x6,y6 : UNSIGNED(7 DOWNTO 0);
SIGNAL D7,x7,y7 : UNSIGNED(7 DOWNTO 0);
SIGNAL D8,x8,y8 : UNSIGNED(7 DOWNTO 0);

FUNCTION binary_to_ascii (SIGNAL input: UNSIGNED(3 DOWNTO 0)) RETURN UNSIGNED
 IS VARIABLE output: UNSIGNED(7 DOWNTO 0);
 BEGIN
 CASE input IS
 WHEN "0000" => output:="00110000";
 WHEN "0001" => output:="00110001";
 WHEN "0010" => output:="00110010";
 WHEN "0011" => output:="00110011";
 WHEN "0100" => output:="00110100";
 WHEN "0101" => output:="00110101";
 WHEN "0110" => output:="00110110";
 WHEN "0111" => output:="00110111";
 WHEN "1000" => output:="00111000";
 WHEN "1001" => output:="00111001";
 WHEN "1010" => output:=X"41";
 WHEN "1011" => output:=X"42";
 WHEN "1100" => output:=X"43";
 WHEN "1101" => output:=X"44";
 WHEN "1110" => output:=X"45";
 WHEN "1111" => output:=X"46";
 WHEN OTHERS => output:=x"2D";
 END CASE;
 RETURN output;
 END binary_to_ascii;

begin
  CPU: MIPS port map (CLK, RST, CS, WE, PC_out, ADDR, Mem_Bus, reg_addr_ext,
reg_output_ext);
  MEM: Memory port map (CS, WE, CLK, ADDR, Mem_Bus,Addr_ext, Output_ext);
  A_Out <= Addr;
  D_Out <= Mem_Bus;

 D1<= binary_to_ascii(Output_ext(3 DOWNTO 0));
 D2<= binary_to_ascii(Output_ext(7 DOWNTO 4));
 D3<= binary_to_ascii(Output_ext(11 DOWNTO 8));
 D4<= binary_to_ascii(Output_ext(15 DOWNTO 12));
 D5<= binary_to_ascii(Output_ext(19 DOWNTO 16));
 D6<= binary_to_ascii(Output_ext(23 DOWNTO 20));
 D7<= binary_to_ascii(Output_ext(27 DOWNTO 24));
 D8<= binary_to_ascii(Output_ext(31 DOWNTO 28));

 X1<= binary_to_ascii(PC_out(3 DOWNTO 0));
 X2<= binary_to_ascii(PC_out(7 DOWNTO 4));
 X3<= binary_to_ascii(PC_out(11 DOWNTO 8));
 X4<= binary_to_ascii(PC_out(15 DOWNTO 12));
 X5<= binary_to_ascii(PC_out(19 DOWNTO 16));
 X6<= binary_to_ascii(PC_out(23 DOWNTO 20));
 X7<= binary_to_ascii(PC_out(27 DOWNTO 24));
 X8<= binary_to_ascii(PC_out(31 DOWNTO 28));

 Y1<= binary_to_ascii(reg_output_ext(3 DOWNTO 0));
 Y2<= binary_to_ascii(reg_output_ext(7 DOWNTO 4));
 Y3<= binary_to_ascii(reg_output_ext(11 DOWNTO 8));
 Y4<= binary_to_ascii(reg_output_ext(15 DOWNTO 12));
```

```
 Y5<= binary_to_ascii(reg_output_ext(19 DOWNTO 16));
 Y6<= binary_to_ascii(reg_output_ext(23 DOWNTO 20));
 Y7<= binary_to_ascii(reg_output_ext(27 DOWNTO 24));
 Y8<= binary_to_ascii(reg_output_ext(31 DOWNTO 28));

  LCD: LCDDRIVER port map(
                  clk=>clkin,
                  rst=>rst,
                  rs=>rs,rw=>rw,e=>e,
                  db=>db, lcd_on=>lcd_on, lcd_blon=>lcd_blon,
                  D1=>D1, D2=>D2,D3=>D3,D4=>D4,D5=>D5,D6=>D6,D7=>D7,D8=>D8,
                  X1=>X1,                  X2=>X2,                  X3=>X3,
X4=>X4,X5=>X5,X6=>X6,X7=>X7,X8=>X8,
                  Y1=>Y1,
Y2=>Y2,Y3=>Y3,Y4=>Y4,Y5=>Y5,Y6=>Y6,Y7=>Y7,Y8=>Y8);

end model;
```

4. Subset : Subset.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MIPS is
  port(CLK, RST: in std_logic;
       CS, WE: out std_logic;
      PC_out : out unsigned(31 downto 0);
       ADDR: out unsigned (31 downto 0);
       Mem_Bus: inout unsigned(31 downto 0);
       reg_addr_ext: in unsigned(4 downto 0);
       reg_output_ext: out unsigned(31 downto 0));
end MIPS;

architecture structure of MIPS is
  component REG is
  port(CLK, RST: in std_logic;
       RegW: in std_logic;
       DR, SR1, SR2: in unsigned(4 downto 0);
       Reg_In: in unsigned(31 downto 0);
       ReadReg1, ReadReg2: out unsigned(31 downto 0);
       reg_addr_ext: in unsigned(4 downto 0);
       reg_output_ext: out unsigned(31 downto 0));
  end component;
  type Operation is (and1,or1,add,sub,slt,shr,shl,jr);
  signal Op, OpSave: Operation := and1;
  type Instr_Format is (R, I, J);  -- (Arithmetic, Addr_Imm, Jump)
  signal Format: Instr_Format := R;
  signal Instr, Imm_Ext: unsigned (31 downto 0);
  signal PC, nPC, ReadReg1, ReadReg2, Reg_In: unsigned(31 downto 0);
  signal ALU_InA, ALU_InB, ALU_Result: unsigned(31 downto 0);
  signal ALU_Result_Save: unsigned(31 downto 0);
  signal ALUorMEM, RegW, FetchDorI, Writing, REGorIMM: std_logic := '0';
  signal REGorIMM_Save, ALUorMEM_Save: std_logic := '0';
  signal DR: unsigned(4 downto 0);
  signal State, nState : integer range 0 to 4 := 0;
  constant addi: unsigned(5 downto 0) := "001000";  -- 8
  constant andi: unsigned(5 downto 0) := "001100";  -- 12
```

```
  constant ori:  unsigned(5 downto 0) := "001101";  -- 13
  constant lw:   unsigned(5 downto 0) := "100011";  -- 35
  constant sw:   unsigned(5 downto 0) := "101011";  -- 43
  constant beq:  unsigned(5 downto 0) := "000100";  -- 4
  constant bne:  unsigned(5 downto 0) := "000101";  -- 5
  constant jump: unsigned(5 downto 0) := "000010";  -- 2
  alias opcode: unsigned(5 downto 0) is Instr(31 downto 26);
  alias SR1: unsigned(4 downto 0) is Instr(25 downto 21);
  alias SR2: unsigned(4 downto 0) is Instr(20 downto 16);
  alias F_Code: unsigned(5 downto 0) is Instr(5 downto 0);
  alias NumShift: unsigned(4 downto 0) is Instr(10 downto 6);
  alias ImmField: unsigned (15 downto 0) is Instr(15 downto 0);
begin
  PC_out<=PC-1;
  A1: Reg port map (CLK, RST, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2,
reg_addr_ext, reg_output_ext);
  Imm_Ext <= x"FFFF" & Instr(15 downto 0) when Instr(15) = '1'
    else x"0000" & Instr(15 downto 0);  -- Sign extend immediate field
  DR <= Instr(15 downto 11) when Format = R
    else Instr(20 downto 16);           -- Destination Register MUX (MUX1)
  ALU_InA <= ReadReg1;
  ALU_InB <= Imm_Ext when REGorIMM_Save = '1' else ReadReg2;    -- ALU MUX
(MUX2)
  Reg_in <= Mem_Bus when ALUorMEM_Save = '1' else ALU_Result_Save; -- Data
MUX
  Format <= R when Opcode = 0 else J when Opcode = 2 else I;
  Mem_Bus <= ReadReg2 when Writing = '1' else
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";  --  drive  memory  bus  only  during
writes
  ADDR <= PC when FetchDorI = '1' else ALU_Result_Save; --ADDR Mux

  process(State, PC, Instr, Format, F_Code, opcode, Op, ALU_InA, ALU_InB,
          Imm_Ext)
  begin
    FetchDorI <= '0'; CS <= '0'; WE <= '0'; RegW <= '0'; Writing <= '0';
    ALU_Result <= "00000000000000000000000000000000";
    npc <= pc; Op <= jr; REGorIMM <= '0'; ALUorMEM <= '0';
    case state is
      when 0 =>  --fetch instruction
        nPC <= PC + 1; CS <= '1'; nState <= 1;
        FetchDorI <= '1';
      when 1 =>
        nState <= 2; REGorIMM <= '0'; ALUorMEM <= '0';
        if Format = J then
          nPC <= "000000" & Instr(25 downto 0); nState <= 0; --jump, and
finish
        elsif Format = R then  -- register instructions
          if    F_code = "100000" then Op <= add;   -- add
          elsif F_code = "100010" then Op <= sub;   -- subtract
          elsif F_code = "100100" then Op <= and1;  -- and
          elsif F_code = "100101" then Op <= or1;   -- or
          elsif F_code = "101010" then Op <= slt;   -- set on less than
          elsif F_code = "000010" then Op <= shr;   -- shift right
          elsif F_code = "000000" then Op <= shl;   -- shift left
          elsif F_code = "001000" then Op <= jr;    -- jump register
          end if;
        elsif Format = I then -- immediate instructions
          REGorIMM <= '1';
          if Opcode = lw or Opcode = sw or Opcode = addi then Op <= add;
          elsif Opcode = beq or Opcode = bne then Op <= sub; REGorIMM <=
'0';
```

```
          elsif Opcode = andi then Op <= and1;
          elsif Opcode = ori then Op <= or1;
          end if;
          if Opcode = lw then ALUorMEM <= '1'; end if;
        end if;
      when 2 =>
        nState <= 3;
        if   OpSave = and1 then ALU_Result <= ALU_InA and ALU_InB;
        elsif OpSave = or1 then ALU_Result <= ALU_InA or ALU_InB;
        elsif OpSave = add then ALU_Result <= ALU_InA + ALU_InB;
        elsif OpSave = sub then ALU_Result <= ALU_InA - ALU_InB;
        elsif   OpSave   =   shr   then   ALU_Result   <=   ALU_InB   srl
to_integer(numshift);
        elsif   OpSave   =   shl   then   ALU_Result   <=   ALU_InB   sll
to_integer(numshift);
        elsif OpSave = slt then -- set on less than
          if ALU_InA < ALU_InB then ALU_Result <= X"00000001";
          else ALU_Result <= X"00000000";
          end if;
        end if;
        if ((ALU_InA = ALU_InB) and Opcode = beq) or
           ((ALU_InA /= ALU_InB) and Opcode = bne) then
          nPC <= PC + Imm_Ext; nState <= 0;
        elsif opcode = bne or opcode = beq then nState <= 0;
        elsif OpSave = jr then nPC <= ALU_InA; nState <= 0;
        end if;
      when 3 =>
        nState <= 0;
        if Format = R or Opcode = addi or Opcode = andi or Opcode = ori then
          RegW <= '1';
        elsif Opcode = sw then CS <= '1'; WE <= '1'; Writing <= '1';
        elsif Opcode = lw then CS <= '1'; nState <= 4;
        end if;
      when 4 =>
       nState <= 0; CS <= '1';
       if Opcode = lw then RegW <= '1'; end if;
    end case;
  end process;

  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      if rst = '0' then
        State <= 0;
        PC <= x"00000000";
      else
        State <= nState;
        PC <= nPC;
      end if;
      if State = 0 then Instr <= Mem_Bus; end if;
      if State = 1 then
         OpSave <= Op;
         REGorIMM_Save <= REGorIMM;
         ALUorMEM_Save <= ALUorMEM;
      end if;
      if State = 2 then ALU_Result_Save <= ALU_Result; end if;
    end if;
  end process;
end structure;
```

5. LCD Driver : LCD.vhd

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


entity lcddriver is
generic(clk_divider:integer :=50000);
port (clk,rst:in STD_LOGIC;
rs,rw:out STD_LOGIC;
E: buffer STD_LOGIC;
d1, d2, d3, d4, d5, d6, d7, d8 : in UNSIGNED(7 downto 0);
X1,X2,X3,X4,X5,X6,X7,X8,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8 : in UNSIGNED(7 downto 0);
db :out Unsigned(7 downto 0);
lcd_on, lcd_blon : out STD_LOGIC);
end lcddriver;
architecture behavioral of lcddriver is
type                                    state                                    is(
FunctionSet1,FunctionSet2,FunctionSet3,FunctionSet4,ClearDisplay,DisplayCon
trol,EntryMode,WriteData1,WriteData2,WriteData3,WriteData4,WriteData5,Write
Data6, WriteData7, WriteData8,
                        WriteData9,        WriteData10,        WriteData11,
WriteData12,WriteData13,  WriteData14,WriteData15,WriteData16,  WriteData17,
WriteData18,

      WriteData19,WriteData20,WriteData21,WriteData22,WriteData23,WriteData
24,WriteData25,WriteData26, WriteData27, WriteData28,
                        WriteData29,WriteData30,WriteData31,WriteData32,
ReturnHome,Nextline);
signal pr_state,nx_state:state;
begin

 process (clk)
variable count :integer range 0 to clk_divider;
begin
if(clk'event and clk='1')then                          -------clock divider
for 50Hz
count :=count +1;
if(count=clk_divider)then
 E<=not E;
count:=0;
end if;
end if;
end process ;

 process(E)
begin
if(E'event and E='1') then
if(rst='0') then
pr_state<=FunctionSet1;
else
pr_state<=nx_state;
end if;
end if;
end process ;

 process(pr_state)
begin                                          -------Initialization  code
1
```

```
case pr_state is
when FunctionSet1 =>
lcd_on<='1';
lcd_blon<='1';
rs<='0';
rw<='0';
db<="00111000";
nx_state<= FunctionSet2;

when FunctionSet2 =>                                    -------Initialization
code 2
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db<="00111000";
nx_state <= FunctionSet3;

when FunctionSet3 =>                                    -------Initialization
code 3
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="00111000";
nx_state <= FunctionSet4;

when FunctionSet4 =>                                    -------Initialization
code 4
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="00111000";
nx_state <= ClearDisplay;

when ClearDisplay =>                                    -------Clear  Display
Data
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="00000001";
nx_state <= DisplayControl;

when displayControl =>                                  -------Display On
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="00001100";
nx_state <=EntryMode;

when EntryMode =>                                      -------Set   the   cursor
moving direction
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="00000110";
nx_state <= WriteData1;
```

```
when WriteData1 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"52";
nx_state <= WriteData2;

when WriteData2 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"65";
nx_state <= WriteData3;

when WriteData3 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"67";
nx_state <= WriteData4;

when WriteData4 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"20";
nx_state <= WriteData5;

when WriteData5 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=x8;
nx_state <= WriteData6;

when WriteData6 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=x7;
nx_state <= WriteData7;

when WriteData7 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=x6;
nx_state <= WriteData8;

when WriteData8 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
```

```
rw <='0';
db <=x5;
nx_state <= Writedata9;

when WriteData9 =>                                  -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X4;
nx_state <= Writedata10;

when WriteData10 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X3;
nx_state <= Writedata11;

when WriteData11 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=x2;
nx_state <= Writedata12;

when WriteData12 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=x1;
nx_state <= Writedata13;

when WriteData13 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"20";
nx_state <= Writedata14;

when WriteData14 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"4d";
nx_state <= Writedata15;

when WriteData15 =>                                 -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"65";
nx_state <= Writedata16;

when WriteData16 =>                                 -------Write data
```

```
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=X"6d";
nx_state <= Nextline;

when Nextline =>
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <=X"C0";
nx_state <= Writedata17;

when WriteData17 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y8;
nx_state <= Writedata18;

when WriteData18 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y7;
nx_state <= Writedata19;

when WriteData19 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y6;
nx_state <= Writedata20;

when WriteData20 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y5;
nx_state <= Writedata21;

when WriteData21 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y4;
nx_state <= Writedata22;

when WriteData22 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y3;
```

```
nx_state <= Writedata23;

when WriteData23 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y2;
nx_state <= Writedata24;

when WriteData24 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=y1;
nx_state <= Writedata25;

when WriteData25 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d8;
nx_state <= Writedata26;

when WriteData26 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d7;
nx_state <= Writedata27;

when WriteData27 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d6;
nx_state <= Writedata28;

when WriteData28 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d5;
nx_state <= Writedata29;

when WriteData29 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d4;
nx_state <= Writedata30;

when WriteData30 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
```

```
rs <='1';
rw <='0';
db <=d3;
nx_state <= Writedata31;

when WriteData31 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d2;
nx_state <= Writedata32;

when WriteData32 =>                                    -------Write data
lcd_on <='1';
lcd_blon <='1';
rs <='1';
rw <='0';
db <=d1;
nx_state <= ReturnHome;

when ReturnHome =>
lcd_on <='1';
lcd_blon <='1';
rs <='0';
rw <='0';
db <="10000000";
nx_state <=WriteData1;
end case;
end process;
end behavioral;
```

6. Intergrating Processor and Memory : integratn_mem.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Complete_MIPS is
  port(CLK, RST: in std_logic;
       A_Out, D_Out: out unsigned(31 downto 0));
end Complete_MIPS;

architecture model of Complete_MIPS is
  component MIPS is
    port(CLK, RST: in std_logic;
         CS, WE: out std_logic;
         ADDR: out unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  component Memory is
    port(CS, WE, Clk: in std_logic;
         ADDR: in unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  signal CS, WE: std_logic;
  signal ADDR, Mem_Bus: unsigned(31 downto 0);
begin
```

```
  CPU: MIPS port map (CLK, RST, CS, WE, ADDR, Mem_Bus);
  MEM: Memory port map (CS, WE, CLK, ADDR, Mem_Bus);
  A_Out <= Addr;
  D_Out <= Mem_Bus;
end model;
```

7. Test bench : test_bench.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MIPS_Testbench is
end MIPS_Testbench;

architecture test of MIPS_Testbench is
  component MIPS
    port(CLK, RST: in std_logic;
         CS, WE: out std_logic;
         ADDR: out unsigned (31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;
  component Memory
    port(CS, WE, CLK: in std_logic;
         ADDR: in unsigned(31 downto 0);
         Mem_Bus: inout unsigned(31 downto 0));
  end component;

  constant N: integer := 8;
  constant W: integer := 26;
  type Iarr is array(1 to W) of unsigned(31 downto 0);
  constant Instr_List: Iarr := (
    x"30000000", -- andi $0, $0, 0  => 0. $0 = 0
    x"20010006", -- addi $1, $0, 6  => 1. $1 = 6
    x"34020012", -- ori $2, $0, 18  => 2. $2 = 18
    x"00221820", -- add $3, $1, $2  => 3. $3 = $1 + $2 = 24
    x"00412022", -- sub $4, $2, $1  => 4. $4 = $2 - $1 = 12
    x"00222824", -- and $5, $1, $2  => 5. $5 = $1 and $2 = 2
    x"00223025", -- or $6, $1, $2   => 6. $6 = $1 or $2 = 22
    x"0022382A", -- slt $7, $1, $2  => 7. $7 = 1 because $1<$2
    x"00024100", -- sll $8, $2, 4   => 8. $8 = 18 * 16 = 288
    x"00014842", -- srl $9, $1, 1   => 9. $9 = 6/2 = 3
    x"10220001", -- beq $1, $2, 1   => 10. Will not branch. $10 incorrect if
fails.
    x"8C0A0004", -- lw $10, 4($0)    => 11. $10 = 5th instr = x"00412022" =
4268066
    x"14620001", -- bne $1, $2, 1    => 12. Will branch to PC+1+1. $1 wrong
if fails
    x"30210000", -- andi $1, $1, 0  => 13. $1 = 0 (skipped)
    x"08000010", -- j 16              => 14. PC = 16 = PC+1+1. $2 wrong if
fails
    x"30420000", -- andi $2, $2, 0  => 15. $2 = 0 (skipped)
    x"00400008", -- jr $2            => 16. PC = $2 = 18 = PC+1+1. $3 wrong
if fails
    x"30630000", -- andi $3, $3, 0  => 17. $3 = 0 (skipped)
    x"AC030040", -- sw $3, 64($0)   => 18. Mem(64) = $3
    x"AC040041", -- sw $4, 65($0)   => Mem(65) = $4
    x"AC050042", -- sw $5, 66($0)   => Mem(66) = $5
```

```
    x"AC060043", -- sw $6, 67($0)   => Mem(67) = $6
    x"AC070044", -- sw $7, 68($0)   => Mem(68) = $7
    x"AC080045", -- sw $8, 69($0)   => Mem(69) = $8
    x"AC090046", -- sw $9, 70($0)   => Mem(70) = $9
    x"AC0A0047"  -- sw $10, 71($0)  => Mem(71) = $10
);
    -- The last instructions perform a series of sw operations that store
    -- registers 3-10 to memory. During the memory write stage, the testbench
    -- will compare the value of these registers (by looking at the bus
value)
    -- with the expected output. No explicit check/assertion for branch
    -- instructions, however if a branch does not execute as expected, an
error
    -- will be detected because the assertion for the instruction after the
    -- branch instruction will be incorrect.
  type output_arr is array(1 to N) of integer;
  constant expected: output_arr:= (24, 12, 2, 22, 1, 288, 3, 4268066);
  signal CS, WE, CLK: std_logic := '0';
  signal Mem_Bus, Address, AddressTB, Address_Mux: unsigned(31 downto 0);
  signal RST, init, WE_Mux, CS_Mux, WE_TB, CS_TB: std_logic;
begin
  CPU: MIPS port map (CLK, RST, CS, WE, Address, Mem_Bus);
  MEM: Memory port map (CS_Mux, WE_Mux, CLK, Address_Mux, Mem_Bus);

  CLK <= not CLK after 10 ns;
  Address_Mux <= AddressTB when init = '1' else Address;
  WE_Mux <= WE_TB when init = '1' else WE;
  CS_Mux <= CS_TB when init = '1' else CS;

  process
  begin
    rst <= '1';
    wait until CLK = '1' and CLK'event;

    --Initialize the instructions from the testbench
    init <= '1';
    CS_TB <= '1'; WE_TB <= '1';
    for i in 1 to W loop
      wait until CLK = '1' and CLK'event;
      AddressTB <= to_unsigned(i-1,32);
      Mem_Bus <= Instr_List(i);
    end loop;
    wait until CLK = '1' and CLK'event;
    Mem_Bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    CS_TB <= '0'; WE_TB <= '0';
    init <= '0';
    wait until CLK = '1' and CLK'event;
    rst <= '0';

    for i in 1 to N loop
      wait until WE = '1' and WE'event;  -- When a store word is executed
      wait until CLK = '0' and CLK'event;
      assert(to_integer(Mem_Bus) = expected(i))
        report "Output mismatch:" severity error;
    end loop;

    report "Testing Finished:";
  end process;
end test;
```

Conclusion

We have successfully completed a 32 bit RISC Microprocessor in VHDL language and implemented on Altera FPGA. The Test bench module is executed in the model-sim software and the LCD module is implemented on the FPGA to display the Register value, Memory value and the Program counter.