1. (8 points) What is the dual of the expression \( A'B + (C' + D)(A + B') + (A+B)'C \)?

\[
(A'+B)(C'D+AB')(AB)'+C
\]

What is its complement?

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

Using Boolean algebra, derive an expression equivalent to \( A'B + (C' + D)(A + B') + (A+B)'C \) that uses at most 2 AND operations and 4 OR operations.

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

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

\[
= A'(B+C) + (C' + D)(A + B')
\]
2. (8 points) How many bits are required to represent the decimal number 3472 in ASCII?

32 bits

How many bits are required to represent 3472 in BCD?

16 bits

How many bits are required to represent 3472 in binary?

12 bits

Give the bits that represent the excess 3 representation of 3472.

0110 0111 1010 0101
3. (12 points) The simulation output shows selected signals from the processor introduced in section 1 of the course notes. The portions of the output corresponding to four different instructions are outlined. Identify the instructions that are being executed. Give the name of the instruction (e.g. direct load, branch-on-positive) and its complete numeric representation. Note that some parts of the simulation output have been blanked out.

The first instruction changes the ACC but does not access memory, so it must be either an immediate load or a negate instruction. It cannot be an immediate load, since the high order hex digit of the ACC becomes 5 and an immediate load can only make the high order hex digit 0 or F. So, it must be a negate instruction, 0001. This conclusion is confirmed by the fact that the new value of the ACC is the 2s complement of the previous value.

The second instruction also changes the ACC, but does not access memory. Since the new ACC value is not the 2s complement of the previous value, this must be an immediate load 1FFF.

The third instruction does read from memory, so it must be a direct load or an add instruction. Since the value returned from memory (which appears on the data bus) is 0001, and the new ACC value is 0000, it must be an add instruction, and since the value on the address bus is 0008, it must be, A008. This is confirmed by the fact that the new ACC value is 1 more than the previous value.

The last instruction changes the PC, so it must be a branch instruction. In fact, we can see the instruction returned on the data bus in this case is 700d, which is the branch-on-zero instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>halt – halt execution</td>
</tr>
<tr>
<td>0001</td>
<td>negate – ACC := (-\text{ACC})</td>
</tr>
<tr>
<td>1xxx</td>
<td>immediate load – if sign bit of (xxx) is 0 then ACC := (0xxx) else ACC := (fxxx)</td>
</tr>
<tr>
<td>2xxx</td>
<td>direct load – ACC := (M[0xxx])</td>
</tr>
<tr>
<td>3xxx</td>
<td>indirect load – ACC := (M[M[0xxx]])</td>
</tr>
<tr>
<td>4xxx</td>
<td>direct store – (M[0xxx] := \text{ACC})</td>
</tr>
<tr>
<td>5xxx</td>
<td>indirect store – (M[M[0xxx]] := \text{ACC})</td>
</tr>
<tr>
<td>6xxx</td>
<td>branch – (PC := 0xxx)</td>
</tr>
<tr>
<td>7xxx</td>
<td>branch if zero – if ACC = 0 then (PC := 0xxx)</td>
</tr>
<tr>
<td>8xxx</td>
<td>branch if positive – if ACC &gt; 0 then (PC := 0xxx)</td>
</tr>
<tr>
<td>9xxx</td>
<td>branch if negative – if ACC &lt; 0 then (PC := 0xxx)</td>
</tr>
<tr>
<td>axxx</td>
<td>add – ACC := ACC + (M[0xxx])</td>
</tr>
</tbody>
</table>
4. (12 points) Consider the circuit shown below. Suppose this circuit is implemented directly, using CMOS, with each AND gate implemented using a NAND and an inverter (similarly for OR gates). What is the worst-case delay for the circuit, assuming that NAND and NOR gates have a delay of 1 ns each, and inverters have a delay of 0.5 ns? Highlight the path through the circuit that accounts for the worst-case delay.

The worst-case delay is 5 ns for the highlighted path.

Show an alternate implementation, using NANDs, NORs and inverters that reduces the delay by at least 1.5 ns.

This circuit has a worst-case delay of 3.5 ns from Y to F and from Y to G.
5. (12 points) Use a Karnaugh map to find a simplest \textit{sum-of-products} expression for $F(X,Y,Z) = \Sigma m(0,2,5)$, $d(X,Y,Z) = \Sigma m(3,6)$.

\[
\begin{array}{ccc|c}
X & YZ & \multicolumn{2}{c}{X'Z' + XY'Z} \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & x \\
1 & 0 & 1 & x \\
1 & 1 & x & x \\
\end{array}
\]

Use a Karnaugh map to find a simplest \textit{product-of-sums} expression for $F(A,B,C,D) = \Sigma m(0,2,4,5,8,10,15)$, $d(A,B,C,D) = \Sigma m(3,6,7,11,13)$.

\[
\begin{array}{ccc|c}
A & CD & \multicolumn{2}{c}{F'} \\
00 & 0 & 1 & 1 \\
01 & 1 & x & x \\
10 & 1 & 0 & 1 \\
11 & 1 & 0 & x \\
\end{array}
\]

$F' = B'D + ABD'$

$F = (B + D')(A' + B' + D)$
6. (15 points) The circuit below implements a four bit first mismatch function. That is, the output $x_i$ is high, if $i$ is the largest integer for which $a_i \neq b_i$. Note that at most one of the $x$ outputs is high and the valid output is high if there is a mismatch in some input bit pair. What is the worst-case propagation delay for a 32 bit version of this circuit, if every simple gate has a delay of 1 ns?

The worst-case propagation delay for the 32 bit version is 34 ns.

Draw another version of the circuit above that uses lookahead to make it faster.
7. (12 points) For each VHDL code fragment shown below, write an equivalent set of simple signal assignments (each involving just a single signal, not a vector or signals).

(a) \[ z(2 \text{ downto } 0) <= (a \text{ or } b) \& b \& (b \text{ xor } c); \]

\[ z(2) <= a \text{ or } b; \]
\[ z(1) <= b; \]
\[ z(0) <= b \text{ xor } c; \]

(b) \[ c <= x \text{ and } y; \]
if \( a \neq b \) then
\[ d <= x \text{ or } y; \]
else
\[ c <= \text{ not } y; \]
\[ d <= '0'; \]
end if;

\[ c <= ((a \text{ xor } b) \text{ and } x \text{ and } y) \text{ or } ((a \text{ xnor } b) \text{ and } \text{ not } y)); \]
\[ d <= (a \text{ xor } b) \text{ and } (x \text{ or } y); \]

(c) for \( i \) in 1 to 2 loop
\[ a(i) <= a(i-1) \text{ and } x(i); \]
\[ x(i) <= b(i) \text{ or } x(i-1); \]
end loop;

\[ a(1) <= a(0) \text{ and } x(1); \]
\[ x(1) <= b(1) \text{ or } x(0); \]
\[ a(2) <= a(1) \text{ and } x(2); \]
\[ x(2) <= b(2) \text{ or } x(1); \]
library IEEE;
use IEEE.std_logic_1164.all;

entity foo is port (  
    A: in std_logic_vector(0 to 3);  
    B: in std_logic_vector(0 to 3);  
    C: in std_logic;  
    X: out std_logic_vector(0 to 4));  
end foo;

architecture bar of foo is
begin
    signal U: std_logic_vector(0 to 4);
    process(A, B, U) begin
        U(0) <= C;
        for i in 0 to 3 loop
            X(i) <= A(i) xor B(i) and (not U(i));
            U(i+1) <= (A(i) and B(i)) or (B(i) and U(i));
        end loop;
        X(4) <= U(4) xor A(0);
    end process;
end bar;