1. In this problem, I want you to consider the implementation of a controller using PLAs. We will take advantage of the tool \texttt{espresso} to optimize our logic.

Consider a controller defined by the following VHDL description:

```vhdl
architecture rtl of controller is
  subtype state_type is std_ulogic_vector(0 to 3);
  constant s0: state_type := "0001";
  constant s1: state_type := "0010";
  constant s2: state_type := "0100";
  constant s3: state_type := "1000";
  signal state, next_state: state_type;
  signal con1, con2, con3: std_ulogic;
  signal out1, out2: std_ulogic;
  signal clk;

begin
  state_logic: process(state, con1, con2, con3) is
  begin
    out1 <= '0';
    out2 <= '0';
    case state is
      when s0 =>
        out1 <= '0';
        out2 <= '0';
        next_state <= s1;
      when s1 =>
        out1 <= '1';
        if con1 = '1' then
          next_state <= s2;
        else
          next_state <= s1;
        end if;
      when others =>
        next_state <= s3;
    end case;
    next_state <= state;
  end process state_logic;
end rtl;
```

next_state <= s1;
end if;
when s2 =>
  out2 <= '1';
  next_state <= s3;
when s3 =>
  if con2 = '0' then
    next_state <= s3;
  elsif con3 = '0' then
    out1 <= '0';
    next_state <= s2;
  else
    next_state <= s1;
  end if;
end case;
end process state_logic;

state_register: process (clk) is
begin
  if (clk = '1' and not(clk’stable)) then
    state <= next_state;
  end if;
end process state_register;
end architecture rtl;

Implement this finite-state machine as a PLA and a (four-bit) flip-flop. Present your results as (hand-drawn) schematics. Don’t worry about sizing or simulating your design. To design the PLA, use espresso for two-level logic optimization. You can assess the espresso man pages on the input file format by typing:

man -s 5 espresso

Additional documentation on espresso can also be found on the main espresso man pages:

man espresso
<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phi1</td>
<td>input</td>
<td>phase 1 clock</td>
</tr>
<tr>
<td>phi2</td>
<td>input</td>
<td>phase 2 clock</td>
</tr>
<tr>
<td>mem_read</td>
<td>input</td>
<td>enable the memory for read</td>
</tr>
<tr>
<td>mem_write</td>
<td>input</td>
<td>enable the memory for write</td>
</tr>
<tr>
<td>iobus&lt;0:7&gt;</td>
<td>bidi</td>
<td>data bus</td>
</tr>
<tr>
<td>addr&lt;0:2&gt;</td>
<td>input</td>
<td>address</td>
</tr>
</tbody>
</table>

The next problem is part of your final design project; you should work on this part of the problem set with your design-project partner.

2. You need to design the memory for your final design project. Your memory stores 8 8-bit (one-byte) words and has the following interface to the rest of your core:

   Please precharge your memory with `phi2` and qualify both the wordline and the write with the `phi1` clock. Use the `mem_read` signal to tristate the read driver. Only one level of decoding should be necessary (no predecoder). You are free to use the SRAM layout found in the cell `sramcell` in the library `arrayLib` in `/tools2/courses/ee4321/arrayLib`. An example of how this is tiled can be found in `array_example`. This is not a particularly dense array, but it's good enough for our purposes.

   To complete your memory design, I expect:

   - Sized schematics
   - Layout of your memory design (remember to begin by making a tiling diagram and then stick layouts of your cells)
   - HSPICE results for the read and write delays of your SRAM. Verify that reads do not disturb the contents of the cells and that you will always be able to write your cell.
   - Use Nanosim to verify functionality for a larger number of patterns.

Please submit your layout electronically.