1. (20 points) Consider the program on page 1-21 and 1-22 of the lecture notes. Suppose the simple processor is augmented with a direct-mapped instruction cache with 16 words. Each cache word includes a tag, consisting of the upper 12 bits of the stored instruction’s address and the instruction itself. Assume that all words in the cache are zero at the time the program starts execution. Show the contents of the cache the first time that the instruction at location 000a is executed. Show the contents of the cache the second time that the instruction at location 000a is executed. Show the contents of the cache the third time that the instruction at location 000a is executed.

The cache contents are shown below, right after the instruction at location 000a is executed.

<table>
<thead>
<tr>
<th>tag</th>
<th>instr.</th>
<th>tag</th>
<th>instr.</th>
<th>tag</th>
<th>instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
<td>001</td>
<td>1</td>
<td>86d0</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
<td>001</td>
<td>2</td>
<td>1006</td>
</tr>
<tr>
<td>2</td>
<td>000</td>
<td>0</td>
<td>001</td>
<td>3</td>
<td>6005</td>
</tr>
<tr>
<td>3</td>
<td>000</td>
<td>4</td>
<td>001</td>
<td>4</td>
<td>2006</td>
</tr>
<tr>
<td>4</td>
<td>000</td>
<td>5</td>
<td>001</td>
<td>5</td>
<td>2004</td>
</tr>
<tr>
<td>5</td>
<td>000</td>
<td>6</td>
<td>001</td>
<td>6</td>
<td>1004</td>
</tr>
<tr>
<td>6</td>
<td>000</td>
<td>7</td>
<td>001</td>
<td>7</td>
<td>8001</td>
</tr>
<tr>
<td>7</td>
<td>000</td>
<td>8</td>
<td>001</td>
<td>8</td>
<td>2005</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
<td>9</td>
<td>001</td>
<td>9</td>
<td>1005</td>
</tr>
<tr>
<td>9</td>
<td>000</td>
<td>a</td>
<td>000</td>
<td>a</td>
<td>000</td>
</tr>
<tr>
<td>a</td>
<td>000</td>
<td>b</td>
<td>000</td>
<td>b</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>000</td>
<td>c</td>
<td>000</td>
<td>c</td>
<td>000</td>
</tr>
<tr>
<td>c</td>
<td>000</td>
<td>d</td>
<td>000</td>
<td>d</td>
<td>000</td>
</tr>
<tr>
<td>d</td>
<td>000</td>
<td>e</td>
<td>000</td>
<td>e</td>
<td>000</td>
</tr>
<tr>
<td>e</td>
<td>000</td>
<td>f</td>
<td>000</td>
<td>f</td>
<td>004</td>
</tr>
<tr>
<td>f</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose that an instruction fetch takes 2 clock cycles for instructions that are present in the cache, but 10 clock cycles for instructions that are not in the cache. How much time is spent on instruction fetches during the first execution of the loop in the program on 1-21
and 1-22? How much time is spent on instruction fetches during the second execution of the loop? Explain your answers.

During the first execution of the loop, each of the instructions has to be retrieved from memory and placed in the cache, so the time spent on instruction fetches during the first execution of the loop is 170 clock cycles, since there are 17 instructions in the loop. During the second execution of the loop, most instructions are in the cache. The only exceptions are the instructions at locations 000a and 001a. So the time spent on instruction fetches during the second execution of the loop is $15 \times 2 + 2 \times 10 = 50$, or less than one third the time spent during the first execution.

(30 points) This problem involves extending the simple processor presented in class. In particular, you are to add a new conditional branch instruction that branches to a specified location when the value in the accumulator is greater than or equal to zero. Use B as the operation code for your new instruction. So, an instruction of the form $Bxxx$ will cause the program counter value to change to $xxx$ if the ACC is greater than or equal to zero, otherwise it will increment to the next instruction, as it normally does.

Turn in a copy of the VHDL code, highlighting all lines that you changed, and including a brief comment explaining what each added or modified code segment does. Simulate your design, by pre-loading the memory with a small test program that exercises the new instruction, showing both when it branches and when it does not branch. Turn in simulation output that shows the values of all the processor registers and the two buses, during the execution of your test program. On the simulation output, highlight the places where your new instruction is used.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity program_counter is
  port (                     
    clk, en_A, ld, inc, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0)
  );
end program_counter;

architecture pcArch of program_counter is
  signal pcReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        pcReg <= "0000000000000000";
      elsif ld = '1' then
        pcReg <= dBus;
      elsif inc = '1' then
        pcReg <= pcReg + "0000000000000001";
      else
        pcReg <= pcReg + "0000000000000000";
      end if;
    end if;
  end process;
end pcArch;
```
library IEEE;
use IEEE.std_logic_1164.all;

entity instruction_register is
port (  
clk, en_A, en_D, ld, reset: in STD_LOGIC;
abus: out STD_LOGIC_VECTOR(15 downto 0);
dbus: inout STD_LOGIC_VECTOR(15 downto 0);
load, store, add, neg, halt: out STD_LOGIC;
branch, cbranch, iload: out STD_LOGIC;
istore, dload, dadd, cbranchB: out STD_LOGIC
);
end instruction_register;

architecture irArch of instruction_register is
begin
process(clk) begin
if clk'event and clk = '0' then
if reset = '1' then
irReg <= "0000000000000000";
elsif ld = '1' then
irReg <= dbus;
end if;
end if;
end process;
abus <= "0000" & irReg(11 downto 0) when en_A = '1' else "ZZZZZZZZZZZZZZZZZ";
dbus <= "0000" & irReg(11 downto 0) when en_D = '1' else "ZZZZZZZZZZZZZZZZZ";

load <= '1' when irReg(15 downto 12) = "0000" else '0';
store <= '1' when irReg(15 downto 12) = "0001" else '0';
add <= '1' when irReg(15 downto 12) = "0010" else '0';
neg <= '1' when irReg = "0011" & "000000000000" else '0';
halt <= '1' when irReg = "0011" & "000000000001" else '0';
branch <= '1' when irReg(15 downto 12) = "0100" else '0';
cbranch <= '1' when irReg(15 downto 12) = "0101" else '0';
iload <= '1' when irReg(15 downto 12) = "0110" else '0';
istore <= '1' when irReg(15 downto 12) = "0111" else '0';
dload <= '1' when irReg(15 downto 12) = "1000" else '0';
dadd <= '1' when irReg(15 downto 12) = "1001" else '0';
cbranchB <= '1' when irReg(15 downto 12) = "1011" else '0';
end architecture;
library IEEE;
use IEEE.std_logic_1164.all;

entity indirect_addr_register is
port ( 
  clk, en_A, ld, reset: in STD_LOGIC;
  aBus: out STD_LOGIC_VECTOR(15 downto 0);
  dBus: in STD_LOGIC_VECTOR(15 downto 0)
);
end indirect_addr_register;

architecture iarArch of indirect_addr_register is
signal iarReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        iarReg <= "0000000000000000";
      elsif ld = '1' then
        iarReg <= dBus;
      end if;
    end if;
  end process;
  aBus <= iarReg when en_A = '1' else "ZZZZZZZZZZZZZZZZ";
end iarArch;

---------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity accumulator is
port ( 
  clk, en_D, ld, selAlu, reset: in STD_LOGIC;
  aluD: in STD_LOGIC_VECTOR(15 downto 0);
  dBus: inout STD_LOGIC_VECTOR(15 downto 0);
  q: out STD_LOGIC_VECTOR(15 downto 0)
);
end accumulator;

architecture accArch of accumulator is
signal accReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        accReg <= "0000000000000000";
      elsif ld = '1' and selAlu = '1' then
        accReg <= dBus;
      end if;
    end if;
  end process;
end accArch;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity alu is
  port (op: in STD_LOGIC_VECTOR(1 downto 0);
         accD: in STD_LOGIC_VECTOR(15 downto 0);
         dBus: in STD_LOGIC_VECTOR(15 downto 0);
         result: out STD_LOGIC_VECTOR(15 downto 0);
         accZ, accN: out STD_LOGIC
  );
end alu;

architecture aluArch of alu is
begin
  result <= (not accD) + "0000000000000001" when op = "00" else
            accD + dBus when op = "01" else
            "0000000000000000";
  accZ <= not (accD(0) or accD(1) or accD(2) or accD(3) or
              accD(4) or accD(5) or accD(6) or accD(7) or
              accD(8) or accD(9) or accD(10) or accD(11) or
              accD(12) or accD(13) or accD(14) or accD(15)
  );
  accN <= accD(15);
end aluArch;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity ram is
  port (r_w, en, reset: in STD_LOGIC;
        aBus: in STD_LOGIC_VECTOR(15 downto 0);
        dBus: inout STD_LOGIC_VECTOR(15 downto 0)
  );
end ram;
architecture ramArch of ram is

    type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
    signal ram: ram_typ;

    begin
        process(reset, r_w) begin
            if reset = '1' then
                ram(0) <= x"8001";
                ram(1) <= x"b003";
                ram(2) <= x"3001";
                ram(3) <= x"3000";
                ram(4) <= x"b006";
                ram(5) <= x"3001";
                ram(6) <= x"3001";
                for i in 7 to 63 loop
                    ram(i) <= x"0000";
                end loop;
            elsif r_w'event and r_w = '0' then
                ram(conv_integer(unsigned(aBus))) <= dBus;
            end if;
        end process;
    end ramArch;

library IEEE;
use IEEE.std_logic_1164.all;

entity controller is
    port (  
        clk, reset: in  STD_LOGIC;
        mem_enD, mem_rw: out STD_LOGIC;
        pc_enA, pc_ld, pc_inc: out STD_LOGIC;
        ir_enA, ir_enD, ir_ld: out STD_LOGIC;
        ir_load, ir_store, ir_add: in  STD_LOGIC;
        ir_neg, ir_halt, ir_branch: in  STD_LOGIC;
        ir_cbranch, ir_iLoad: in  STD_LOGIC;
        ir_istore, ir_dload, ir_dadd: in  STD_LOGIC;
        iar_enA, iar_ld: out STD_LOGIC;
        acc_enD, acc_ld, acc_selAlu: out STD_LOGIC;
        alu_accZ, alu_accN: in  STD_LOGIC;
        alu_op: in  STD_LOGIC_VECTOR(1 downto 0);
    );
end controller;

architecture controllerArch of controller is
    type state_type is ( reset_state, fetch0, fetch1,
load0, load1,
store0, store1,
add0, add1,
negate0, negate1,
halt,
branch0, branch1,
cbranch0, cbranch1,
iload0, iload1, iload2, iload3,
istore0, istore1, istore2, istore3,
dload0, dload1,
dadd0, dadd1,
cbranchB0, cbranchB1
);

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 <= fetch0;
          when fetch0 => state <= fetch1;
          when fetch1 =>
            if ir_load = '1' then state <= load0;
            elsif ir_store = '1' then state <= store0;
            elsif ir_add = '1' then state <= add0;
            elsif ir_neg = '1' then state <= negate0;
            elsif ir_halt = '1' then state <= halt;
            elsif ir_branch = '1' then state <= branch0;
            elsif ir_cbranch = '1' then state <= cbranch0;
            elsif ir_iload = '1' then state <= iload0;
            elsif ir_istore = '1' then state <= istore0;
            elsif ir_dload = '1' then state <= dload0;
            elsif ir_dadd = '1' then state <= dadd0;
            elsif ir_cbranchB = '1' then
              state <= cbranchB0;
          end if;
          when load0 => state <= load1;
          when load1 => state <= fetch0;
          when store0 => state <= store1;
          when store1 => state <= fetch0;
          when add0 => state <= add1;
          when add1 => state <= fetch0;
          when negate0 => state <= negate1;
          when negate1 => state <= fetch0;
          when halt => state <= halt;
        end case;
      end if;
    end if;
  end process;
end;
when branch0 => state <= branch1;
when branch1 => state <= fetch0;
when cbranch0 => state <= cbranch1;
when cbranch1 => state <= fetch0;
when iload0 => state <= iload1;
when iload1 => state <= iload2;
when iload2 => state <= iload3;
when iload3 => state <= fetch0;
when istore0 => state <= istore1;
when istore1 => state <= istore2;
when istore2 => state <= istore3;
when istore3 => state <= fetch0;
when dload0 => state <= dload1;
when dload1 => state <= fetch0;
when dadd0 => state <= dadd1;
when dadd1 => state <= fetch0;
when cbranchB0 => state <= cbranchB1;
when cbranchB1 => state <= fetch0;
when others => state <= halt;
end case;
end if;
end if;
end process;
process(clk) begin -- special process for memory write timing
if clk'event and clk = '0' then
if state = store0 or state = istore2 then
   mem_rw <= '0';
else
   mem_rw <= '1';
end if;
end if;
end process;
mem_enD <= '1' when state = fetch0 or state = fetch1 or state = load0 or state = load1 or state = add0 or state = add1 or state = iload0 or state = iload1 or state = iload2 or state = iload3 or state = istore0 or state = istore1 else '0';
pc_enA <= '1' when state = fetch0 or state = fetch1 else '0';
pc_ld <= '1' when state = branch0 or (state=cbranch0 and alu_accZ = '1') or (state = cbranchB0 and alu_accN = '0') else '0';
pc_inc <= '1' when state = fetch1
load PC when branch condition holds

enable ir during branch

globally visible copy of ACC<0 signal
component program_counter
port (
    clk, en_A, ld, inc, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0)
);end component;

component instruction_register
port (
    clk, en_A, en_D, ld, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0);
    load, store, add, neg, halt, branch: out STD_LOGIC;
    cbranch, iload, istore, dload, dadd: out STD_LOGIC;
    cbranchB: out STD_LOGIC
);end component;

component indirect_addr_register
port (
    clk, en_A, ld, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0)
);end component;

component accumulator
port (
    clk, en_D, ld, selAlu, reset: in STD_LOGIC;
    aluD: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0);
    q: out STD_LOGIC_VECTOR(15 downto 0)
);end component;

component alu
port (
    op: in STD_LOGIC_VECTOR(1 downto 0);
    accD: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0);
    result: out STD_LOGIC_VECTOR(15 downto 0);
    accZ, accN: out STD_LOGIC
);end component;

component ram
port (
    r_w, en, reset: in STD_LOGIC;
    aBus: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0)
);end component;
component controller
port (
  clk, reset: in STD_LOGIC;
  mem_enD, mem_rw: out STD_LOGIC;
  pc_enA, pc_ld, pc_inc: out STD_LOGIC;
  ir_enA, ir_enD, ir_ld: out STD_LOGIC;
  ir_load, ir_store, ir_add: in STD_LOGIC;
  ir_neg, ir_halt, ir_branch: in STD_LOGIC;
  ir_cbranch, ir_iload: in STD_LOGIC;
  ir_istore, ir_dload, ir_dadd: in STD_LOGIC;
  ir_cbranchB: in STD_LOGIC;
  iar_enA, iar_ld: out STD_LOGIC;
  acc_enD, acc_ld, acc_selAlu: out STD_LOGIC;
  alu_accZ, alu_accN: in STD_LOGIC;
  alu_op: out STD_LOGIC_VECTOR(1 downto 0)
);
end component;

signal abus: STD_LOGIC_VECTOR(15 downto 0);
signal dbus: STD_LOGIC_VECTOR(15 downto 0);
signal mem_enD, mem_rw: STD_LOGIC;
signal pc_enA, pc_ld, pc_inc: STD_LOGIC;
signal ir_enA, ir_enD, ir_ld: STD_LOGIC;
signal ir_load, ir_store, ir_add: STD_LOGIC;
signal ir_neg, ir_halt, ir_branch: STD_LOGIC;
signal ir_cbranch, ir_iload, ir_istore: STD_LOGIC;
signal ir_dload, ir_dadd, ir_cbranchB: STD_LOGIC;
signal iar_enA, iar_ld: STD_LOGIC;
signal acc_enD, acc_ld, acc_selAlu: STD_LOGIC;
signal acc_Q: STD_LOGIC_VECTOR(15 downto 0);
signal alu_op: STD_LOGIC_VECTOR(1 downto 0);
signal alu_accZ, alu_accN: STD_LOGIC;
signal alu_result: STD_LOGIC_VECTOR(15 downto 0);

begin
  pc: program_counter port map(clk, pc_enA, pc_ld, pc_inc, reset, abus, dbus);
  ir: instruction_register port map(clk, ir_enA, ir_enD, ir_ld, reset, abus, dbus,
                                   ir_load, ir_store, ir_add,
                                   ir_neg, ir_halt, ir_branch,
                                   ir_cbranch, ir_iload, ir_istore,
                                   ir_dload, ir_dadd, ir_cbranchB);
  iar: indirect_addr_register port map(clk, iar_enA, iar_ld, reset,
                                       abus, dbus);
  acc: accumulator port map(clk, acc_enD, acc_ld, acc_selAlu, reset,
                             alu_result, dbus, acc_Q);
connect ir to controller
alu: alu port map(alu_op, acc_Q, dbus, alu_result, alu_accZ, alu_accN);

mem: ram port map(mem_rw, mem_enD, reset, abus, dbus);

ctl: controller port map(clk, reset, mem_enD, mem_rw, pc_enA, pc_ld, pc_inc, ir_enA, ir_enD, ir_ld, ir_load, ir_store, ir_add, ir_negate, ir_halt, ir_branch, ir_cbranch, ir_iload, ir_istore, ir_dload, ir_dadd, ir_cbranchB, iar_enA, iar_ld, acc_enD, acc_ld, acc_selAlu, alu_accZ, alu_accN, alu_op);

abusX <= abus;
dbusX <= dbus;
mem_enDX <= mem_enD;
mem_rwX <= mem_rw;
pc_enAX <= pc_enA;
pc_ldX <= pc_ld;
pc_incX <= pc_inc;
ir_enAX <= ir_enA;
ir_enDX <= ir_enD;
ir_ldX <= ir_ld;
 iar_enAX <= iar_enA;
 iar_ldX <= iar_ld;
 acc_enDX <= acc_enD;
 acc_ldX <= acc_ld;
 acc_selAlux <= acc_selAlu;
 acc_QX <= acc_Q;
 alu_opX <= alu_op;
 alu_accZX <= alu_accZ;
 alu_accNX <= alu_accN;

end topArch;