```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-------------------------- Declaring the entity of CONTROL BLOCK
--------------------------
entity control_block is
port (
  opcode :     in std_logic_vector(2 downto 0);    -- 3-bit input port for opcode to
use it for decoding the instruction register
  reset,clk:   in std_logic;                       -- clock and reset
  alu_op:      out std_logic;                      -- output logic control signal for
ALU operation
  mem_read1:   out std_logic;                      -- output logic control signal to
read first data from data memory
  mem_read2 :  out std_logic;                      -- output logic control signal to
read second data from data memory
  mem_write :  out std_logic;                      -- output logic control signal to
write data from accumulator to data memory
  Pc_inc :     out std_logic;                      -- output logic control signal to
increment program counter
  Load_address:out std_logic;                      -- output logic control signal to
load prgram memory address from program counter
  branch_eq :  out std_logic;                      -- output logic control signal to
activate branch operation
  Load_IR :    out std_logic;                      -- output logic control signal to
load instruction register from program memory
  load_output: out std_logic;                      -- output logic control signal to
send the the accumulator toward output port
  jal :        out std_logic;                      -- output logic control signal to
perform jump operation
  ex_data :    out std_logic;                      -- output logic control signal to
write/store external input to data memory
  inc_address: out std_logic;                      -- output logic controlsignal
input to increment the address of external data input
  acc_bus :    out std_logic);                     -- output logic control signal to
send the accumulator data to the Register A
end control_block;

architecture Behavioral of control_block is
-------------------- Declaring FSM --------------------------------------
type FSM is (S0_fetch,S1_fetch,S2_decode,
load_all,data,data2,output,beq,store,load_ext,load,jump); -------------- Declaring
the States
signal present_state,next_state: FSM;                      -------------- Declaring
Present and Next State of FSM
begin
------------------ Memory of State Machines -------------------------------
```

```
 process(clk,reset)
 begin
 if(reset = '1') then
   present_state <= S0_fetch;                          -------------- when active
reset is high, the present state will be assigned as S0_fetch
   elsif(rising_edge(clk)) then
   present_state <= next_state;                        -------------- when rising
edge of clock is detected, the present state will be assigned as mext_state
   end if;
  end process;
------------------ Next State Logic ---------------------------------------

process(opcode,present_state)
begin
 case present_state is
  when S0_fetch => next_state <=S1_fetch;
  when S1_fetch => next_state <=S2_decode;
  when S2_decode =>
  if(opcode = "000")then
  next_state<= load_all;
  elsif(opcode = "001")then
  next_state<= load_ext;
  elsif(opcode = "010")then
  next_state<= load;
  elsif(opcode = "011")then
  next_state<= store;
  elsif(opcode = "100")then
   next_state<= beq;
  elsif(opcode = "101")then
   next_state<= data;
  elsif(opcode = "110")then
   next_state<= jump;
  elsif(opcode = "111")then
   next_state<= output;
   else
   next_state<=S1_fetch;
   end if;

 when output    => next_state <=S0_fetch;
 when beq       => next_state <=S0_fetch;
 when data      => next_state <=data2;
 when data2     => next_state <=S0_fetch;
 when store     => next_state <=S0_fetch;
 when load      => next_state <=S0_fetch;
 when load_all  => next_state <=S0_fetch;
 when load_ext  => next_state <=S0_fetch;
 when jump      => next_state <=S0_fetch;
 when others    => next_state <=S0_fetch;
 end case;
 end process;

 ---------------------------------------------------------------------------
 -------------------- Output Logic Circuit --------------------------------
 process(present_state)
  begin
 case present_state is
   when S0_fetch  =>  -- load prgoramm memory address from program counter and
increment the program counter
     Pc_inc <= '1';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '1';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when S1_fetch  => -- load instruction register from instruction memory after
loading programm address
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '1';
     jal<= '0';
     inc_address<= '0';
    when S2_decode  => -- decode the instruction register
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
    when load_all => --  load two data from data memory toward the two registers A 
and B
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '1';
     mem_read2 <= '1';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
    when load_ext => -- store the external input in the data memory
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus <= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '1';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when load => -- load data from data memory to the register B
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '1';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when store => -- store the accumlator data in the data memory
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '1';
     acc_bus <= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '1';
   when beq => -- activate branch operations
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='1';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when output =>-- send the accumalator data to the output port
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='1';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
  when data2 =>  -- load the data from accumalator to the A register
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus <='1';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when data => -- allow the ALU circuit to peform logical or arithmetic operation
     Pc_inc <= '0';
     alu_op <= '1';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus <='0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
   when jump => -- execute the jump instruction
     Pc_inc <= '0';
     alu_op <= '0';
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus <= '1';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '1';
     inc_address<= '0';
  when others => -- set all the output logic to zero when the opcode input is unknown
     Pc_inc <= '0';
     alu_op <= '0';
```

```
     mem_read1 <= '0';
     mem_read2 <= '0';
     mem_write <= '0';
     acc_bus<= '0';
     branch_eq <='0';
     load_output <='0';
     ex_data <= '0';
     Load_address <= '0';
     Load_IR <= '0';
     jal<= '0';
     inc_address<= '0';
  end case;
end process;
end Behavioral;
```