1. The figure below is a block diagram of an 8 bit pipelined multiplier. This circuit produces the product of two 8 bit numbers after a delay of several clock ticks. The use of registers between the ranks of adders makes it possible to input a new multiplier and multiplicand on every clock tick and get a product out on every clock tick. This kind of pipelined operation is commonly used in high performance systems.
Assuming that we start with no multiplication operations in progress, how many multiplies can be completed in 30 ns, assuming that the clock period is 4 ns? How many can be multiplies can be completed in 100 ns?

It takes 4 clock periods to produce the first product, so in the 7 clock periods available in 30 ns, 4 products can be produced. In 100 ns (25 clock periods) we get 22 products. The initial delay before the first product is produced is called the pipeline delay.

Assuming that the adders are implemented using the look-ahead circuit illustrated below what is the maximum gate delay we can tolerate and still have the circuit operate correctly with a clock period of 4 ns? Assume that the flip flop setup time is 700 ps, that the maximum propagation delay is 1 ns and that the maximum clock skew is 300 ps.

With a clock period of 4 ns, a setup time of 700 ps, a flip flop propagation delay of 1 ns and a clock skew of 300 ps, there is 2 ns available for the combinational circuit delays. The longest circuit path in the adder has 8 gates. The registers will also have a combinational circuit delay of perhaps 2 gates, meaning that we need a maximum gate delay of 200 ps.
2. Draw a schematic for a 4 bit grey code counter that has a load input, a count input and a reset input. If reset is high, the circuit should reset to 0000, otherwise if load is high, it should load a new value, otherwise if count is high it should advance by 1, otherwise, it should retain its current value. When counting, your counter should produce the sequence 0000, 0001, 0011, 0010, 0111, 0101, 0100, 1111, 1101, 1110, 1010, 1011, 1001, 1000 and should then wrap around to 0000. Show the Karnaugh maps and logic equations used to produce the increment logic.

<table>
<thead>
<tr>
<th>present state</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>0011</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
</tr>
<tr>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>0100</td>
<td>1100</td>
</tr>
<tr>
<td>0101</td>
<td>0100</td>
</tr>
<tr>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>0111</td>
<td>0101</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
</tr>
<tr>
<td>1001</td>
<td>1000</td>
</tr>
<tr>
<td>1010</td>
<td>1011</td>
</tr>
<tr>
<td>1011</td>
<td>1001</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>1101</td>
<td>1111</td>
</tr>
<tr>
<td>1110</td>
<td>1010</td>
</tr>
<tr>
<td>1111</td>
<td>1110</td>
</tr>
</tbody>
</table>

\[ D_A = A(C + D) + BC'D' \]

\[ D_B = B(C' + D) + A'CD' \]

\[ D_C = CD' + D(A \oplus B)' \]

\[ D_D = A \oplus B \oplus C \]
3. In the simple processor, the controller determines when each component in the system is permitted to use the bus. In other types of systems, there may be several independent subsystems that share a common bus. In such situations a bus arbiter is used to determine which subsystem gets to use the bus. In this problem you are to design a bus arbiter that can support three bus users. For each user, there is a request input and a grant output. The arbiter is a sequential circuit, which keeps track of the state of the bus. If the bus is free and one or more of the request lines is high, the arbiter selects one of the users and raises the corresponding grant signal. When the user is done with the bus, it is required to drop its grant signal for at least one clock tick. Design your arbiter so that if more than one user needs to use the bus multiple times, they take turns. Start by producing a state diagram for the arbiter, then design a VHDL module that implements the state diagram. Include an asynchronous reset input.

To provide equal access to the bus, the arbiter should give preference to users that haven’t used it recently. The circuit below does this by maintaining three separate idle states. In idle0, user 0 is given top priority for access to the bus, followed by user 1 and user 2. In idle1, user 1 is given top priority, then users2 and 0. In idle2, user 2 is given top priority, then users 0 and 1. Whenever a user releases the bus, the arbiter goes to the idle state that assigns the lowest priority to that user. The inputs are the request signals and the outputs are the grants.

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

entity arbiter is
    port(
        clk, reset: in STD_LOGIC;
        req: in STD_LOGIC_VECTOR (2 downto 0);
        grant: out STD_LOGIC_VECTOR (2 downto 0)
    );
end arbiter;
```
architecture arbiter_arch of arbiter is
  type state_type is (idle0, idle1, idle2, busy0, busy1, busy2);
  signal state: state_type;
begin
  process(clk, reset) begin
    if reset = '1' then
      state <= idle0;
    elsif clk'event and clk = '1' then
      if state = idle0 and req(0) = '1' then
        state <= busy0;
      elsif state = idle0 and req(0) = '0' and req(1) = '1' then
        state <= busy1;
      elsif state = idle0 and req(0) = '0' and req(1) = '0' and req(2) = '1' then
        state <= busy2;
      elsif state = idle1 and req(1) = '1' then
        state <= busy1;
      elsif state = idle1 and req(1) = '0' and req(2) = '1' then
        state <= busy2;
      elsif state = idle1 and req(1) = '0' and req(2) = '0' and req(0) = '1' then
        state <= busy0;
      elsif state = idle2 and req(2) = '1' then
        state <= busy2;
      elsif state = idle2 and req(2) = '0' and req(0) = '1' then
        state <= busy0;
      elsif state = idle2 and req(2) = '0' and req(0) = '0' and req(1) = '1' then
        state <= busy1;
      elsif state = busy0 and req(0) = '0' then
        state <= idle1;
      elsif state = busy1 and req(1) = '0' then
        state <= idle2;
      elsif state = busy2 and req(2) = '0' then
        state <= idle0;
    end if;
  end if;
end process;
grant <=
  "001" when state = busy0 else
  "010" when state = busy1 else
  "100" when state = busy2 else
  "000";
end arbiter_arch;
4. The figure below is a block diagram of a special-purpose processor that can perform any of several operations on the data in its two registers, $A$ and $B$. In particular, if op=000, it loads the $A$ register from the input, if op=001, it loads the $B$ register, if op=010, it sends $A$ to the output, if op=011, it sends $B$ to the output, if op=100, it replaces $A$ with the sum of $A$ and $B$, if op=101 it replaces $B$ with the negation of $B$, if op = 110, it replaces $A$ with the bitwise AND of $A$ and $B$ and if op=111, it replaces $B$ with the bitwise OR of $A$ and $B$. Write VHDL modules to implement the registers, the ALU, the controller and a top level module that combines them using structural VHDL. Assume that the clock period is 10 ns, and that the ALU requires 12 ns to do an AND or OR operation and 22 ns to perform an add or negate operation. Make the registers 16 bits wide.

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

entity reg is
  port (    clk, reset: in STD_LOGIC;
             LD: in STD_LOGIC;
             Din: in STD_LOGIC_VECTOR (15 downto 0);
             Dout: out STD_LOGIC_VECTOR (15 downto 0)
  );
end reg;

architecture reg_arch of reg is
signal reg: STD_LOGIC_VECTOR (15 downto 0);
begin
  process(clk,reset) begin
    if reset = '1' then
      reg <= x"0000";
    end if;
    if rising_edge(clk) then
      if LD = '1' then
        reg <= Din;
      else
        reg <= reg;
      end if;
    end if;
  end process;
end reg_arch;
```
elsif clk'event and clk = '1' then
    if LD = '1' then
        reg <= Din;
    end if;
end if;
end process;
Dout <= reg;
end reg_arch;

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

entity alu is
    port (
        ctl: in STD_LOGIC_VECTOR(2 downto 0);
        A, B: in STD_LOGIC_VECTOR (15 downto 0);
        R: out STD_LOGIC_VECTOR (15 downto 0)
    );
end alu;

architecture alu_arch of alu is
begin
    R <= A when ctl = "010" else
        B when ctl = "011" else
        A+B when ctl = "100" else
        (not B)+'1' when ctl = "101" else
        A and B when ctl = "110" else
        A or B when ctl = "111" else
        x"0000"
    end alu_arch;

library IEEE;
use IEEE.std_logic_1164.all;

entity control is
    port (clk, reset: in STD_LOGIC;
          OP: in STD_LOGIC_VECTOR(2 downto 0);
          aluC: out STD_LOGIC_VECTOR(2 downto 0);
          ldA, ldB: out STD_LOGIC;
          muxC: out STD_LOGIC
    );
end control;

architecture control_arch of control is
begin
    type state_type is (getOp, step0, step1, step2);
    signal state: state_type;
    signal opReg: STD_LOGIC_VECTOR (2 downto 0);
    begin
        process(clk, reset) begin
            if reset = '1' then
                state <= getOp; opReg <= "000";
            elsif clk'event and clk = '1' then
                if (state = getOp) then
                    state <= step0;
                    opReg <= "000";
                elsif (state = step0) then
                    state <= step1;
                    opReg <= "001";
                elsif (state = step1) then
                    state <= step2;
                    opReg <= "010";
                elsif (state = step2) then
                    state <= getOp;
                    opReg <= "000";
                end if;
            end if;
        end process;
    end control_arch;
elsif clk'event and clk = '1' then
  if state = getOp then
    state <= step0; opReg <= OP;
  elsif state = step0 and opReg(1) = '0' then
    state <= getOp; opReg <= "000";
  elsif state = step0 then
    state <= step1;
  elsif state = step1 and 
    (opReg = "110" or opReg = "111") then
    state <= getOp; opReg <= "000";
  elsif state = step1 then
    state <= step2;
  elsif state = step2 then
    state <= getOp; opReg <= "000";
  end if;
end if;
end process;
muxC <= '1' when opReg = "000" or opReg = "001" else '0';
ldA <= '1' when (opReg = "000" and state = step0) or 
  (opReg = "100" and state = step2) or 
  (opReg = "110" and state = step1) 
else '0';
ldB <= '1' when (opReg = "001" and state = step0) or 
  (opReg = "101" and state = step2) or 
  (opReg = "111" and state = step1) 
else '0';
aluC <= opReg;
end control_arch;

library IEEE;
use IEEE.std_logic_1164.all;

entity top is
  port (
    clk, reset: in STD_LOGIC;
    operation: in STD_LOGIC_VECTOR(2 downto 0);
    Din: in STD_LOGIC_VECTOR(15 downto 0);
    Dout: out STD_LOGIC_VECTOR(15 downto 0)
  );
end top;

architecture top_arch of top is
  component reg
    port (
      clk, reset: in STD_LOGIC;
      LD: in STD_LOGIC;
      Din: in STD_LOGIC_VECTOR (15 downto 0);
      Dout: out STD_LOGIC_VECTOR (15 downto 0)
    );
  end component;
  component alu
    port (
      ctl: in STD_LOGIC_VECTOR(2 downto 0);
      A, B: in STD_LOGIC_VECTOR (15 downto 0);
    );
  end component;
begin
R: out STD_LOGIC_VECTOR (15 downto 0)
);
end component;
component control
  port(
    clk, reset: in STD_LOGIC;
    OP: in STD_LOGIC_VECTOR(2 downto 0);
    aluC: out STD_LOGIC_VECTOR(2 downto 0);
    ldA, ldB: out STD_LOGIC;
    muxC: out STD_LOGIC
  );
end component;
signal ldA, ldB, muxC: STD_LOGIC;
signal aluC: STD_LOGIC_VECTOR(2 downto 0);
signal rAout, rBout, aluOut, muxOut: STD_LOGIC_VECTOR(15 downto 0);
beg
  rA: reg port map(clk, reset, ldA, muxOut, rAout);
  rB: reg port map(clk, reset, ldB, muxOut, rBout);
  alunit: alu port map(aluC, rAout, rBout, aluOut);
  ctl: control port map(clk, reset, operation, aluC, ldA, ldB, muxC);
  muxOut <= Din when muxC = '1' else aluOut;
  Dout <= aluOut;
end top_arch;
5. Write a program for the simple processor from section 6 of the notes that checks to see if a given ASCII character string is a palindrome. The inputs to your program are stored at locations 30 and 31 (hex). The value at location 30 is a pointer to the first character in the character string. The value in location 31 is the number of characters in the string. Your program should write 1 in location 32 if the string is a palindrome and 0, if it is not. Try loading your program in the memory for the simple processor and running a simulation that executes your program. Does your program work correctly?

The code is shown below.

```
-- this code checks for a palindrome
ram(0) <= x"0030"; -- lo = start
ram(1) <= x"102e";
ram(2) <= x"8001"; -- hi = start + length - 1
ram(3) <= x"3000";
ram(4) <= x"2030";
ram(5) <= x"2031";
ram(6) <= x"102f";
ram(7) <= x"002e"; -- loop: if lo = hi or lo = hi + 1 then
ram(8) <= x"3000";
ram(9) <= x"202f";
ram(10) <= x"500e";
ram(11) <= x"9001";
ram(12) <= x"500e";
ram(13) <= x"4011";
ram(14) <= x"8001"; -- result = 1
ram(15) <= x"1032";
ram(16) <= x"3001"; -- quit
ram(17) <= x"602e"; -- if *lo != *hi then
ram(18) <= x"3000";
ram(19) <= x"1032"; -- (store -*lo temporarily)
ram(20) <= x"602f";
ram(21) <= x"2032";
ram(22) <= x"501a";
ram(23) <= x"8000"; -- result = 0
ram(24) <= x"1032";
ram(25) <= x"3001"; -- quit
ram(26) <= x"002e"; -- lo = lo + 1
ram(27) <= x"9001";
ram(28) <= x"102e";
ram(29) <= x"8001"; -- hi = hi - 1
ram(30) <= x"3000";
ram(31) <= x"202f";
ram(32) <= x"102f";
ram(33) <= x"4007"; -- goto loop

ram(46) <= x"0000"; -- lo
ram(47) <= x"0000"; -- hi
ram(48) <= x"0033"; -- start
ram(49) <= x"0005"; -- length
ram(50) <= x"0000"; -- result
ram(51) <= x"0061"; -- 'a'
ram(52) <= x"0062"; -- 'b'
ram(53) <= x"0063"; -- 'c'
ram(54) <= x"0062"; -- 'b'
ram(55) <= x"0061"; -- 'a'
```
6. Show how to modify the VHDL for the simple processor to implement a new instruction that performs a logical AND operation. Specifically, extend it so that the instruction code Bxxx causes the value in the accumulator to be replaced with the bitwise AND of its original value with the value in location xxx. The VHDL for the simple processor is on the web site.

```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
begin
    signal pcReg: STD_LOGIC_VECTOR(15 downto 0);
    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 "ZZZZZZZZZZZZZZZZZZZZZZZ";
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, branch: out STD_LOGIC;
    cbranch, iload, istore, mload, madd, andd: out STD_LOGIC
); 
end instruction_register;

architecture irArch of instruction_register is
begin
    signal irReg: STD_LOGIC_VECTOR(15 downto 0);
    process(clk) begin
        if clk'event and clk = '1' then 
            if reset = '1' then
                irReg <= x"0000";
            elsif ld = '1' then
                irReg <= dBus;
            end if;
        end if;
    end process;
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
            "ZZZZZZZZZZZZZZZ";
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);
    );
end accumulator;
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);
begnin
process(clk) begin
if clk'event and clk = '1' then
  if reset = '1' then
    accReg <= x"0000"
  elsif ld = '1' and selAlu = '1' then
    accReg <= aluD;
  elsif ld = '1' and selAlu = '0' then
    accReg <= dBus;
  end if;
end if;
end process;
dBus <= accReg when en_D = '1' else
  "ZZZZZZZZZZZZZZZZ";
q <= accReg;
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: out STD_LOGIC)
);
end alu;
architecture aluArch of alu is
begin
  result <= (not accD) + x"0001" when op = "00" else
            accD + dBus when op = "01" else
            accD and dBus when op = "10" else
            x"0000"
  accZ <= '1' when accD = x"0000" else '0'
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)
)
);

-14-
end ram;

architecture ramArch of ram is
begin
  process(reset, r_w) begin
    if reset = '1' then
      -- this code checks the AND instruction
      ram(0) <= x"85a5"; -- acc = 0585
      ram(1) <= x"b005"; -- acc &= xfff
      ram(2) <= x"b006"; -- acc &= 0000
      ram(3) <= x"3001"; -- halt
      ram(5) <= x"ffff";
      ram(6) <= x"0000";
    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_add: in STD_LOGIC;
     ir_neg, ir_halt, ir_branch: in STD_LOGIC;
     ir_cbranch, ir_iload: in STD_LOGIC;
     ir_istore, ir_mload, ir_madd: in STD_LOGIC;
     iar_enA, iar_ld:
     acc_enD, acc_ld, acc_selAlu: out STD_LOGIC;
     alu_accZ: in STD_LOGIC;
     alu_op: out STD_LOGIC_VECTOR(1 downto 0)
   );
end controller;

architecture controllerArch of controller is
begin
  architecture ramArch of ram is
  type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
signal ram: ram_typ;

end ramArch;
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_load = '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_mload = '1' then state <= mload0;
            elsif ir_madd = '1' then state <= madd0;
            elsif ir_andd = '1' then state <= andd0;
            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;
          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 mload0 => state <= mload1;
          when mload1 => state <= fetch0;
        end case;
      end if;
    end if;
  end process;
end;

when madd0 =>  state <= madd1;
when madd1 =>  state <= fetch0;

when andd0 =>  state <= andd1;
when andd1 =>  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 or
  state = andd0 or state = andd1
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')
else '0';

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

ir_enA <= '1' when state = load0 or state = load1 or
  state = store0 or state = store1 or
  state = add0 or state = add1 or
  state = iload0 or state = iload1 or
  state = istore0 or state = istore1 or
  state = andd0 or state = andd1
else '0';

ir_enD <= '1' when state = branch0 or state = cbranch0 or state = mload0
  or state = madd0 or state = madd1
else '0';

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

iar_enA <= '1' when state = iload2 or state = iload3 or
  state = istore2 or state = istore3
else '0';

iar_ld <= '1' when state = iload1 or state = istore1
else '0';

acc_enD <= '1' when state = store0 or state = store1 or
  state = istore2 or state = istore3
else '0';

acc_ld <= '1' when state = load1 or state = add1 or state = negate1 or
  state = iload3 or state = mload0 or state = madd0 or
  state = madd1 or
  state = andd1
else '0';

acc_selAlu <= '1' when state = add1 or state = negate1 or state = madd1
or state = andd1
else '0';
alu_op <= "01" when state = add0 or state = add1 or
        state = madd0 or state = madd1 else
       "10" when state = andd0 or state = andd1
else "00";
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;
                 iar_enAX, iar_ldX: out STD_LOGIC;
                 acc_enDX, acc_ldX, acc_selAluX: out STD_LOGIC;
                 acc_QX: out STD_LOGIC_VECTOR(15 downto 0);
                 alu_accZX: out STD_LOGIC;
                 alu_opX: out STD_LOGIC_VECTOR(1 downto 0)
    );
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: in STD_LOGIC_VECTOR(15 downto 0);
                   load, store, add, neg, halt, branch: out STD_LOGIC;
                   cbranch, iload, istore, mload, madd, andd: 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)
        );
    end component;
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: 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_i1load: in STD_LOGIC;
ir_istore, ir_mload, ir_madd: in STD_LOGIC;
ir_andd: in STD_LOGIC;
ir_enA, iar ld: out STD_LOGIC;
acc_enD, acc ld, acc selAlu: out STD_LOGIC;
alu_accZ: 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_negate, ir halt, ir branch: STD_LOGIC;
signal ir_cbranch, ir i1load, ir istore: STD_LOGIC;
signal ir_mload, ir madd, ir andd: STD_LOGIC;
signal iar_enA, iar ld: STD_LOGIC;
signal acc_enD, acc ld, acc selAlu: STD_LOGIC;
signal alu accZ: STD_LOGIC;
signal alu_op: STD_LOGIC_VECTOR(1 downto 0);
signal acc Q: STD_LOGIC_VECTOR(15 downto 0);
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_negate, ir_halt, ir_branch, 
ir_cbranch, ir_iload, ir_istore, 
ir_mload, ir_madd, \texttt{ir\_andd});

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);

aluu: alu port map(alu_op, acc_Q, dbus, alu_result, alu_accZ);

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_mload, ir_madd, \texttt{ir\_andd}, 
 iar_enA, iar_ld, acc_enD, acc_ld, acc_selAlu, 
alu_accZ, 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;
end topArch;