CPU Design and Implementation

ECE 250 Final Project

20/12/2014

Haneen Mohammad S10102109
Areej Sabo S12102863
Tala Sallam S12103228

Abstract

Central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations specified by the instructions. In this project, a 32-bit non-pipelined processor was design and implementation using VHDL. The processor was tested using a program stored in an instruction memory of type ROM (Read Only Memory). The simulation worked successfully and the ALU Control unit generated outputs for all inputs successfully.

Contents

List of Figures

1 INTRODUCTION AND OVERVIEW

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations specified by the instructions. The term has been used in the computer industry at least since the early 1960s. The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains much the same. Principal components of a CPU include the arithmetic logic unit (ALU), which performs arithmetic and logical operations, and the control unit (CU), which fetches instructions from memory and decodes and executes them, calling on the services of the ALU when necessary. Most modern CPUs are microprocessors, meaning they are contained on a single integrated circuit (IC) chip. Some computers have two or more CPUs on a single chip and thus are capable of multiprocessing. An IC that contains a CPU may also contain memory, peripheral devices, and other components of a computer.

1.1 Software Used

- ISE
- ModelSim

1.2 System Specification

- Data memory of 128B.
- Instruction memory to store the provided program.
- A 32-bit general purpose register: R0 - R7.
- Two special purpose registers: PC (program counter), &IR (instruction register)
- Two special purpose registers: MAR (Memory Address Register), & MDR (Memory Data Register)

1.3 Instruction encoding

To encode the instruction, a 16-bit format was chosen as shown in the table below.

3 bits	3 bits	3 bits	7 bits
opcode	REG_OP1	REG_OP2	REG_OP3/IMMEDIATE

1.4 Encoding of OPCODES

With a 3 bit opcode 8 instructions are possible. In our system only 7 are used, which are:

Instruction	OPCODE	Description
XOR	000	Logic XOR for op2 and op3
ADDI	001	Add the value in register[op2] to the immediate value
LOAD	010	load memory[op3] into resgister[op2]
STORE	011	store value in resgister[op2] into memory[op3]
BLT	100	(op1 less than op2) ?
BNEQ	101	(op1 == op2) ?
EOP	110	halt program

1.5 Algorithm of the processor

1. Instruction is read form the Instruction memory:
   (a) The address of the instruction given by the Program Counter (PC).
   (b) After the fetch, the PC is incremented.
   (c) The instruction is stored in the Instruction Register (IR).
2. The instruction in IR is passed to the instruction decoder that passes the opcode to the control unit to execute the instruction accordingly:
   (a) ADDI: one of the operands of the ALU would be taken from the register file based on the register specified in the instruction IR[9:7] and the second would be an immediate value IR[6:0] and the result would be saved in the register[IR[12:10]].
   (b) XOR: the two operands of the ALU would be taken from the register file of indexes IR[9:7] and IR[2:0] and the result will be saved in the register[IR[12:10]].
   (c) BLT: to test if the value in the register of index IR[12:10] I larger than of index IR[9:7]. The two values in the register will be passed to the ALU and the result would be either 1 if condition was true or 0 otherwise. If true update the PC by the value in IR[3:0] .
   (d) BNEQ: same as BLT except it test the equality of the two operands passed to the ALU.
   (e) LOAD: load from memory of index IR[9:7] into register file of index IR[12:10].
   (f) STORE: store to memory of index IR[9:7] from register file of index IR[12:10].
   (g) EOP: end of the program.
3. The appropriate data is then written either to the register file, or data memory based on the instruction.

2 DESIGN AND IMPLEMENTATION

In this project a CPU with the previews specification would be designed and implemented in VHDL. The following program is stored in the instruction memory in binary format using the instruction encoding specified above.

- SORT PROGRAM
  i0: XOR R1 R1 R1// R1=0
  i1: ADDI R5 R1 0x50 // R5=80 (0x50)
  i2: ADDI R6 R1 0x4C // R6=76 (0x4C)
  i3: ADDI R2 R1 0x4 // R2=R1+4
  i4: LOAD R3 (R1) // R3=DataMem[R1]
  i5: LOAD R4 (R2) // R4=DataMem[R2]
  i6: BLT R3 R4 i9 // if (R3¡R4) jump to instruction i9
  i7: STORE R3 (R2) // DataMem[R2]=R3
  i8: STORE R4 (R1) // DataMem[R1]=R4
  i9: ADDI R2 R2 0x4 // R2=R2+4
  i10: BNEQ R2 R5 i4 // if (R2!=R5) jump to instruction i4 recall that R5 = 80
  i11: ADDI R1 R1 0x4 // R1=R1+4
  i12: BNEQ R1 R6 i3 // if (R1!=R6) jump to instruction i3 recall that R6 = 76
  i13: EOP

During every cycle, the CPU will execute a single instruction and will not move to the next instruction until the present instruction has been completed, and the program counter is updated.

2.1 Module Specification

The following are the major component in the designed CPU. A common input for all the modules is the clock which is used to synchronize the operations between them.

2.1.1 Instruction Memory

Instructions specify commands to Transfer information or Perform arithmetic and logic operations. program is a sequence of instructions to perform a task, and it is stored in the instruction memory. The program to be executed using this CPU has 14 instructions, for that an instruction memory of [16 bits] by [16 bits] were designed and the program provided was stored inside the memory that is read later during instruction fetch. Figure 1 shows the simulation for the instruction memory with test vector for addresses from 0 until 1110 to show the stored instructions inside the memory.

- addr: [3:0] the program counter PC is used to access the instructions in the Instruction memory
- dout: [31:0] Fetch an instruction and save it to the IR register

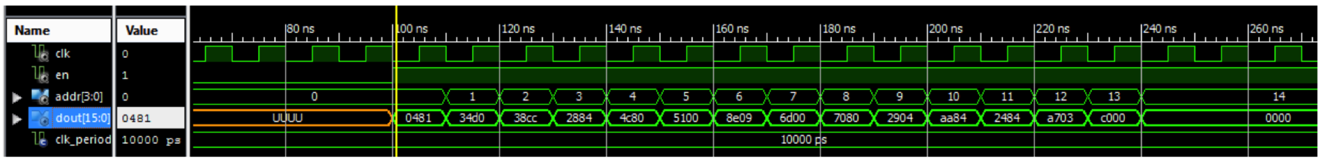

Figure 1: Instruction Memory Simulation results

2.1.2 Data Memory

Data memory is used for storing data used or processed by the program. It is a sequential component. For the CPU to be designed, it has a128B data memory which would be implemented as [32 bit] by [32 bit] registers. Memory ports: - addr: [4:0] To point to the memory location to be read from or written to. - din: [31:0] The data would be written in the memory. - dout: [31:0] Data read out of the memory - r_w: memory write or read enable. Data on din get written when r_w == 1 is set.

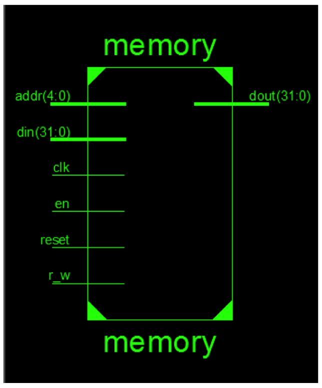

Figure 2: Main Memory

2.1.3 Register Files and Special purpose registers

The register file is sequential component. In this CPU the register file were designed to have 8 registers labeled R0,R1,..R7; each of 32-bits wide. The module has the following ports:

- din : Data to be written in the registers [31:0]
- dout : Data read out of the registers [31:0]
- addr : To specify the register to be written in when write enable is set [2:0]
- en: if set high changes are allowed in the contents of the file
- r_w: if set high data in din is written inside the register pointed to by addr

In addition to the general purpose registers, special registers are used for specific purposes:
Memory Address Register (MAR): 5 bits Address of the memory location to be accessed
Memory Data Register (MDR): 32 bit wide, stores data to be read into or read out of the current location
Instruction Register (IR): Instruction that is currently being executed
Program Counter (PC): 4 bits address of the next instruction to be fetched and executed. It either increment automatically or if jump operation it use the address specified in op3 of the instruction. Figure 3 shows the PC module

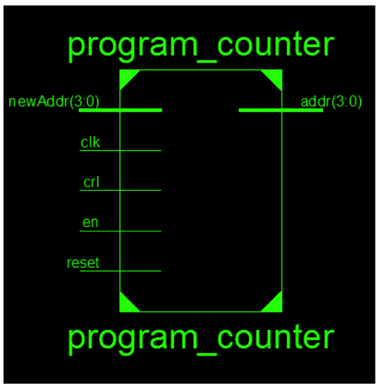

Figure 3: Program Counter Module

2.1.4 ALU

The ALU component is combinational. It performs the computations required based on the executed instruction and the operands at the inputs.

- dinA : the first operand [31:0] wide
- dinB : the second operand [31:0] wide
- opcode : the operation performed [2:0] wide
- result : results relevant when ADDI or XOR [31:0]
- blt and bneg flags: results relevant when BNEQ or BLT

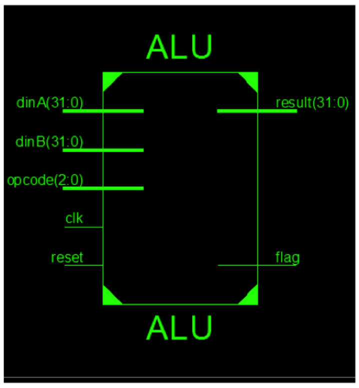

Figure 4: ALU Module

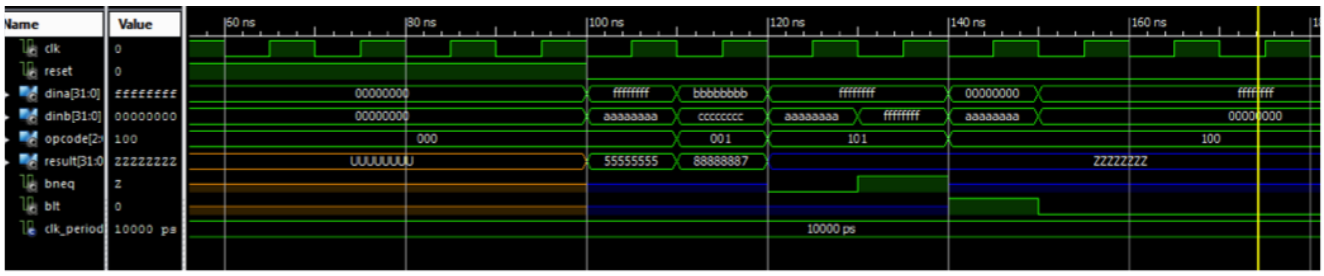

Figure 5: ALU Simulation results

2.1.5 Control Unit

The Control unit is the heart of the CPU. It direct operations of the processor through setting the control lines as appropriate for the instruction. After studying the instruction set provided the major states was recognized and a state machine diagram for instruction specific was designed Figure 6:

1. Instruction fetch
2. Instruction decode
3. Instruction execute

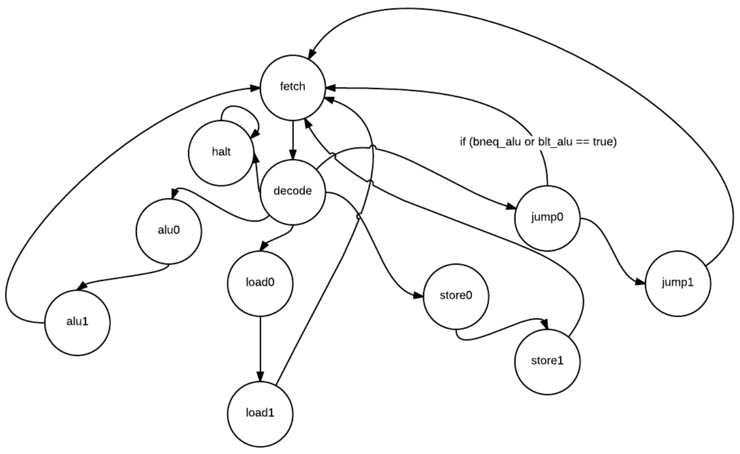

Figure 6: Finite State Machine

2.2 Simulation Results

At this stage we have designed all the necessary components for single cycle processor. Our next and final task is to merge everything together and simulate the CPU. Each module in isolation was simulated and verified. Figure shows the simulation of the top module which is a top-level schematic that ties all of the system components together. The program comes to halt after 40405 ns Figure 8.

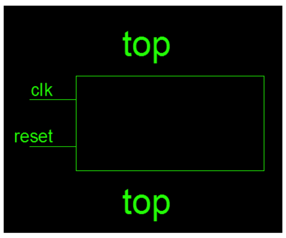

Figure 7: Top module

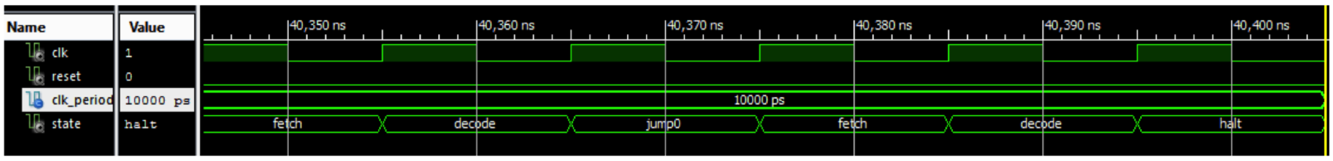

Figure 8: CPU simulation

3 CONCLUSION

We successfully managed to implement a working CPU. The simulation worked like it was expected. The ALU Control unit generated correct output for every input like it was described. The ALU carried out all the arithmetic operations correctly, and the integrated MUX was performing correctly. For us to be able to accomplish this task we learned a lot of very useful information about how the CPU works. Now we have much better understanding how the CPU works which includes: register file, instruction fetcher, memory, control unit and ALU. The design and implementation can be further improved to improve the performance.

Appendices

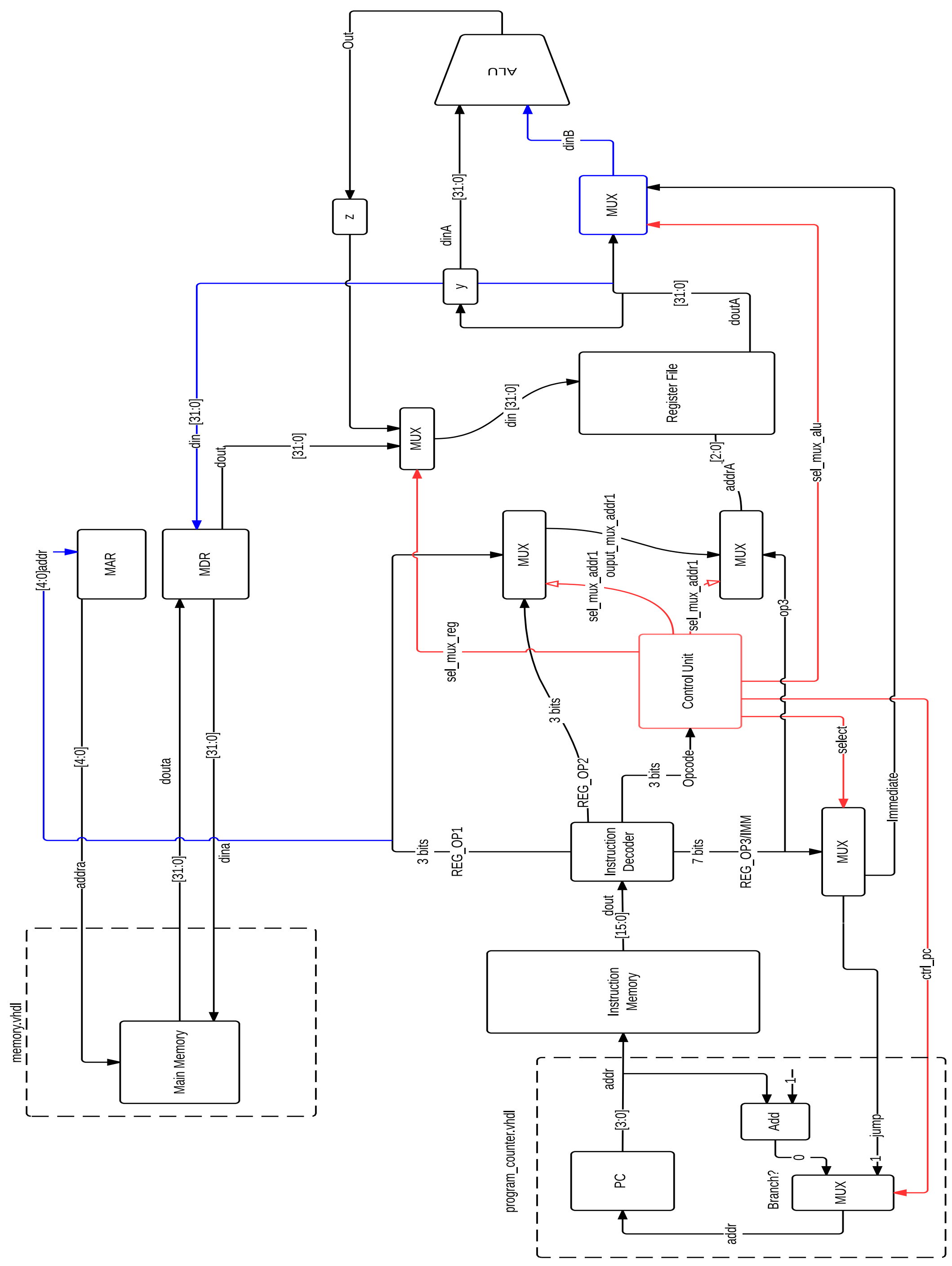

Figure 9: CPU Design

Listing 1: Top

```
--
    ------------------------------------------------------------------------------------
-- Engineer: Haneen Mohammed
-- Create Date:    11:07:31 12/13/2014
-- Module Name:    top - Behavioral
-- Project Name: 32 bits unpiplined CPU
--
    ------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity top is
port (
  clk, reset : IN STD_LOGIC
);
end top;

architecture Behavioral of top is

 COMPONENT mux
    PORT(
         inputA : IN  std_logic_vector(31 downto 0);
         inputB : IN  std_logic_vector(31 downto 0);
         sel : IN  std_logic;
         output : OUT  std_logic_vector(31 downto 0)
        );
    END COMPONENT;

 COMPONENT mux3b
    PORT(
         inputA : IN  std_logic_vector(2 downto 0);
         inputB : IN  std_logic_vector(2 downto 0);
         sel : IN  std_logic;
         output : OUT  std_logic_vector(2 downto 0)
        );
    END COMPONENT;

 COMPONENT program_counter
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         crl : IN  std_logic; --from control
         en : IN  std_logic; --from control
         newAddr : IN  std_logic_vector(3 downto 0); -- from instruction
         addr : OUT  std_logic_vector(3 downto 0) --defualt
        );
    END COMPONENT;

  COMPONENT instruction_memory
    PORT(
         clk  : IN   std_logic;
         en : IN   std_logic; --from control
         addr : IN   std_logic_vector(3 downto 0); -- from PC
         dout : OUT  std_logic_vector(15 downto 0) -- to instruction
  decoder
        );
    END COMPONENT;

  COMPONENT instruction_decoder
    PORT(
         clk  : IN   std_logic;
         reset : IN   std_logic;
         en : IN   std_logic; --from control
         dbus : IN   std_logic_vector(15 downto 0);
         opcode : OUT  std_logic_vector(2 downto 0);
         op1  : OUT  std_logic_vector(2 downto 0);
         op2  : OUT  std_logic_vector(2 downto 0);
         op3  : OUT  std_logic_vector(6 downto 0)
        );
    END COMPONENT;

   COMPONENT ALU
    PORT(
         clk  : IN   std_logic;
         reset  : IN   std_logic;
         dinA  : IN   std_logic_vector(31 downto 0);
         dinB  : IN   std_logic_vector(31 downto 0);
         opcode  : IN   std_logic_vector(2 downto 0);
         result  : OUT  std_logic_vector(31 downto 0);
         bneq, blt: OUT  std_logic  --to control
        );
    END COMPONENT;


    COMPONENT memory
    PORT(
         clk  : IN   std_logic;
         r_w  : IN   std_logic; --from control
         en : IN   std_logic; --from control
         reset  : IN   std_logic;
         addr  : IN   std_logic_vector(4 downto 0); -- abus
         din  : in   std_logic_vector(31 downto 0); -- dbus
      dout: out  std_logic_vector(31 downto 0) -- dbus
        );
    END COMPONENT;

   COMPONENT memory_address_register
    PORT(
         clk  : IN   std_logic;
      reset  : IN   std_logic;
         en : IN   std_logic; --from control
      r_w  : IN   std_logic; --from control
      din  : IN   std_logic_vector(4 downto 0);
         dout  : OUT  std_logic_vector(4 downto 0)
        );
    END COMPONENT;

   COMPONENT reg32bits
    PORT(
         clk  : IN   std_logic;
      reset  : IN   std_logic;
         en  : IN   std_logic;  --from control
      r_w  : IN   std_logic; --from control
         din  : IN   std_logic_vector(31 downto 0);
         dout  : OUT  std_logic_vector(31 downto 0)
        );
    END COMPONENT;

   COMPONENT register_file
    PORT(
         clk  : IN   std_logic;
         reset  : IN   std_logic;
         en  : IN   std_logic; --from control
         r_w  : IN   std_logic;--from control
         dout  : OUT  std_logic_vector(31 downto 0); -- to mem/alu  -- dbus
         addr  : IN   std_logic_vector(2 downto 0); -- from instruction
         din  : IN   std_logic_vector(31 downto 0) --from alu/mem   -- dbus
        );
    END COMPONENT;

 COMPONENT control_unit
    PORT(
         clk  : IN   std_logic;
         reset  : IN   std_logic;
         crl_pc  : OUT  std_logic; -- to PC
         en_pc  : OUT  std_logic;  -- to PC
         en_irm  : OUT  std_logic; -- to Instruction Memory
         en_id  : OUT  std_logic; -- to Instruction Decoder
         opcode  : IN   std_logic_vector(2 downto 0); -- from ALU
         alu_mux  : OUT  std_logic; -- t
      r_w_z  : OUT  std_logic;
      en_z  : OUT  std_logic;
      r_w_y  : OUT  std_logic;
      en_y  : OUT  std_logic;
      reg_mux: OUT  std_logic;
         bneq_alu  : IN   std_logic;
         blt_alu  : IN   std_logic;
         en_mar  : OUT  std_logic;
      r_w_mar  : OUT  std_logic;
         en_mdr  : OUT  std_logic;
      r_w_mdr  : OUT  std_logic;
         r_w_mem  : OUT  std_logic;
         en_mem  : OUT  std_logic;
         en_reg  : OUT  std_logic;
         r_w_reg  : OUT  std_logic;
      sel_mux_addr1  : OUT std_logic;
         sel_mux_addr  : OUT std_logic
        );
    END COMPONENT;

```

```
-- signals for PC
signal addr_pc :std_logic_vector(3 downto 0);
signal en_pc :std_logic;
signal crl_pc :std_logic;

 -- signals for Instruction Memory
signal dout_irm :std_logic_vector(15 downto 0);
signal en_irm :std_logic;
 -- signals for Instruction decoder
 signal en_id :std_logic;
 signal opcode_ird, op1_ird, op2_ird  :std_logic_vector(2 downto 0);
signal op3_ird :std_logic_vector(6 downto 0);
  -- signals for alu
signal bneq_alu, blt_alu  :std_logic;

 -- signals for mem
signal dout_mem :std_logic_vector(31 downto 0);
signal r_w_mem, en_mem :std_logic:='0';

-- MAR
signal en_mar : std_logic;
signal dout_mar : std_logic_vector(4 downto 0);
signal r_w_mar :std_logic;

-- MDR
signal en_mdr : std_logic;
signal dout_mdr : std_logic_vector(31 downto 0);
signal r_w_mdr :std_logic;

-- signals for Reg file
 signal  dout_reg :std_logic_vector(31 downto 0);
signal en_reg, r_w_reg :std_logic;

signal sel_mux_alu :std_logic;
signal sel_mux_reg :std_logic;

signal output_mux_alu : std_logic_vector(31 downto 0);
signal output_mux_reg : std_logic_vector(31 downto 0);
signal alu_out : std_logic_vector(31 downto 0);
signal result : std_logic_vector(31 downto 0);

-- z register
signal r_w_z :std_logic;
signal en_z :std_logic;
signal dout_z : std_logic_vector(31 downto 0);

-- y register
signal r_w_y :std_logic;
signal en_y :std_logic;
signal dout_y : std_logic_vector(31 downto 0);


signal sel_mux_addr1  :std_logic;
signal sel_mux_addr  :std_logic;
signal output_mux_addr1 : std_logic_vector(2 downto 0);
```

```vhdl
signal output_mux_addr  : std_logic_vector(2 downto 0);


begin
 ctl: control_unit PORT MAP (
          clk => clk,
          reset => reset,
          crl_pc => crl_pc, --to pc
          en_pc => en_pc, -- to pc
          en_irm => en_irm, -- to instruction mem
          en_id => en_id, -- to instruction mem
          opcode => opcode_ird, -- from decoder
          alu_mux => sel_mux_alu, -- to mux
       r_w_z => r_w_z,
       en_z => en_z,
       r_w_y => r_w_y,
       en_y => en_y,
       reg_mux => sel_mux_reg,
          bneq_alu => bneq_alu, -- from alu
          blt_alu => blt_alu, -- from alu
          en_mar => en_mar,-- to mar
       r_w_mar => r_w_mar,
          en_mdr => en_mdr, -- to mar
       r_w_mdr => r_w_mdr,
          r_w_mem => r_w_mem, -- to mem
          en_mem => en_mem, -- to mem
          en_reg => en_reg, -- to reg
          r_w_reg => r_w_reg, -- to reg
          sel_mux_addr1  => sel_mux_addr1,
         sel_mux_addr  => sel_mux_addr
      );
  pc: program_counter port map(
     clk => clk,
      reset =>   reset,
      crl=>   crl_pc, -- from control unit
      en  =>   en_pc,  -- from control unit
      newAddr =>   op3_ird(3 downto 0), -- if the instruction specified an
   address to jumb to
      addr =>   addr_pc   -- out to intruction reg
  );
  irm: instruction_memory port map(
     clk => clk,
      en  => en_irm,  -- from control unit
      addr  => addr_pc, -- from PC
      dout => dout_irm -- to IR

  );
  ird: instruction_decoder PORT MAP (
          clk => clk,
          reset => reset,
          en => en_id,-- from control unit
          dbus => dout_irm,-- from IRM
          opcode => opcode_ird,-- to control unit and alu
          op1 => op1_ird,
          op2 => op2_ird,
          op3 => op3_ird
  );
  y: reg32bits PORT MAP (
          clk => clk,
      reset => reset,
          en => en_y,-- from control unit
      r_w => r_w_y,-- from control unit
          din => dout_reg, -- from register
          dout => dout_y -- to memory
        );
  mux_alu: mux PORT MAP (
          inputA => dout_y,
          inputB => "000000000000000000000000" & op3_ird,
          sel => sel_mux_alu,
          output => output_mux_alu
        );

  alu2: ALU PORT MAP (
          clk => clk,
          reset => reset,
          dinA => dout_reg, -- always from Register
          dinB => output_mux_alu, -- either Register or Immediate value
          opcode => opcode_ird, -- should it be only from Ucontroller?
          result => alu_out, -- always to a register | (result_alu)
          bneq => bneq_alu, -- to control
      blt=> blt_alu -- to control
      );

  mux_reg: mux PORT MAP (
          inputA => dout_mem,
          inputB => dout_z,
          sel => sel_mux_reg,
          output => output_mux_reg
        );

  z: reg32bits PORT MAP (
          clk => clk,
      reset => reset,
          en => en_z,-- from control unit
      r_w => r_w_z,-- from control unit
          din => alu_out, -- from register
          dout => dout_z -- to memory
        );
  mux_reg_addr: mux3b PORT MAP (
          inputA => op1_ird, -- sel = 1 when alu1  to write to it, and when
    jump0
          inputB => op2_ird, -- sel = 0 when decode
          sel => sel_mux_addr1,
          output => output_mux_addr1
        );
  mux_reg_addr2: mux3b PORT MAP (
          inputA => output_mux_addr1, --sel = 1 when decode, alu1
          inputB => op3_ird(2 downto 0), --sel = 0 when alu0
          sel => sel_mux_addr,
          output => output_mux_addr
```

```vhdl
        );
  registers: register_file PORT MAP (
            clk => clk,
            reset => reset,
            en => en_reg,-- from control unit
            r_w => r_w_reg,-- from control unit
            dout => dout_reg , -- to memory or alu
            addr => output_mux_addr, -- from the instruction
            din => output_mux_reg  -- either from mem or result of ALU
        );
      --reslut alu
      -- XOR R5 R1 R2 : R5 = R1 xor R2
      -- ADDI R5 R1 0x50 : R5 = R1 + 0x50
      --from mem
      -- LOAD R3 (R1) : R3 = DataMem[R1]
  ram: memory PORT MAP (
            clk => clk,
            r_w => r_w_mem,-- from control unit
            en => en_mem,-- from control unit
            reset => reset,
            addr => dout_mar, -- mar
            din => dout_mdr, -- from mdr
        dout => dout_mem
        );
  -- used for load and store
  -- LOAD R3 (R1) : R3 = DataMem[R1]
   mar: memory_address_register PORT MAP (
            clk => clk,
        reset => reset,
        en => en_mar, -- from control unit
        r_w => r_w_mar,-- from control unit
            din => "00" & op2_ird,  -- from instruction
            dout => dout_mar
        );
  -- used for store
  -- STORE R3 (R1) : DataMem[R1] = R3
  mdr: reg32bits PORT MAP (
            clk => clk,
        reset => reset,
            en => en_mdr,-- from control unit
        r_w => r_w_mdr,-- from control unit
            din => dout_y, -- from register
            dout => dout_mdr -- to memory
        );


end Behavioral;
```

Listing 2: Top Test Bench

```vhdl
-- Top Test Bench
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY top_tb IS
END top_tb;
```

```

ARCHITECTURE behavior OF top_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT top
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '1';
   constant clk_period : time := 10 ns;

BEGIN

  -- Instantiate the Unit Under Test (UUT)
   uut: top PORT MAP (
          clk => clk,
          reset => reset
        );

   -- 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
      -- hold reset state for 100 ns.
      wait for 100 ns;
    reset <= '0';
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
    wait for clk_period;
      wait;
   end process;

END;
```

Listing 3: Control Unit

```
--
   ----------------------------------------------------------------------------------
-- Create Date:    13:34:43 12/13/2014
-- Module Name:    control_unit - Behavioral
--
   ----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity control_unit is
port(
   clk, reset: IN STD_LOGIC;
   -- pc: program_counter
   crl_pc : OUT STD_LOGIC;  -- crl_pc = 1 =>  jump to the address
   en_pc: OUT STD_LOGIC; -- enable the PC -> always = 1

   -- irm: instruction_memory
   en_irm: OUT STD_LOGIC;   -- set when state == fetch

   -- to instruction decoder
   en_id: OUT STD_LOGIC; -- set when state == decode

   -- from instruction decoder
   opcode: IN  STD_LOGIC_VECTOR(2 DOWNTO 0);

   -- alu2: ALU
   alu_mux : OUT  std_logic;
   r_w_z : OUT  std_logic;
   en_z : OUT  std_logic;
     r_w_y : OUT  std_logic;
   en_y : OUT  std_logic;

   reg_mux: OUT  std_logic;
  -- this will affect program counter
  -- if set then ctl_pr = 1 and state == jump

    bneq_alu : IN STD_LOGIC;
   blt_alu : IN STD_LOGIC;

   --  to MAR
   en_mar: OUT STD_LOGIC; -- enable when load/store
   r_w_mar : OUT  std_logic;

   -- to MDR
   en_mdr: OUT STD_LOGIC; -- enable when store
   r_w_mdr : OUT  std_logic;
   -- to MEMORY
    r_w_mem: OUT STD_LOGIC; -- set when write | clear when read
    en_mem : OUT STD_LOGIC; -- enable when read | write

   -- to register files
   en_reg: OUT STD_LOGIC;
   r_w_reg  : OUT STD_LOGIC;
    sel_mux_addr1   : OUT std_logic;
         sel_mux_addr   : OUT  std_logic
 );

end control_unit;

architecture Behavioral of control_unit is

type state_type is (  reset_state,
      fetch,
      decode,
      load0, load1,
      store0, store1,
      alu0, alu1,
      halt,
      jump0, jump1
      );

signal state: state_type;
begin


 process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then state <= reset_state;
      else
        case state is
          when reset_state => state <= fetch;
        when fetch => state <= decode;
        when decode =>
          if  opcode = "000" or opcode = "001" then state <= alu0;
          elsif opcode = "100" or opcode = "101" then state <= jump0;
          elsif opcode = "010" then state <= load0;
          elsif opcode = "011" then state <= store0;
          elsif opcode = "110" then state <= halt;
          end if;
        when load0 =>   state <= load1;
        when load1 =>   state <= fetch;

        when store0 =>  state <= store1;
        when store1 =>  state <= fetch;

        when alu0 =>  state <= alu1;
        when alu1 =>  state <= fetch;

        when halt =>  state <= halt;

        when jump0 =>
          if blt_alu =  '1' or bneq_alu =  '1' then
            state <= jump1;
          else
            state <= fetch;
          end if;

        when jump1 =>    state <= fetch;
        when others =>   state <= halt;
        end case;
      end if;
    end if;
  end process;

  crl_pc    <= '1' when state = jump1
  else  '0';
  en_pc <= '1'     when state =  fetch
          else  '0';
  alu_mux <= '0'     when opcode = "001" -- for immediate
          else  '1';
  reg_mux <=  '1'     when state =  load1 and  opcode = "010"
          else   '0';
  en_irm <= '1'     when state =  fetch
          else  '0';
  en_id <= '1'     when state =  decode
          else  '0';
  en_reg <= '1'     when state =  decode or state =  load0 or state =  load1
              or state = store0 or state = store1
              or state = alu0  or state = alu1
              or state = jump0  or state = jump1
          else  '0';
  r_w_reg <= '1'     when state =  load1 or state = alu1
          else  '0';

  r_w_z <= '1' when  (state = alu0 and opcode = "000" )or (state = decode
   and opcode = "001" )
  else  '0';

  en_z <= '1'     when state = alu0 or state = alu1 or state = decode
          else  '0';
  r_w_y <= '1'     when state = decode
          else  '0';
  en_y <= '1'     when state = decode or state = alu0 or  state = store0
             or state = alu1  or state = jump0  or state = jump1
          else  '0';
  en_mar <= '1'   when state =   load0 or state = load1 or state = decode
             or state = store0 or state = store1
          else  '0';
  r_w_mar <= '1'     when state = store0 or state =   load0-- op 2
          else  '0';
  en_mdr <= '1'   when state =   load0 or state =   load1
             or state = store0 or state = store1
          else  '0';
  r_w_mdr <= '1'     when  state = store0
          else  '0';
  en_mem  <= '1'   when state =   load1 or  state = store1
          else  '0';
  r_w_mem <= '1'   when  state = store1
          else  '0';
  sel_mux_addr1  <= '0' when state = decode -- chose op 2
      else '1'; -- chose op 1

  sel_mux_addr  <=  '0' when  state = alu0 -- chose op3
      else  '1'; -- chose op1 or 2

end Behavioral;
```

Listing 4: ALU

```
--
    ----------------------------------------------------------------------------------
-- Create Date:    13:34:01 12/13/2014
-- Module Name:    ALU - Behavioral
-- Description:
-- ALU has two inputs data and one output data
-- one flag test output to test data it decide if
-- we should jumb to a certain instruction or not
--
    ----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity ALU is
port (
   clk, reset  : IN  std_logic;
  dinA, dinB  : IN STD_LOGIC_VECTOR(31 DOWNTO 0); -- operand 1 and 2
  opcode: IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- to decide the operation
  result: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  bneq, blt: OUT STD_LOGIC
);
end ALU;

architecture Behavioral of ALU is

begin
   process (dinA, dinB,  opcode)   is
  begin
    case  opcode  is
      when "000" =>
        result <= dinA xor dinB;
          bneq <= 'Z';
          blt <= 'Z';
      when "001" =>
        result <= dinA + dinB;
          bneq <= 'Z';
          blt <= 'Z';
      when "100" => --blt
         if  (dinA < dinB) then -- is A < B?
             blt <= '1';
          else
             blt <= '0';
        end if;
        bneq <= 'Z';
        result <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      when "101" => --bneq
        if ((dinA - dinB) = 0) then   -- is A==B?
            bneq <= '0';
          else
            bneq <= '1';
        end if;
        blt <= 'Z';
        result <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      when others =>
        bneq <= 'Z';
        blt <= 'Z';
        result <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    end case;

  end process;
end Behavioral;
```

Listing 5: Instruction Memory

```
--
   ------------------------------------------------------------------------------------
-- Create Date:    23:37:40 12/12/2014
-- Module Name:    instruction_memory - Behavioral
--
   ------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- for <conv_integer>
use IEEE.STD_LOGIC_ARITH.all;

entity instruction_memory is
port (
   clk,  en: IN STD_LOGIC;
   addr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
end instruction_memory;

architecture Behavioral of instruction_memory is

  type reg_array is array (0 to 15) of STD_LOGIC_VECTOR(15 downto 0);
  signal reg_file : reg_array;

begin
  reg_file(0) <= x"0481"; -- initialize
  reg_file(1) <= x"34D0"; -- initialize
  reg_file(2) <= x"38CC"; -- initialize
  reg_file(3) <= x"2884"; -- initialize
  reg_file(4) <= x"4C80"; -- initialize
  reg_file(5) <= x"5100"; -- initialize
  reg_file(6) <= x"8E09"; -- initialize
  reg_file(7) <= x"6D00"; -- initialize
  reg_file(8) <= x"7080"; -- initialize
  reg_file(9) <= x"2904"; -- initialize
  reg_file(10) <= x"AA84"; -- initialize
  reg_file(11) <= x"2484"; -- initialize
  reg_file(12) <= x"A703"; -- initialize
  reg_file(13) <= x"C000"; -- initialize
  reg_file(14) <= x"0000"; -- initialize
  reg_file(15) <= x"0000"; -- initialize

  dout <= reg_file(conv_integer(Unsigned(addr)))
  when  en = '1';
end Behavioral;
```

Listing 6: Instruction Decoder

```
--
    ----------------------------------------------------------------------------------

-- Create Date:    18:36:24 12/15/2014
-- Module Name:    instruction_decoder - Behavioral
-- Description:
-- take the instruction from the IR and decode the instruction then sends
    the
-- relevent data to ALU or control unit or Registers
--
    ----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity instruction_decoder is
port(
    clk, reset, en: IN STD_LOGIC;
     dbus : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    -- control flags from the decoder
    opcode: out  STD_LOGIC_VECTOR(2 DOWNTO 0); -- to alu
    op1, op2: out  STD_LOGIC_VECTOR(2 DOWNTO 0); -- to alu
    op3: out  STD_LOGIC_VECTOR(6 DOWNTO 0) -- to alu when xor and addi
    -- to memory when load or store
    );
end instruction_decoder;

architecture Behavioral of instruction_decoder is

signal IR: STD_LOGIC_VECTOR(15 downto 0);

begin
  process(clk) begin
    if clk'event and clk = '0' then
      if reset = '1' then
        IR <= x"0000";
      elsif en = '1' then
        IR <= dbus;
      end if;
    end if;
  end process;
     opcode <= IR(15 downto 13) ;
    op1 <= IR(12 downto 10);
    op2 <= IR(9 downto 7);
      op3 <= IR(6 downto 0);
end Behavioral;
```

Listing 7: Memory

```
--
    ----------------------------------------------------------------------------------------------
-- Create Date:    02:04:49 12/18/2014
-- Module Name:    memory - Behavioral
-- Description:
 -- 128B = 128 * 8 = 32 * 32
-- 2^5 = 32
--
    ----------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity memory is
port (
    clk, r_w, en, reset: in STD_LOGIC;
      addr: in STD_LOGIC_VECTOR(4 downto 0);
      din: in STD_LOGIC_VECTOR(31 downto 0);
    dout: out STD_LOGIC_VECTOR(31 downto 0)
);
end memory;
architecture Behavioral of memory is
type ram_typ is array(0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal ram: ram_typ;
begin
  process(reset, clk) begin
    if reset = '1' then
      for i in 0 to 31 loop
        ram(i) <= x"00000000";
      end loop;
  elsif  clk'event and clk = '0'   and r_w = '1' then
      ram(conv_integer(unsigned(addr))) <= din;
    end if;
  end process;
  dout <= ram(conv_integer(unsigned(addr)))
      when reset = '0' and en = '1' and r_w = '0' else
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end Behavioral;
```

Listing 8: Register File

```
--
    ----------------------------------------------------------------------------------------------
-- Create Date:    17:30:18 12/19/2014
-- Module Name:    Register file module - Behavioral
-- Description:   8 X 32 bits
```

```
--
    ----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- for <conv_integer>
use IEEE.std_logic_arith.all;


entity register_file is
port (
    -- r_w = 0 then read enable else write
    clk, reset, en, r_w : IN STD_LOGIC;
   -- for read operation
   dout : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    -- for write
    addr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
   din : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
);
end register_file;

architecture regArch of register_file is
  type reg_array is array (0 to 7) of std_logic_vector(31 downto 0);
  signal reg_file : reg_array;

begin
  process(clk, reset) begin
    if reset = '1' then
      for i in 0 to 7 loop
        reg_file(i) <= "00000000000000000000000000000000"; -- initialize
      end loop;
    elsif clk'event and clk = '0'  and r_w = '1' and en='1' then  -- write
    enabled
      reg_file(conv_integer(Unsigned(addr))) <= din;
    end if;
  end process;

  dout <= reg_file(conv_integer(Unsigned(addr)))
  when reset = '0' and en = '1' and r_w = '0' else -- read enabled
  "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";


end regArch;
```

Listing 9: Mux of 32 bits

```
--
    ----------------------------------------------------------------------------------
-- Create Date:    17:30:18 12/19/2014
-- Module Name:    mux - Behavioral
-- Description: Binary Mux, 32 bits wide
--
    ----------------------------------------------------------------------------------
```

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

entity mux is
port (
  inputA : in std_logic_vector (31 downto 0);
  inputB : in std_logic_vector (31 downto 0);
  sel : in std_logic;
  output : out std_logic_vector (31 downto 0)
);
end mux;

architecture Behavioral of mux is

begin
  process(sel, inputA, inputB) begin
    if sel='1' then
    output <= inputA;
  else
    output <=  inputB;
  end if;
  end process;

end Behavioral;
```

Listing 10: Mux of 3 bits

```
--
    ----------------------------------------------------------------------------------
-- Create Date:    17:30:18 12/19/2014
-- Module Name:    mux3b - Behavioral
-- Description: Binary Mux, 3 bits wide
--
    ----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity mux3b is
port (
  inputA : in std_logic_vector (2 downto 0);
  inputB : in std_logic_vector (2 downto 0);
  sel : in std_logic;
  output : out std_logic_vector (2 downto 0)
);
end mux3b;

architecture Behavioral of mux3b is

begin
  process(sel, inputA, inputB) begin
    if sel='1' then
    output <= inputA;
  else
    output <=  inputB;
  end if;
  end process;
end Behavioral;
```

Listing 11: Register of 32 bits

```
--
  ----------------------------------------------------------------------------------

-- Create Date:    13:39:20 12/13/2014
-- Module Name:    reg32bits - Behavioral
-- Description: Register of 32 bits wide
--
  ----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reg32bits is
port(
   clk, reset, en, r_w: IN STD_LOGIC;
   din : IN STD_LOGIC_VECTOR(31 DOWNTO 0); --from
    dout : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
   );
end reg32bits;

architecture Behavioral of reg32bits is
signal reg_file: std_logic_vector(31 downto 0);

begin
process(clk, reset) begin
    if reset = '1' then
        reg_file <= "00000000000000000000000000000000"; -- initialize
     elsif clk'event and clk = '1'  and r_w = '1' and en='1' then  -- write
     enabled
       reg_file  <= din;
    end if;
  end process;

  dout <= reg_file
  when reset = '0' and en = '1' and r_w = '0' else -- read enabled
  "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end Behavioral;
```

Listing 12: Memory Address Register

```
--
  ----------------------------------------------------------------------------------

-- Create Date:    13:38:42 12/13/2014
-- Module Name:    memory_address_register - Behavioral
-- Description:
-- LOAD R3 (R1) : R3 = DataMem[R1]
-- STORE R3 (R1) : DataMem[R1] = R3
```

```vhdl
-- so for MAR we will write the address (R1) in it
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity memory_address_register is
port(
   clk, reset, en, r_w: IN STD_LOGIC;
   din : IN STD_LOGIC_VECTOR(4 DOWNTO 0); --from
    dout : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
   );
end memory_address_register;

architecture Behavioral of memory_address_register is
signal reg_file: std_logic_vector(4 downto 0);

begin

process(clk, reset) begin
    if reset = '1' then
        reg_file <= "00000"; -- initialize
    elsif clk'event and clk = '1'  and r_w = '1' and en='1' then  -- write enabled
      reg_file  <= din;
    end if;
  end process;

  dout <= reg_file
  when reset = '0' and en = '1' and r_w = '0' else -- read enabled
  "ZZZZZ";

end Behavioral;
```