1. (10 points) A 64Kx16 SRAM chip uses coincident decoding by splitting the internal decoder into a row decoder and a column decoder. (a) Assuming that the RAM cell array is square, what is the size of each decoder, and how many AND gates are required for decoding an address?

The memory has 1 Mbits or $2^{20}$ bits, so it will have 1024 rows and columns. This means that the row decoder will require 10 bits. That is, it’s a 10-to-1024 decoder. Since the memory has 64K=$2^{16}$ words, it will need 16 address bits overall, so the column decoder will use 6 bits and be a 6-to-64 decoder (of course there will be 16 of these decoders). The column decoder has 1024 outputs, so can be implemented with 1024 AND gates with 10 inputs each. The column decoders are 6-to-64 decoders, so these will each have 64 AND gates, but there 16 for the input and also 16 multiplexors for the outputs, so it will be 1024+1024=2048 AND gates for the column decoding.

(b) Determine the row and column selection lines that are enabled when the input address is the binary equivalent of 37650$_{10}$. Assume that the high order address bits go to the row decoder and that the low order address bits go to the column decoder.

37650 is x9312 in hex, so the first ten bits are 1001 0011 00 or x24c, so row x24c is the row decoder output that is enabled. The column lines that are enabled are x012, x032, x052, x072, x092, x0b2, x0d2, x0f2, x112, x132, x152, x172, x19x, x1b2, x1d2, x1f2.

2. (12 points) Suppose we want to implement the following logic equations.

\[
X = AB' + C(B+D) \quad Y = (A' + B)(AC + C'D) \quad Z = A \oplus B + CD'
\]

(a) If these equations are implemented with a PROM, how many words must the PROM have and how many bits per word must it have?

Since there are 4 input signals, it must have 16 words.

(b) If these equations are implemented with a PLA, how many product terms must it have?

The number of product terms needed to express these equations are 3, 4 and 3, assuming that they are implemented directly, without any simplification. However, the term AB' appears in both the first and third equations, and AA'C can be dropped from the second equation, so 7 terms are enough.
(c) If the equations are implemented with a PAL, how many product terms must it have?

*The PAL must have 3 outputs and each of the outputs requires three terms, so the PAL as a whole must have 9.*

3. (80 points) This problem involves redesigning the basic processor to equip it with a set of four registers, as illustrated on page 8-3 of the notes. Design a register file block to replace the Accumulator block. The register file will have the control inputs shown on page 8-3, and described in more detail below.

| **EnD** | enable output onto the data bus |
| **LD**  | load a register from the data bus or from the ALU |
| **Target** | 2 bit value specifying register to be loaded |
| **srcA** | 2 bit value specifying which register should be connected to the A bus going to the ALU |
| **srcB** | 2 bit value specifying which register should be connected to the B bus going to the ALU |
| **srcD** | 2 bit value specifying which register output should be connected to the data bus |
| **selALU** | when asserted, the register to be loaded (if any) is loaded from the R bus coming from the ALU, otherwise, it is loaded from the data bus |

Design a replacement for the ALU that implements the following operations:

\[
R <= A, R <= -A, R <= A + B, R <= A - B, R <= A \ll B, R <= A \gg B
\]

where \(\ll\) is the shift left operation and \(\gg\) is the shift right operation. Your ALU should also provide the following status signals.

\[
A = 0, A < 0
\]

Modify the instruction register block and the controller, so that the processor implements the following set of instructions (these replace the original instruction set). In the list below, the instructions are shown in binary format.

| 00aa rrdd dddd dddd | Load the register specified by rr. The aa bits specify the addressing mode: 00 for immediate, 01 for direct and 10 for indirect. For immediate addressing, the register is loaded with the value specified by the d bits. For direct addressing, it is loaded from the location specified by the d bits. For indirect addressing, it is loaded from the address stored in the location specified by the d bits. |
Store the value in the register specified by rr. The aa bits specify the addressing mode: 01 for direct, 10 for indirect.

Copy the value in register aa to register rr.

Add the values in registers aa and bb and put the result in register rr.

Subtract the value in register bb from the value in register aa and put the result in rr.

Negate the value in register aa and put the result in rr.

Shift the value in register aa to the left by the number of positions specified by register bb and put the result in rr.

Shift the value in register aa to the right by the number of positions specified by register bb and put the result in rr.

Halt

Branch if the value in register aa is zero to the location specified by the t bits.

Branch if the value in register aa is less than zero to the location specified by the t bits.

Branch if the value in register aa is greater than zero to the location specified by the t bits.

Branch to the location specified by the t bits.

Use the same timing as for the basic processor. So, all instructions except the indirect load and store instructions use 4 clock ticks (including the instruction fetch) while the indirect load and store instructions use 6. Create a timing diagram for each instruction, like the ones on pages 7-26 and 7-27.

Demonstrate your processor by running the following test program. Turn in a copy of your source code and the simulation output from a functional simulation of the processor running the test program. Your simulation output should show the contents of all registers (including the four in the register file) as well as what’s on the address and data buses.

<table>
<thead>
<tr>
<th>addr.</th>
<th>instr.</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0821</td>
<td>Load “21” into r2</td>
</tr>
<tr>
<td>0001</td>
<td>5820</td>
<td>Store r2 to 20</td>
</tr>
<tr>
<td>0002</td>
<td>0001</td>
<td>Load “1” into r0</td>
</tr>
<tr>
<td>0003</td>
<td>8560</td>
<td>r1 = r2 &gt;&gt; r0</td>
</tr>
<tr>
<td>0004</td>
<td>6420</td>
<td>Store r1 to *20</td>
</tr>
<tr>
<td>0005</td>
<td>84c0</td>
<td>r3 = r0 &lt;&lt; r0</td>
</tr>
<tr>
<td>0006</td>
<td>81f4</td>
<td>r3 = r3 + r1</td>
</tr>
<tr>
<td>0007</td>
<td>82f0</td>
<td>r3 = r3 - r0</td>
</tr>
<tr>
<td>0008</td>
<td>8350</td>
<td>r1 = -r1</td>
</tr>
<tr>
<td>0009</td>
<td>e40d</td>
<td>if r1 &gt; 0 goto 000d</td>
</tr>
<tr>
<td>000a</td>
<td>e80c</td>
<td>if r2 &gt; 0 goto 000c</td>
</tr>
<tr>
<td>000b</td>
<td>f00d</td>
<td>goto 000d</td>
</tr>
<tr>
<td>000c</td>
<td>f008</td>
<td>goto 0008</td>
</tr>
<tr>
<td>000d</td>
<td>8f00</td>
<td>halt</td>
</tr>
</tbody>
</table>
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 <= x"0000";
            elsif ld = '1' then
                pcReg <= dBus;
            elsif inc = '1' then
                pcReg <= pcReg + x"0001";
            end if;
        end if;
    end process;
    aBus <= pcReg when en_A = '1' else "ZZZZZZZZZZZZZZZZZ";
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, copy, add, sub, negate: out STD_LOGIC;
    shiftL, shiftR, halt, brZero, brLess: out STD_LOGIC;
    brGtr, branch: out STD_LOGIC;
    treg, areg, breg, amode: out STD_LOGIC_VECTOR(1 downto 0);
    irRegX: out STD_LOGIC_VECTOR(15 downto 0)
);
end instruction_register;

architecture irArch of instruction_register is
signal irReg: STD_LOGIC_VECTOR(15 downto 0);
begin
    process(clk) begin
        if clk'event and clk = '0' then
if reset = '1' then
    irReg <= x"0000";
elsif ld = '1' then
    irReg <= dBus;
end if;
end if;
end process;

aBus <= "000000" & irReg(9 downto 0) when en_A = '1' else
       "ZZZZZZZZZZZZZZ";

load    <= '1' when irReg(15 downto 14) = "00"  else '0';
store   <= '1' when irReg(15 downto 14) = "01"  else '0';
copy    <= '1' when irReg(15 downto 8)  = x"80"  else '0';
add     <= '1' when irReg(15 downto 8)  = x"81"  else '0';
sub     <= '1' when irReg(15 downto 8)  = x"82"  else '0';
negate  <= '1' when irReg(15 downto 8)  = x"83"  else '0';
shiftL  <= '1' when irReg(15 downto 8)  = x"84"  else '0';
shiftR  <= '1' when irReg(15 downto 8)  = x"85"  else '0';
halt    <= '1' when irReg(15 downto 8)  = x"8f"  else '0';
brZero  <= '1' when irReg(15 downto 12) = x"c"  else '0';
brLess  <= '1' when irReg(15 downto 12) = x"d"  else '0';
brGtr   <= '1' when irReg(15 downto 12) = x"e"  else '0';
brn     <= '1' when irReg(15 downto 12) = x"f"  else '0';
treg   <= irReg(11 downto 10) when irReg(15) = '0' else
               irReg( 7 downto  6) when irReg(15 downto 12) = x"8" else
               "00";
areg   <= irReg( 5 downto  4) when irReg(15 downto 14) = "10" else
               irReg(11 downto 10) when irReg(15 downto 14) = "11" else
               "00";
breg   <= irReg( 3 downto  2) when irReg(15 downto 14) = "10" else
               "00";
amode  <= irReg(13 downto 12);
end irArch;
---------------------------------------------------------

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 <= x"0000";
    elsif ld = '1' then  
      iarReg <= dBus;
    end if;  
  end if;  
end process;  

aBus <= iarReg when en_A = '1' else  
  "ZZZZZZZZZZZZZZZZZ";
end iarArch;

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

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

entity regFile is  
  port (  
    clk, en_D, ld, selALU, reset: in STD_LOGIC;
    target, srcA, srcB, srcD: in STD_LOGIC_VECTOR(1 downto 0);
    ALUbusR: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0);
    ALUbusA, ALUbusB: out STD_LOGIC_VECTOR(15 downto 0);
    r0, r1, r2, r3: out STD_LOGIC_VECTOR(15 downto 0)
  );
end regFile;

architecture regFileArch of regFile is  
  type regFile_typ is array(0 to 3) of STD_LOGIC_VECTOR(15 downto 0);
  signal reg: regFile_typ;
begin  
  process begin  
    wait until clk = '1';  
    if reset = '1' then  
      for i in 0 to 3 loop  
        reg(i) <= x"0000";
      end loop;
    elsif ld = '1' and selALU = '1' then  
      reg(conv_integer(unsigned(target))) <= ALUbusR;
    elsif ld = '1' and selALU = '0' then  
      reg(conv_integer(unsigned(target))) <= dBus;
    end if;  
  end process;  
  dBus <= reg(conv_integer(unsigned(srcD))) when en_D = '1' else  
    "ZZZZZZZZZZZZZZZZZ";
  ALUbusA <= reg(conv_integer(unsigned(srcA)));  
  ALUbusB <= reg(conv_integer(unsigned(srcB)));  
  r0 <= reg(0);
  r1 <= reg(1);
  r2 <= reg(2);  
end regFileArch;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity alu is
    port (op: in STD_LOGIC_VECTOR(2 downto 0);
             rf_BusA, rf_BusB: in STD_LOGIC_VECTOR(15 downto 0);
             dBus: in STD_LOGIC_VECTOR(15 downto 0);
             rf_BusR: out STD_LOGIC_VECTOR(15 downto 0);
             regZ, regLZ: out STD_LOGIC);
end alu;

architecture aluArch of alu is
begin
    zeros <= x"0000" when op = "000" else
             (not rf_BusA) + '1' when op = "001" else
             rf_BusA + rf_BusB when op = "010" else
             rf_BusA - rf_BusB when op = "011" else
             rf_BusA when op = "100" and rf_BusB=x"0000" else
             rf_BusA(14 downto 0) & "0" when op = "100" and rf_BusB=x"0001" else
             rf_BusA(13 downto 0) & "00" when op = "100" and rf_BusB=x"0002" else
             rf_BusA(12 downto 0) & "000" when op = "100" and rf_BusB=x"0003" else
             rf_BusA(11 downto 0) & x"0" when op = "100" and rf_BusB=x"0004" else
             rf_BusA(10 downto 0) & x"0" & "0" when op = "100" and rf_BusB=x"0005" else
             rf_BusA( 9 downto 0) & x"0" & "00" when op = "100" and rf_BusB=x"0006" else
             rf_BusA( 8 downto 0) & x"0" & "000" when op = "100" and rf_BusB=x"0007" else
             rf_BusA( 7 downto 0) & x"00" when op = "100" and rf_BusB=x"0008" else
             rf_BusA( 6 downto 0) & x"00" & "0" when op = "100" and rf_BusB=x"0009" else
             rf_BusA( 5 downto 0) & x"00" & "00" when op = "100" and rf_BusB=x"000a" else
             rf_BusA( 4 downto 0) & x"00" & "000" when op = "100" and rf_BusB=x"000b" else
             rf_BusA( 3 downto 0) & x"000" when op = "100" and rf_BusB=x"000c" else
             rf_BusA( 2 downto 0) & x"000" & "0" when op = "100" and rf_BusB=x"000d" else
             rf_BusA( 1 downto 0) & x"000" & "00" when op = "100" and rf_BusB=x"000e" else
             rf_BusA( 0 downto 0) & x"000" & "000" when op = "100" and rf_BusB=x"000f" else
             "0" & rf_BusA(15 downto 1) when op = "101" and rf_BusB=x"0001" else
             "00" & rf_BusA(15 downto 2) when op = "101" and rf_BusB=x"0002" else
             "000" & rf_BusA(15 downto 3) when op = "101" and rf_BusB=x"0003" else
             x"0" & rf_BusA(15 downto 4) when op = "101" and rf_BusB=x"0004" else
             x"0" & "0" & rf_BusA(15 downto 5) when op = "101" and rf_BusB=x"0005" else
             x"0" & "00" & rf_BusA(15 downto 6) when op = "101" and rf_BusB=x"0006" else
             x"0" & "000" & rf_BusA(15 downto 7) when op = "101" and rf_BusB=x"0007" else
             x"00" & rf_BusA(15 downto 8) when op = "101" and rf_BusB=x"0008" else
             x"00" & "0" & rf_BusA(15 downto 9) when op = "101" and rf_BusB=x"0009" else
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 31) of STD_LOGIC_VECTOR(15 downto 0);   
signal ram: ram_typ;
begin
  process(reset, r_w) begin
    if reset = '1' then
      ram(0) <= x"0821";
      ram(1) <= x"5820";
      ram(2) <= x"0001";
      ram(3) <= x"8560";
      ram(4) <= x"6420";
      ram(5) <= x"84c0";
      ram(6) <= x"81f4";
      ram(7) <= x"82f0";
      ram(8) <= x"8350";
      ram(9) <= x"e40d";
      ram(10) <= x"e80c";
      ram(11) <= x"f00d";
      ram(12) <= x"f008";
      ram(13) <= x"8f00";
    elsif r_w'event and r_w = '0' then
      ram(conv_integer(unsigned(aBus))) <= dBus;
    end if;
  end process;
  dBus <= ram(conv_integer(unsigned(aBus)));
    when reset = '0' and en = '1' and r_w = '1' else
      "ZZZZZZZZZZZZZZZZZ";
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_copy:  in STD_LOGIC;
    ir_add, ir_sub, ir_negate:  in STD_LOGIC;
    ir_shiftL, ir_shiftR:       in STD_LOGIC;
    ir_halt, ir_brZero, ir_brLess:   in STD_LOGIC;
    ir_brGtr, ir_branch:   in STD_LOGIC;
    ir_treg, ir_areg:   in STD_LOGIC_VECTOR(1 downto 0);
    ir_breg, ir_amode:   in STD_LOGIC_VECTOR(1 downto 0);
    iar_enA, iar_ld:   out STD_LOGIC;
    rf_enD, rf_ld, rf_selALU:  out STD_LOGIC;
    rf_target, rf_srcA:   out STD_LOGIC_VECTOR(1 downto 0);
    rf_srcB, rf_srcD:   out STD_LOGIC_VECTOR(1 downto 0);
    alu_regZ, alu_regLZ: in  STD_LOGIC;
    alu_op:    out STD_LOGIC_VECTOR(2 downto 0)
);
end controller;

architecture controllerArch of controller is 
type state_type is (  reset_state, 
    fetch0, fetch1, 
    load0, load1, load2, load3, 
    store0, store1, store2, store3, 
    copy0, copy1, 
    add0, add1, 
    sub0, sub1, 
    negate0, negate1, 
    shiftL0, shiftL1, 
    shiftR0, shiftR1, 
    halt, 
    brZero0, brZero1, 
    brLess0, brLess1, 
    brGtr0, brGtr1, 
    branch0, branch1 
);

signal state: state_type;

begin
    process(clk) begin 
        if clk'event and clk = '1' then 

            -- Code implementation here 

        end if;
    end process;
end controllerArch;
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_copy = '1' then state <= copy0;
      elsif ir_add = '1' then state <= add0;
      elsif ir_sub = '1' then state <= sub0;
      elsif ir_negate = '1' then state <= negate0;
      elsif ir_shiftL = '1' then state <= shiftL0;
      elsif ir_shiftR = '1' then state <= shiftR0;
      elsif ir_halt = '1' then state <= halt;
      elsif ir_brZero = '1' then state <= brZero0;
      elsif ir_brLess = '1' then state <= brLess0;
      elsif ir_brGtr = '1' then state <= brGtr0;
      elsif ir_branch = '1' then state <= branch0;
      end if;
    when load0 => state <= load1;
    when load1 =>
      if ir_amode = "10" then state <= load2;
      else state <= fetch0;
    end if;
    when load2 => state <= load3;
    when load3 => state <= fetch0;
    when store0 => state <= store1;
    when store1 =>
      if ir_amode = "10" then state <= store2;
      else state <= fetch0;
    end if;
    when store2 => state <= store3;
    when store3 => state <= fetch0;
    when copy0 => state <= copy1;
    when copy1 => state <= fetch0;
    when add0 => state <= add1;
    when add1 => state <= fetch0;
    when sub0 => state <= sub1;
    when sub1 => state <= fetch0;
    when negate0 => state <= negate1;
    when negate1 => state <= fetch0;
    when shiftL0 => state <= shiftL1;
    when shiftL1 => state <= fetch0;
    when shiftR0 => state <= shiftR1;
    when shiftR1 => state <= fetch0;
    when halt => state <= halt;
  end case;
when brZero0 => state <= brZero1;
when brZero1 => state <= fetch0;

when brLess0 => state <= brLess1;
when brLess1 => state <= fetch0;

when brGtr0 => state <= brGtr1;
when brGtr1 => state <= fetch0;

when branch0 => state <= branch1;
when branch1 => 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 (ir_amode = "01" and state = store0) or
        state = store2 then
        mem_rw <= '0';
    else
        mem_rw <= '1';
    end if;
end if;
end process;

mem_enD <= '1' when state = fetch0 or state = fetch1 or
((ir_amode = "01" or ir_amode = "10") and
 (state = load0 or state = load1 or
  state = load2 or state = load3)) or
((ir_amode = "10" and
 (state = store0 or state = store1))
else '0';

pc_enA <= '1' when state = fetch0 or state = fetch1
else '0';

pc_ld <= '1' when state = branch0 or
  (state = brZero0 and alu_regZ = '1') or
  (state = brLess0 and alu_regLZ = '1') or
  (state = brGtr0 and alu_regZ = '0' and alu_regLZ = '0')
else '0';

pc_inc <= '1' when state = fetch1
else '0';

ir_enA <= '1' when ((state = load0 or state = load1) and ir_amode /= "00") or
  state = store0 or state = store1
else '0';

ir_enD <= '1' when state = branch0 or state = brZero0 or
  state = brLess0 or state = brGtr0 or

-11-
((state = load0 or state = load1) and ir_amode = "00")
  else '0';
ir_ld <= '1' when state = fetch1
  else '0';

iar_enA <= '1' when state = load2 or state = load3 or
  state = store2 or state = store3
  else '0';
iar_ld <= '1' when (state = load1 and ir_amode = "10") or
  (state = store1 and ir_amode = "10")
  else '0';
rf_enD <= '1' when (ir_amode = "01" and (state= store0 or state=store1)) or
  (ir_amode = "10" and (state= store2 or state=store3))
  else '0';

rf_ld <= '1' when (ir_amode = "00" and state = load1) or
  (ir_amode = "01" and state = load1) or
  (ir_amode = "10" and state = load3) or
  state = copy1 or state = add1 or state = sub1 or
  state = negat1 or state = shiftL1 or state = shiftR1
  else '0';
rf_selAlu <= '1' when state = copy1 or state = add1 or state = sub1 or
  state = negat1 or state = shiftL1 or state = shiftR1
  else '0';
rf_target <= ir_treg;
rf_srcA <= ir_areg;
rf_srcB <= ir_breg;
rf_srcD <= ir_treg;
alu_op <= "000" when state = copy0 or state = copy1 else
  "001" when state = negat0 or state = negat1 else
  "010" when state = add0 or state = add1 else
  "011" when state = sub0 or state = sub1 else
  "100" when state = shiftL0 or state = shiftL1 else
  "101" when state = shiftR0 or state = shiftR1 else
  "000";

end controllerArch;

---------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity top_level is
  port (
    clk, reset: in STD_LOGIC;
    abusX: out STD_LOGIC_VECTOR(15 downto 0);
    dbusX: out STD_LOGIC_VECTOR(15 downto 0);
    mem_enDX, mem_rwX: out STD_LOGIC;
    pc_enAX, pc_ldX, pc_incX: out STD_LOGIC;
    ir_enAX, ir_enDX, ir_ldX: out STD_LOGIC;
  );
end top_level;
architecture topArch of top_level is

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, copy, add, sub, negate: out STD_LOGIC;  
        shiftL, shiftR, halt, brZero, brLess: out STD_LOGIC;  
        brGtr, branch: out STD_LOGIC;  
        treg, areg, breg, amode: out STD_LOGIC_VECTOR(1 downto 0);  
        irRegX: out STD_LOGIC_VECTOR(15 downto 0)  
    );
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 regFile
    port (  
        clk, en_D, ld, selALU, reset: in STD_LOGIC;  
        target, srcA, srcB, srcD: in STD_LOGIC_VECTOR(1 downto 0);  
        ALUbusR: in STD_LOGIC_VECTOR(15 downto 0);  
        dBus: inout STD_LOGIC_VECTOR(15 downto 0);  
        ALUbusA, ALUbusB: out STD_LOGIC_VECTOR(15 downto 0);  
        r0, r1, r2, r3: out STD_LOGIC_VECTOR(15 downto 0)  
    );
end component;
component alu
  port {
    op: in STD_LOGIC_VECTOR(2 downto 0);
    rf_BusA, rf_BusB: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0);
    rf_busR: out STD_LOGIC_VECTOR(15 downto 0);
    regZ, regLZ: 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_copy: in STD_LOGIC;
    ir_add, ir_sub, ir_negate: in STD_LOGIC;
    ir_shiftL, ir_shiftR: in STD_LOGIC;
    ir_halt, ir_brZero, ir_brLess: in STD_LOGIC;
    ir_brGtr, ir_branch: in STD_LOGIC;
    ir_treg, ir_areg: in STD_LOGIC_VECTOR(1 downto 0);
    ir_breg, ir_amode: in STD_LOGIC_VECTOR(1 downto 0);
    iar_enA, iar_ld: out STD_LOGIC;
    rf_enD, rf ld, rf selALU: out STD_LOGIC;
    rf_target, rf srcA: out STD_LOGIC_VECTOR(1 downto 0);
    rf srcB, rf srcD: out STD_LOGIC VECTOR(1 downto 0);
    alu_regZ, alu_regLZ: in STD_LOGIC;
    alu op: out STD_LOGIC_VECTOR(2 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 copy: STD_LOGIC;
signal ir_add, ir sub, ir negate: STD_LOGIC;
signal ir_shiftL, ir shiftR: STD_LOGIC;
signal ir_halt, ir_brZero, ir_brLess: STD_LOGIC;
signal  ir_brGtr, ir_branch:   STD_LOGIC;
signal  ir_treg, ir_areg:   STD_LOGIC_VECTOR(1 downto 0);
signal  ir_breg, ir_amode:   STD_LOGIC_VECTOR(1 downto 0);
signal iar_enA, iar_ld:  STD_LOGIC;
signal rf_enD, rf_ld, rf_selAlu:   STD_LOGIC;
signal rf_target, rf_srcA, rf_srcB:   STD_LOGIC_VECTOR(1 downto 0);
signal rf_srcD:   STD_LOGIC_VECTOR(1 downto 0);
signal alu_op:    STD_LOGIC_VECTOR(2 downto 0);
signal alu_regZ, alu_regLZ:   STD_LOGIC;
signal alu_busR:   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_copy, ir_add,
    ir_sub, ir_negate, ir_shiftL, ir_shiftR,
    ir_halt, ir_brZero, ir_brLess,
    ir_brGtr, ir_branch,
    ir_treg, ir_areg, ir_breg, ir_amode,
    ir_regX);

iar: indirect_addr_register port map(clk, iar_enA, iar_ld,reset,abus, dbus);

rf: regFile port map(clk, rf_enD, rf_ld, rf_selAlu, reset,
    rf_target, rf_srcA, rf_srcB, rf_srcD,
    alu_busR, dBus, rf_busA, rf_busB, r0, r1, r2, r3);

aluu: alu port map(alu_op, rf_BusA, rf_BusB, dBus,
    alu_BusR, alu_regZ, alu_regLZ);

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_copy, ir_add,
    ir_sub, ir_negate, ir_shiftL, ir_shiftR,
    ir_halt, ir_brZero, ir_brLess,
    ir_brGtr, ir_branch,
    ir_treg, ir_areg, ir_breg, ir_amode,
    iar_enA, iar_ld,
    rf_enD, rf_ld, rf_selAlu,
    rf_target, rf_srcA, rf_srcB, rf_srcD,
    alu_regZ, alu_regLZ, 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;

rf_enDX <= rf_enD;
rf_ldX <= rf_ld;
rf_selAluX <= rf_selAlu;
rf_targetX <= rf_target;
rf_srcAX <= rf_srcA;
rf_srcBX <= rf_srcB;
rf_srcDX <= rf_srcD;

alu_opX <= alu_op;
alu_regZX <= alu_regZ;
alu_regLZX <= alu_regLZ;
end topArch;