18-240 Study Guide

Part 1, Combinational Logic Questions

This study guide is provided as an aid in helping you to study for the ECE Department’s 18-240, Fundamentals of Computer Engineering. The guide is a collection of previous test and homework questions for which solutions were handed out. Thus some of these questions are indicative of questions you might find on a test. Since answers are provided for these questions, you can use this guide to get extra practice on course-related problems.

The guide consists of several files covering different topics. Please don’t infer that this guide is all inclusive in terms of course topics or possible test questions. Further, the topics are distributed among several files. Please don’t infer that the first test corresponds to file number one. It probably doesn’t. Check with the course announcements regarding topics to be covered on a test.

Oh yes, you might find some errors. Please let us know so that we can fix them for others.

This file was produced using FrameMaker and saved in PDF format for Adobe Acrobat readers. Acrobat Exchange was used to include hot links between questions and answers.

1.1 Implicants

For the Karnaugh map shown below:

a. Give one example of an implicant.

a. List all of the prime implicants,

a. List all of the essential prime implicants.

\[
\begin{array}{cccc}
\text{ab} & 00 & 01 & 11 & 10 \\
\text{cd} & & & & \\
00 & 1 & & & \\
01 & & 1 & 1 & \\
11 & & 1 & & \\
10 & & & 1 & \\
\end{array}
\]
1.2 Simple logic function

Show a truth table, K-map, and simplified Boolean expression for the following function \( f \).

The months of the year are encoded as January = 0000\(_2\), February = 0001\(_2\), … December = 1011\(_2\); these are the only legal inputs to the function. The function \( f \) should be true when the month has 31 days in it. (30 days has September, April, June, and November. All the rest have 31 except February which comes out on the short end of the stick with 28 or 29).

1.3 Theorem Proving

Prove that \( X \cdot (X + Y) = X \) using only the following axioms and theorems from Boolean algebra. Justify each step of your proof with a reference to one of these axioms/theorems.

Operations with 0, 1:
1. \( X + 0 = X \)
2. \( X + 1 = 1 \)
3. \( X + X = X \)
4. \( (X')' = X \)
5. \( X + X' = 1 \)
6. \( X + Y = Y + X \)
7. \( (X+Y)+Z = X+(Y+Z) \)
8. \( X\cdot(Y+Z) = X\cdot Y + X\cdot Z \)
9. \( X + X'\cdot Y = X + Y \)
10. \( X\cdot(X' + Y) = X\cdot Y \)

1.4 Theorem Proving

Prove that \( (X \cdot Y') + Y = X + Y \) using only the following axioms and theorems from Boolean algebra. Justify each step of your proof with a reference to one of these axioms/theorems.

Operations with 0, 1:
1. \( X + 0 = X \)
2. \( X + 1 = 1 \)
3. \( X + X = X \)
4. \( (X')' = X \)
5. \( X + X' = 1 \)
6. \( X + Y = Y + X \)
7. \( (X+Y)+Z = X+(Y+Z) \)
8. \( X\cdot(Y+Z) = X\cdot Y + X\cdot Z \)
9. \( X + X'\cdot Y = X + Y \)
10. \( X•(X'+Y) = X•Y \)
1.5 Canonical Forms

Given the following gate design, answer the questions below.

a. Write function $F$ from the gate diagram. (There is no need to minimize, just write it from the diagram.)

$$F(A, B, C) =$$

b. What is the canonical SOP form for function $F$?

$$F(A, B, C) = \Sigma m ( )$$

c. What is the canonical POS form for function $F$?

$$F(A, B, C) = \Pi M ( )$$

1.6 Canonical Forms

Given the following implementation of function $F$, write this function as a sum of minterms and product of maxterms. Fill in the truth table first. Use the variable ordering shown in the truth table.

Fill these in:

$$F = \Sigma m ( )$$

$$F = \Pi M ( )$$
1.7 Canonical forms of Combinational Logic

Given the Karnaugh map below, implement a circuit in minimized POS (product of sums) form. You may either give the equation for the new circuit or draw the circuit, you need not do both!

1.8 Factoring a Boolean expression

A Boolean function, \( f(a, b, c, d) \) is to be implemented using a 2:1 multiplexor. As shown in the figure below, the variable \( c \) has been attached to the mux select input. This means that when \( c=1 \), \( f_c \) is selected by the mux. Also, when \( c=0 \), \( f_{c'} \) is selected by the mux (note that the prime -'-' is on the subscript \( c \), not on the \( f \)). Write \( f_c \) and \( f_{c'} \) in Boolean form. (no K-maps, minimization, etc. is necessary, just write \( f_c = \text{blah'} \) blah, and \( f_{c'} = \text{blah'} \) blah blah’)

\[
\begin{align*}
\text{AB} & \quad \text{CD} \\
00 & \quad 001 \quad 11 \quad 10 \\
01 & \quad 011 \quad 10 \\
11 & \quad 011 \quad 10 \\
10 & \quad 000 \quad 00
\end{align*}
\]

\[
f_c = \\
f_{c'} = \\
\]

1.9 Implementation Methods

Given the function: \( F(A, B, C, D) = \Sigma m(0, 1, 5, 7, 8, 10, 11,15) \)

a. Implement \( F \) using a 16:1 multiplexor

b. Implement \( F \) using an 8:1 multiplexor. Use \( D \) as the save variable, i.e. use inputs \( A, B, \) and \( C \) as the select lines. Assume that you have \( D' \) available.

c. Implement the function using a 4:16 decoder and an OR gate.

1.10 Boolean Derivative

Boolean Algebra defines a Boolean derivative as:
\[ \frac{dF}{dX} = F_X' \oplus F_X \]

where:

• \( F_X' \) is the function obtained by replacing every occurrence of \( X \) in \( F \) with 0
• \( F_X \) is the function obtained by replacing every occurrence of \( X \) in \( F \) with 1

For example, if \( F(A, B, C) = A+B\cdot C \), then \( F_B(A,B,C) = F(A,0,C) = A+0\cdot C = A \). And, \( F_B(A,B,C) = F(A,1,C) = A+C \).

It turns out that this thing, which usually is called the \textit{Boolean Difference}, has lots of the nice properties of ordinary derivatives.

So, who cares? The expression that results from the Boolean Difference (when you XOR the two terms together and do the algebra) tells you how the function \( F \) changes with respect to \( X \). That is, \textit{it tells what the inputs (other than \( X \)) have to be for a change in \( X \) to be seen on output \( F \)}.

Aha! This could be used to generate a test vector for input \( X \). That is, set the other inputs as specified by \( \frac{dF}{dX} \) (or possibly as don’t cares), and when \( X \) changes, so will the output.

**What to do:** Here’s a function \( G \). Draw a 2-level gate diagram of this circuit. Calculate \( \frac{dG}{dC} \) by expanding the XOR in terms of its equivalent sum of products. Use the result \( \frac{dG}{dC} \) to specify values for inputs \( A \) and \( B \) that will allow a change on input \( C \) to propagate to \( G \). Show the path(s) through the logic from \( C \) to \( G \).

\[ G = AC + BC + AB'C' \]

**1.11 Write the cofactor**

a. Write the cofactor \( \bar{F}_{xy} \) of the following function \( F \):

\[ F(v, w, x, y, z) = vwyz + vxz + xyz + xz + wyz \]

b. When picking variables for writing cofactors, it is good to pick one where the cofactor becomes 0. What single variable would you pick in \( F \) above so that the cofactor is 0?

**1.12 Quine-McCluskey**

Use the Quine McCluskey method to derive a minimal SOP form of the function \( F \). Show carefully (label) all the steps of your work for partial credit. Write the final Boolean expression for minimized \( F \).

\[ F(A,B,C,D) = \Sigma m(3, 5, 8, 11, 13, 15) + d(2, 7) \]

**1.13 Quine McCluskey**

Use the Quine-McCluskey method to minimize the following Boolean function. Show all of your work, including the implication table and the prime implicant chart. Express the final answer in a minimized sum of products.

\[ f(a, b, c, d) = \Sigma m (0, 2, 8, 10, 12) + \Sigma d(9, 13, 15) \]
1.14 Combinational Logic Design

Consider the 3-variable function $F$ shown below. You must implement this function as cheaply as possible.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

You have only these parts available:

- Inverters. These are free.
- Any 2-input gate. These cost $1 per gate.
- Any 3-input gate. These cost $2 per gate.
- 4:1 Muxes and 3:8 decoders (all active-high inputs and outputs). These cost $3 each.

This function can be implemented in three different “styles”: minimal 2-level form, mux-logic form, or decoder-logic form.

**Do this:** Design and draw a logic diagram of the implementation of this function in each of these styles, using only the above parts. Calculate the cost of each. Which is cheapest? Show your work.

1.15 Word Problem

A logic network has four inputs (i3, i2, i1, and i0) and two outputs (o1, and o0). At least one of the inputs is always asserted high. If a given input line has a logic 1 applied to it, the output signals will encode its index in binary. For example, if only i2 is asserted, the output reads o1=1, o0=0. If two or more inputs are asserted, the output will be set according to which input has the lowest index (i3 > i2 > i1 > i0). Thus, if i2 and i1 are asserted, the output will be o1=0, o0=1.

**Do this:** Fill in a truth table for functions o1 and o0. Minimize them using K-maps and find the Boolean expressions for the minimum sum of products. Circle this final expression.
1.16 Combinational Logic

Above is a circuit that you are to re-engineer to try to make it faster.

a. In English, **what does this circuit do?** (Hint: To figure out what the circuit does, first figure out what each of the two parts of the circuit do.)

b. On the next page, **redesign** the circuit into standard minimized SOP (sum of products) form, and **draw** it using only AND, OR, and NOT gates.

c. Is the new circuit any faster than the old circuit? (**yes** or **no**)

1.17 Word Problem

In a digital logic simulator such as LogicWorks, the simulator must remember whether a signal has the value of logic 1, logic 0, X (also called “unknown”), and Z (also called high impedance). The software keeps track of this four-valued logic by using two bits to encode each signal bit as shown in the table below:

<table>
<thead>
<tr>
<th>value</th>
<th>2-bit encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>Z</td>
<td>11</td>
</tr>
</tbody>
</table>

Consider building a half adder that uses this encoding. For instance, using the above representation, if binary 1 and 0 were being added together, the inputs to the circuit would be a=01 and b=00. The result would be 01 (i.e. a 1). If, instead, we added a=10 (an unknown value X) and b=00, the result would be X; this would be represented as 10. The sum would be an X because input a was an X. That is, since we don’t know which value the input is, we don’t know what the sum output will be.

a. Let’s build the **carryout** portion of a half adder circuit that produces the carryout signal. The inputs are a1 a0 and b1 b0, and the output will be c1 c0, all as encoded above. There is no carry in.
As above, if there is an X (10) on the input, we don’t know what value it has — it may be either 0 or 1. Even though we don’t know, our circuit should still output an appropriate value. This value may be either a 0, 1, or X. That is, you may be able to determine that the output is a 0 or 1 even though there is an X on some of the inputs.

Assume that Z (11) on the input will be treated as an unknown (i.e. an X). For now, we’ll assume that our adder can’t output a Z — only 0, 1, or X.

**Do this:** Show the carryout functions for c1 and c0 in terms of a1, a0, b1, and b0 in the following Karnaugh maps. Write the minimized SOP for each bit.

b. There is another input to the circuit. This signal is the output enable signal (OE). When asserted high, OE causes the final output of our circuit to be represented as a Z (11); when not asserted, the final output is just the output of part a. Show a logic diagram for this part of the circuit — the part that has the two carryout bits (c1, and c0) from part a and OE as input, and cout1 and cout0 as the final outputs of our circuit.

![Karnaugh maps for part a](image)

1.18 PAL Stuff

The following functions each have A, B, C, and D as inputs. Implement them on the PAL circuit given. Do not add to the PAL circuit. Write the product terms used on the left. Put X’s in the circuit to show connections. Note that only 3 product terms are allowed per output.
E = Σm (3, 4, 9, 13)
F = Σm (0, 1, 3, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15)
G = Σm (0, 1, 2, 3, 4, 7, 9, 13)

1.19 PALs

Implement the following three functions using the PAL. Show dots for connections, and label any product or feedback terms. Throughout the diagram, make sure it is clear whether the true or complement values are to be used. Implement X on the leftmost output, Y on the middle, Z on the rightmost. The variable ordering on the minterms is A, B, C, D. Hint: a K-map or two might help.

X = A'B'CD + ABC'D
Y = Σ m (5, 10, 11, 13)
Z = Σ m (0, 1, 3, 4, 5, 6, 9, 11, 12, 14, 15)
1.20 Name that signal

The system below detects if it is possible to drive to school. The four boxes on the left are sensors that detect the indicated situation and produce Boolean values.

a. What are the assertion levels of the signals at A and B?

b. Write a sentence that starts “I’ll drive to school when ….” The phrase should indicate that you understand the logic of the situation.
I’ll drive to school when …
1.21 Lawrence Welk Logic Family

a. Write a sentence describing what logic function is implemented by this circuit. (Say something like “if a, b, c, d, e, and f are asserted but not g, h, i, and j, then k is asserted.”)

b. What is the assertion level of the output? Why?

![Logic Circuit Diagram]

1.22 Mixed Logic

You are trying to debug the following circuit:

![Logic Circuit Diagram]

What is the voltage on each signal when asserted? When will your logic probe read a logic 1 for each signal: when it is asserted or deasserted? Put your answers in the following table:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Voltage when asserted</th>
<th>Logic probe shows logic 1 when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.23 Circuit Timing

Consider the logic diagram below. Assume each gate has *delay 1* and each logic block has *delay 3*. Arrival times for each signal are shown using “@T” notation.

Trace the path from input to output that determines how long it takes the circuit to compute its slowest output.

At what time does this worst-case slowest output become valid?

1.24 Verilog Analysis

Consider the following K-map, which should look familiar from lecture. Your lab partner, Spock, claims that he can translate K-maps with four variables or less directly into accurate Verilog module descriptions. He takes one glance at the K-map and produces the following Verilog.

Spock claims that by executing the Verilog program, he can quickly determine whether his description matches the K-map. How will he do it? Your answer *must* relate the output of the Verilog simulation to a truth table description of the function *and* the minterms represented on the K-map. What output will the Verilog simulation produce if Spock is correct?
module hw2_ckt(f, a, b, c, d);
  output  f;
  input   a, b, c, d;

  and    (f3, f1, d),
          (f4, f2, b);
  xnor   (f1, a, c);
  not    (f2, f1);
  or     (f, f3, f4);
endmodule

module test_hw2_ckt(a, b, c, d, f_out);
  output a, b, c, d;
  reg    a, b, c, d;
  input  f_out;
  
  always begin
    $monitor($time, "a = %b, b = %b, c = %b, d = %b, f_out = %b", a, b, c, d, f_out);
    #5 a = 0; b = 0; c = 0; d = 0;
    #5 d = 1;
    #5 c = 1; d = 0;
    #5 d = 1;
    #5 b = 1; c = 0; d = 0;
    #5 d = 1;
    #5 c = 1; d = 0;
    #5 d = 1;
    #5 b = 1; c = 0; d = 0;
    #5 d = 1;
    #5 c = 1; d = 0;
    #5 d = 1;
    #5 b = 1; c = 0; d = 0;
endmodule
module system;
wire  F_Out, A_in, B_in, C_in, D_in;

hw2_ckt  My_Ckt(Ckt_Out, A_in, B_in, C_in, D_in);
test_hw2_ckt  Test_My_Ckt(A_in, B_in, C_in, D_in, Ckt_Out);

endmodule

1.25 Verilog Programming
Consider the following Verilog mux2to1 module which implements a 2:1 mux. It is similar to the version presented in lecture 6 slide 3.
a. Implement the following 4:1 mux circuit by writing a **Structural Model** in Verilog using the concept of **instantiation**. You can call this new module `mux4to1`. You **must** use the above `mux2to1` module three times to produce the new module. Moreover, you are not allowed to use any other gate-level primitives in writing your module.

![Diagram](image)

b. Complete the Verilog program by writing a **test module** called `test_mux4to1` and a **system module** called `testbench`. In the test module, simulate two sets of data inputs \((a, b, c, d) = (1, 0, 1, 0)\) and \((0, 0, 1, 1)\) with all possible combinations of select inputs to verify that your description is correct.
Answers

1.1 Implicants

a. An implicant is defined as either a single element of the ON-set or a group of elements that can be combined together in a K-map. Therefore, any single 1 or legal grouping of 1’s will be an implicant.

b. A prime implicant is defined as an implicant that cannot be combined with another implicant to eliminate a term. The implicants A’BC’, A’BD’, A’C’D, and B’D, can be classified as prime implicants. Although A’BC’D’ is an implicant, it is not considered prime because it can be combined with A’BCD’ to make A’BD’.

c. An essential prime implicant occurs when only a single prime implicant covers an element of the ON-set. This implicant is essential. The essential prime implicants in this example are: B’D and A’BD’. B’D is an essential prime because it is the only prime implicant to cover AB’CD. However, the prime implicant A’BC’ is not essential because it’s 1’s can be covered by the other prime implicants, A’BD’ and A’C’D.

The Karnaugh Map, and its implicants are shown in the figure below.

1.2 Simple Logic Function

With the information given, we can encode the twelve months into binary representation. The output F is a 1 when the month has 31 days, and a 0 when the month has any other number of days. We can now put these months and the output F into a truth table. F can be set as don’t-care for the four slots of the truth table without a month. Using these don’t cares, we are able to minimize the function into fewer terms. We can place the information contained in the truth table into a K-Map to find the function. The truth table and the K-Map are shown below.
1.3 Theorem Proving

- $X \cdot (X + Y) = X$  
  Given
- $X \cdot X + X \cdot Y = X$  
  8 (Distributive)
- $X + X \cdot Y = X$  
  3D (Idempotence)
- $X \cdot 1 + X \cdot Y = X$  
  1D (Operations with 1)
- $X \cdot (1 + Y) = X$  
  8 (Distributive)
- $X \cdot (Y + 1) = X$  
  6 (Commutative)
- $X \cdot (1) = X$  
  2 (Operations with 1)
- $X = X$  
  1D (Operations with 1)

1.4 Theorem Proving

- $(X \cdot Y') + Y = X + Y$  
  Given
- $(X \cdot Y') + Y = (X \cdot Y') + (X+1) \cdot Y$  
  2 (Operations with 1)
- $(X \cdot Y') + Y = (X \cdot Y') + X \cdot Y + 1 \cdot Y$  
  8 (Distributive)
- $(X \cdot Y') + Y = X \cdot (Y' + Y) + 1 \cdot Y$  
  8 (Distributive)
- $(X \cdot Y') + Y = X \cdot 1 + 1 \cdot Y$  
  5 (Complementary)
- $(X \cdot Y') + Y = X + Y$  
  1D (Operations with 1)
1.5 Canonical Forms

a. To find the function, we will start evaluating from the inputs and work our way over to F.

\[ F(A, B, C) = ABC' + BC' + AB'. \]

By expanding the function using the distributive law, we have:

\[ F(A, B, C) = ABC' + BC' + AB'. \]

Now we want to expand each term to use each variable of the function. For example for the term AC’, we know that it does not matter what the variable B is. Therefore, we can see that AC’ is equivalent to ABC’ + AB’C’. We can now expand the term AC’ to ABC’ + AB’C’. We can also do the same for BC’ and AB’. The new expanded function F is:

\[ F(A, B, C) = ABC + ABC + ABC + A'BC + AB'C' + AB'C + ABC + AB'C \]

We also know that the notation for minterms of three variables is:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A'B'C' = m_0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A'B'C = m_1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A'BC' = m_2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A'BC = m_3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB'C' = m_4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB'C = m_5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC' = m_6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC = m_7</td>
</tr>
</tbody>
</table>

Now, knowing the four terms of the function F(A, B, C) correspond to the minterms m_2, m_4, m_5, and m_6, we can now express the function in its canonical SOP form:

\[ F(A, B, C) = \Sigma m(2, 4, 5, 6) \]

b. By expanding the function using the distributive law, we have:

\[ F(A, B, C) = AC' + BC' + AB'. \]

F(A,B,C) = AC' + BC' + AB'.

Now we want to expand each term to use each variable of the function. For example for the term AC’, we know that it does not matter what the variable B is. Therefore, we can see that AC’ is equivalent to ABC’ + AB’C’. We can now expand the term AC’ to ABC’ + AB’C’. We can also do the same for BC’ and AB’. The new expanded function F is:

\[ F(A, B, C) = ABC + ABC + ABC + A'BC + AB'C' + AB'C + ABC + AB'C \]

Two terms are repeated, so we can simplify to F(A,B,C) = ABC + ABC + ABC + A'BC + AB'C

We also know that the notation for minterms of three variables is:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A'B'C' = m_0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A'B'C = m_1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A'BC' = m_2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A'BC = m_3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB'C' = m_4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB'C = m_5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC' = m_6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC = m_7</td>
</tr>
</tbody>
</table>

Now, knowing the four terms of the function F(A,B,C) correspond to the minterms m_2, m_4, m_5, and m_6, we can now express the function in its canonical SOP form:

\[ F(A, B, C) = \Sigma m(2, 4, 5, 6) \]

c. To find the maxterms of the function, we have to find the terms where F is equal to zero. These terms are the opposite of the minterms.

\[ F(A, B, C) = \Pi M(0, 1, 3, 7) \]
1.6 Canonical Forms

To solve this problem, we will use the same method used in the previous problem. First we will find the function $F$. Then, using the Boolean expression for $F$, we can fill in the truth table to determine the minterms and maxterms.

$$F = \Sigma m (1, 2, 4, 5, 6, 7)$$
$$F = \Pi M (0, 3)$$

Now we can find the canonical SOP and POS forms.

The SOP form is the combination of terms where $F = 1$.

The POS form is the combination of terms where $F = 0$.

$F = \Sigma m (1, 2, 4, 5, 6, 7)$
$F = \Pi M (0, 3)$

1.7 Canonical Forms of Combinational Logic

There are two ways to find the new expression for circuit in POS form. For the first method we will find the expression using the K-Map.

$F' = B'D + CD'$

Now we can use DeMorgan’s Law to put the expression in POS form.

$F'' = (BD' + CD')'$ Complement both sides
$F = ((B + D')' + (C' + D)')'$ DeMorgan’s Law on Inner terms
$F = (B + D')(C' + D)$ DeMorgan’s Law again
The second way to find the expression in POS form is to find the minimum cover for the 0’s directly on the K-Map. The minimum cover is shown below.

Covering the 0’s on the K-Map gives us the function \( F = (C' + D)(B + D') \)

The POS circuit for these equations would look like:

1.8 Factoring a Boolean Expression

The function \( f_c \) is the function \( f(a, b, c, d) \) with all \( c \)'s set to 1. The function \( f_{c'} \) is the function \( f(a, b, c, d) \) with all \( c \)'s set to 0.

- \( f_c = ab'(1)d' + a'b(0)d + a'b'(0)d + a'b'(1)d' = ab'd' + a'b'd' = b'd' \)
- \( f_{c'} = a'b'(0)d' + a'b(1)d + a'b'(1)d + a'b'(0)d' = a'bd + a'b'd = a'd \)

1.9 Implementation Methods

a. Implementing this function with a 16:1 multiplexor is quite straightforward. Since \( F \) is a function of 4 variables, and we have 4 select lines on a 16:1 multiplexor, we can use the mux as a lookup table. We put the variables A, B, C, and D on the select lines of the mux. Next, we can place the truth table row \( k \) value on the \( k \)-th mux input. Since the set of min-terms are at \( (0, 1, 5, 7, 8, 10, 11, \) and \( 15) \), we will connect these inputs to logic 1. The other inputs will be connected to logic 0. For example when the row for \( ABCD = 1010 \), \( 110 \) of the multiplexor will be selected.
b. Using an 8:1 multiplexor is just a little bit different. Now we only have 3 select lines to work with. By using A, B, and C as the select variables, we have to look at the function a little differently. By looking at the truth table, we can see that for the two slots where A, B, and C are the same, there can be four different values for F (0, 1, D, and D). We can place the appropriate value of F (0, 1, D, or D) on the mux inputs that correspond to the select values.

<table>
<thead>
<tr>
<th>A B C D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Truth Table Diagram](image-url)

1. By using A, B, and C as the select variables, the function can be looked at differently.
2. For the two slots where A, B, and C are the same, there can be four different values for F (0, 1, D, and D).
3. We can place the appropriate value of F (0, 1, D, or D) on the mux inputs that correspond to the select values.
c. To implement this function with a 4:16 decoder, we must remember that a decoder acts as a minterm generator by using its select lines to activate the minterm for the corresponding row of the truth table. By placing the input variables (A, B, C, and D) on the select lines, setting enable to 1, and connecting an OR gate to each output corresponding to a minterm, we have implemented the function.
1.10 Boolean Derivative

The 2-level form of the function $G = AC + BC + AB'C'$ is:

In order to find $dG/dC$, we first have to find $G_{c'}$ and $G_c$.

- To find $G_{c'}$, we replace all instances of $C$ with (0).
  
  $$G_{c'} = A(0) + B(0) + AB'(1) = AB'$$

- To find $G_c$, we replace all instances of $C$ with (1).
  
  $$G_c = A(1) + B(1) + AB'(0) = A + B$$

- Now, knowing $G_{c'}$ and $G_c$, we can solve for $dG/dC$.
  
  $$dG/dC = G_{c'} \text{xor} G_c$$

  $$= (A + B) \text{xor} AB'$$

  $$= (A + B)(AB')' + (A + B)'(AB')$$

  $$= (A + B)(A' + B) + (A'B')(AB')$$

  $$= AA' + AB + A'B + BB + 0$$

  $$= AB + A'B + B$$

  $$= B$$

- We now have determined that to allow a change on input $C$ to propagate to $G$, we need the input $B = 1$. The input $A$ can be either a 0 or a 1. The paths through the logic gates are shown below.
1.11 Write the Cofactor

a. The cofactor for the function $F_{xy}$ is the function $F(v, w, x, y, z)$ with all $x$’s set to 0, and all $y$’s set to 1.

$$F_{xy} = vw(1)z + v(1)z + (1)(1)z + (0)z + w(0)z = vwz + vz + z$$

b. Since the variable $z$ is in each term of the function, $F_z = 0$.

1.12 Quine - McCluskey

Finding an expression by using the Quine-McCluskey algorithm is as simple as following the steps learned in class.

a. Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1’s.

This step is quite straight-forward. In this example, column 1 is filled with the minterms and the don’t care terms.

<table>
<thead>
<tr>
<th>Column 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) 0010</td>
</tr>
<tr>
<td>(8) 1000</td>
</tr>
<tr>
<td>(3) 0011</td>
</tr>
<tr>
<td>(5) 0101</td>
</tr>
<tr>
<td>(7) 0111</td>
</tr>
<tr>
<td>(11) 1011</td>
</tr>
<tr>
<td>(13) 1101</td>
</tr>
<tr>
<td>(15) 1111</td>
</tr>
</tbody>
</table>

IMPLICATION TABLE
b. Step 2: Apply Uniting Theorem - Compare elements of group with N 1’s with elements of group with (N+1) 1’s. Difference by one bit implies adjacency. Eliminate variable and place in next column.

In this step, we will first look at the elements in the group of implicants with one 1, and compare them with the elements in the group with two 1’s. We can see that 0010 (from ones group) differs from 0011 (from twos group) by only the last bit. Therefore, we check these two off, and add the new 001- to the next column. We can continue this until we have checked all implicants. If an implicant is not eliminated, place a * next to it.

After the first column is completed, we can continue with the same step on the second column. We continue on the third column until no more combinations can be made. Here is the completed implication table.

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) 0010 3</td>
<td>(2,3) 001- *</td>
<td>(3,7,11,15) --11 *</td>
</tr>
<tr>
<td>(8) 1000 *</td>
<td>(3,7) 0-11 3</td>
<td>(5,7,13,15) -1-1 *</td>
</tr>
<tr>
<td>(3) 0011 3</td>
<td>(3, 11) -011 3</td>
<td></td>
</tr>
<tr>
<td>(5) 0101 3</td>
<td>(5,7) 01-1 3</td>
<td></td>
</tr>
<tr>
<td>(7) 0111 3</td>
<td>(5, 13) -101 3</td>
<td></td>
</tr>
<tr>
<td>(11) 1011 3</td>
<td>(7,15) -111 3</td>
<td></td>
</tr>
<tr>
<td>(13) 1101 3</td>
<td>(11,15) 1-11 3</td>
<td></td>
</tr>
<tr>
<td>(15) 1111 3</td>
<td>(13,15) 11-1 3</td>
<td></td>
</tr>
</tbody>
</table>

c. Now we have found all of the prime implicants. Next, we want to find the smallest set of prime implicants that cover the ON-set. We create a prime implicant chart. The rows of this chart are the prime implicants. The columns of the chart are the ON-set elements. An x represents a place where the ON-set element is covered by that prime implicant. We can take note that don’t-care terms do not appear in the ON-set column.

<table>
<thead>
<tr>
<th>PRIME IMPLICANT CHART</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>2,3 (001-)</td>
</tr>
<tr>
<td>3,7,11,15 (--11)</td>
</tr>
<tr>
<td>5,7,13,15 (-1-1)</td>
</tr>
<tr>
<td>8 (1000)</td>
</tr>
</tbody>
</table>

The first step with this chart, is to find all columns that have a single x. These implicants associated with these rows are essential implicants. They must appear in the minimum
cover. This step is show below. In our example, the implicants --11, -1-1, and 1000 are the essential implicants.

PRIME IMPLICANT CHART

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>11</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,3 (001-)</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,7,11,15 (--11)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5,7,13,15 (-1-1)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 (1000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

The next step is to eliminate all columns covered by essential primes. This would eliminate column 3, which has the last x. If any x’s remained at this stage, we would have to find the minimum set of rows that covered the remaining columns. These rows would be the final primary implicants. The implicants in this example are (--11, -1-1, and 1000) which correspond to the terms (CD, BD, and AB’C’D’)

Our final answer is: F = AB’C’D’ + CD + BD

1.13 Quine - McCluskey

This problem is solved in the exact same manner as the one above. The implication table for this problem looks like the following:

IMPLICATION TABLE

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) 0000 3</td>
<td>(0,2) 00-0 3</td>
<td>(0,2,8,10) -0-0*</td>
</tr>
<tr>
<td>(2) 0010 3</td>
<td>(0,8) -000 3</td>
<td>(8,9,12,13) 1-0-*</td>
</tr>
<tr>
<td>(8) 1000 3</td>
<td>(2,10) -010 3</td>
<td></td>
</tr>
<tr>
<td>(10) 1010 3</td>
<td>(8,10) 10-0 3</td>
<td></td>
</tr>
<tr>
<td>(12) 1100 3</td>
<td>(8,12) 1-00 3</td>
<td></td>
</tr>
<tr>
<td>(9) 1001 3</td>
<td>(8,9) 100- 3</td>
<td></td>
</tr>
<tr>
<td>(13) 1101 3</td>
<td>(12,13) 110- 3</td>
<td></td>
</tr>
<tr>
<td>(15) 1111 3</td>
<td>(9,13) 1-01 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(13,15) 11-1 *</td>
</tr>
</tbody>
</table>

Now, the prime implicant chart:
Again, if a column has a single x, then the implicant associated with that row is essential. The implicants -0-0 and 1-0- are essential. The remaining x’s in column 8, are removed because they are in the same rows as an essential prime. Our final implicants are -0-0 and 1-0-, which correspond to (B’D’ and AC’). Note that 13,15 is not included in the final cover. This is because both 13 and 15 are don’t-cares. Including this term would require an extra gate.

Our final answer is: \( F = AC' + B'D' \)

### 1.14 Combinational Logic Design

a. Minimal 2-Level Form

The first step to finding the minimal 2-Level form is to find the function \( F \). We will create a K-Map from the truth table.

<table>
<thead>
<tr>
<th>AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The minimal cover of the K-Map gives us the function: \( F = A'BC + AB' + AC' \)

Here is the 2-Level implementation of this function:
This design uses two 2-input gates and two 3-input gates, which costs a total of $6. However, using a little bit of Boolean algebra, we can minimize the expression using an xor.

\[
F = A'BC + AB' + AC' \quad \text{Original Expression}
\]

\[
F = A'BC + A(B' + C') \quad \text{Distributive Law}
\]

\[
F = A'BC + A(BC)' \quad \text{DeMorgan’s Law}
\]

\[
F = A \text{ xor } BC \quad \text{XOR Definition}
\]

Here is the 2-Level implementation using the xor’ed function.

\[
\begin{array}{ccc}
A & B & C \\
0 & 0 & 0 & F = 0 \\
0 & 0 & 1 & F = 0 \\
0 & 1 & 0 & F = 0 \\
0 & 1 & 1 & F = 1 \\
1 & 0 & 0 & F = 1 \\
1 & 0 & 1 & F = 1 \\
1 & 1 & 0 & F = 1 \\
1 & 1 & 1 & F = 0 \\
\end{array}
\]

The new design uses two 2-input gates, leaving a total cost of $2.

b. Mux-Logic Form

Implementing the function F with a 4:1 multiplexor can be done by placing two of the input variables as the inputs to the select lines. Then we can make each input to the mux, a function of the other variable. We will use the variables A and B on the select lines, and F will be a function of C. (Either B,C or A,C will work as well.)

Any of these designs use only one 4:1 mux, and inverters, so the total cost is $3.

c. Decoder-Logic Form

By setting enable on the decoder to logic 1, and connecting OR gates to the appropriate outputs, we can use the decoder as a minterm generator. Because F is a 1 with the minterms 3,4,5, and 6, we will connect the OR gates to the 3rd, 4th, 5th, and 6th output of the decoder.
Our implementation uses one 8:3 decoder, and three 2-input OR gates. Other combinations of OR gates can also be used. The total cost of this implementation is $6.

1.15 Word Problem

With the information given in the problem statement, we can create a truth table.

<table>
<thead>
<tr>
<th>i3 i2 i1 i0</th>
<th>o1 o2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>d d</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

The minterm and don’t care terms for the two outputs are:

\[ o0 = \Sigma m(2, 6, 8, 10, 14) + \Sigma d(0) \]
\[ o1 = \Sigma m(4, 8, 12) + \Sigma d(0) \]
Using the K-map to find the boolean expression for the minimum sum of products:

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>i0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>o1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The output expressions, from the K-Maps, are as follows:
- \( o_0 = i_1i_0' + i_2'i_0' \)
- \( o_1 = i_1'i_0' \)

### 1.16 Combinational Logic

a. The first box represents a multiplexor. A and B are the inputs, and S is the select line. By looking at the diagram, we can see that the function performed by this box is \( AS + BS' \), which is the same as the function of a multiplexor.

The second box represents a XOR with the output of the mux as one input and C as the other. Again, using F as the input from the mux, we have the function \( Z = FC' + F'C \), which is a XOR’s function.
b. To perform this step, we can first create a truth table to show $Z$ as a function of $A, B,$ and $S$. The truth table is then used to create a K-Map and find the minimum expression. $G$ lists the output of the mux.

\[
\begin{array}{cccc|c}
A & B & C & S & G \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Here is the minimized SOP form of the circuit.

\[
Z = BC'S' + AC'S + A'CS + B'CS'
\]

\[
\begin{array}{c|c|c|c|c}
& & & & \\
& 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 1 & 1 \\
01 & 1 & 0 & 1 & 0 \\
11 & 1 & 1 & 0 & 0 \\
10 & 0 & 1 & 0 & 1 \\
\end{array}
\]

c. This new circuit will be faster than the old circuit because of delays through all gates. For example in the old circuit, a change in $A$ or $B$ would have to propagate through 4 gates (not counting inverters) to be seen on the output $Z$. In the new circuit, any of the signals only have to pass through 2 gates to reach the output.

1.17 Word Problem

a. To solve this problem, the first thing that we must realize is that we are adding two things together to form the carry out. The things being added together can have one of four values (0, 1, x, and z). Consider all of the possible combinations.
The top left four entries are the normal carry out entries that we have seen before. The top row, and the left column are all 0’s. This is because when you are adding together two numbers, and one is a zero, there is no way that you can get a carry. The rest of the entries are all x. Either we are adding 1+x, in which case the carry out is unknown (it could be either a 1 or a 0), or we are adding x+x, where nothing is known. Now we can take these entries and encode them using two bits. The K-Maps for these are below:

The minimized expressions are:

- \( c_1 = a_1b_1 + a_1b_0 + a_0b_1 \)
- \( c_0 = a_1\bar{a}_0b_1\bar{b}_0 \)

b. For this part, we know that the outputs should both be one if OE is one. If OE is zero, the outputs will just be \( c_1 \) and \( c_0 \) as found in part a. A simple way to implement this is to put each output from part a (\( c_1 \) and \( c_0 \)) into one input of an OR gate, then put OE into the other input of the OR gate. This is shown below.
1.18 PAL Stuff

We are given the sets of minterms for each function. The first step we can take is to place these terms into a K-Map and find the minimum boolean expression for each term. The functions for E and F can easily be done with 3 terms, however, for G, we must use a different method. G looks like it will need at least 4 terms, but we see that the ON-set for E fits into G also. Knowing that we can use feedback in our PAL, we can give G in terms of E. Also note that it is easier to implement $\overline{F}$ than $F$.

<table>
<thead>
<tr>
<th>E</th>
<th>AB</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 01 11 10</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th>AB</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 01 11 10</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The K-Map for G contains E, with a few extra 1’s. We can use the feedback variable for E, then just cover the remaining 1’s.

<table>
<thead>
<tr>
<th>F</th>
<th>AB</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 01 11 10</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Our functions are as follows:
\[ E = A'BC'D' + AC'D + A'B'CD \]
\[ F' = A'BC'D' + ABC'D + A'B'CD' \]
\[ G = E + A'B' + A'CD \]

We now have each function in 3 terms or less. Now we can make the appropriate connections in the PAL.

1.19 PALs

Again, we are given the minterms for two of the functions, and we are given the actual boolean expression for \( X \). We want to first find the Boolean expressions for \( Y \) and \( Z \) by creating a K-Map for each. We will find \( Z' \) instead of \( Z \).

We find that:
\[ Y = AC'B' + BC'D \]
\[ Z' = AB'D' + B'CD' + A'BC'D + ABC'D, \]
but the last two terms of $Z'$ is just $X$. Therefore, we can minimize $Z'$ to

$$Z' = AB'D' + B'CD' + X$$

Since we now have $X$, $Y$, and $Z'$ all in 3 terms or less, we can make the connections on the PAL.
1.20 Name that Signal

When deciding the assertion level of a signal, always remember that assertion levels match bubbles. If an output of a gate has a bubble, that output signal is asserted low. If there is no bubble, then that signal is asserted high.

a. A is asserted high. B is asserted low because of the bubble on the output of the gate.

b. Here, don’t forget to cancel out the bubbles. When doing this, always make sure there are no mismatches due to the signal fanout. The sentence would read as follows:

I’ll drive to school when... my keys are in my pocket, and the car is in the garage, and it’s either colder than 40 or the clouds are dripping (or both).

1.21 Lawrence Welk Logic Family

a. If m is asserted or both p and t are asserted, then y is asserted.

b. The assertion level of the output is high because there is no bubble on the output of the gate.
1.22 Mixed Logic

The important point to remember here is that when a signal is asserted high, a logic 1 indicates the event. When a signal is asserted low, a logic 0 indicates the event.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Voltage when asserted</th>
<th>Logic probe shows logic 1 when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.H</td>
<td>logic 1</td>
<td>asserted</td>
</tr>
<tr>
<td>B.L</td>
<td>logic 0</td>
<td>deasserted</td>
</tr>
<tr>
<td>C.L</td>
<td>logic 0</td>
<td>deasserted</td>
</tr>
</tbody>
</table>

1.23 Circuit Timing

For any gate or module, the time where its outputs are valid is the latest time an input becomes valid, plus the delay of the gate or module. We can find the worst case valid output by tracing from input to output. The arrival times are shown for each signal at the inputs to the gates. The worst-case slowest output, J, becomes valid @10.
1.24 Verilog Analysis

Consider the following K-map, which should look familiar from lecture. Your lab partner, Spock, claims that he can translate K-maps with four variables or less directly into accurate Verilog module descriptions. He takes one glance at the K-map and produces the following Verilog.

Spock claims that by executing the Verilog program, he can quickly determine whether his description matches the K-map. How will he do it? Your answer must relate the output of the Verilog simulation to a truth table description of the function and the minterms represented on the K-map. What output will the Verilog simulation produce if Spock is correct?

```
module hw2_ckt(f, a, b, c, d);
  output f;
  input a, b, c, d;

  and (f3, f1, d),
       (f4, f2, b);
  xnor (f1, a, c);
  not (f2, f1);
  or (f, f3, f4);
endmodule

module test_hw2_ckt(a, b, c, d, f_out);
  output a, b, c, d;
  reg a, b, c, d;
  input f_out;

  always begin
    $monitor($time, "a = %b, b = %b, c = %b, d = %b, f_out = %b", a, b, c, d, f_out);
    #5 a = 0; b = 0; c = 0; d = 0;
  end
endmodule
```
module system;
wire  F_Out, A_in, B_in, C_in, D_in;

hw2_ckt     My_Ckt (Ckt_Out, A_in, B_in, C_in, D_in);
test_hw2_ckt Test_My_Ckt (A_in, B_in, C_in, D_in, Ckt_Out);
endmodule

If Spock is correct, the output will look like the following:

0a = x, b = x, c = x, d = x, f_out = x
5a = 0, b = 0, c = 0, d = 0, f_out = 0
10a = 0, b = 0, c = 0, d = 1, f_out = 1
15a = 0, b = 0, c = 1, d = 0, f_out = 0
20a = 0, b = 0, c = 1, d = 1, f_out = 0
25a = 0, b = 1, c = 0, d = 0, f_out = 0
30a = 0, b = 1, c = 0, d = 1, f_out = 1
35a = 0, b = 1, c = 1, d = 0, f_out = 1
40a = 0, b = 1, c = 1, d = 1, f_out = 1
45a = 1, b = 0, c = 0, d = 0, f_out = 0
50a = 1, b = 0, c = 0, d = 1, f_out = 0
55a = 1, b = 0, c = 1, d = 0, f_out = 0
60a = 1, b = 0, c = 1, d = 1, f_out = 1
65a = 1, b = 1, c = 0, d = 0, f_out = 1
70a = 1, b = 1, c = 0, d = 1, f_out = 1
75a = 1, b = 1, c = 1, d = 0, f_out = 0
80a = 1, b = 1, c = 1, d = 1, f_out = 1

Basically, Spock is monitoring the output of his logic for different combinations of inputs in the Verilog simulation. At every 5 time units, Spock sets the inputs of the logic corresponding to one row of the truth table. He is lazy (and smart) in writing the Verilog code. He changes only the inputs that are necessary to change every 5 time units. For those
inputs that he does not change, they keep the same values. For example, at time 5, the inputs are (0, 0, 0, 0). At time 10, he only changes d to 1. So, the inputs become (0, 0, 0, 1). Thus, the simulation result is actually a truth table description of the function he created.

Since each square on the K-map corresponds to a minterm, he can verify his design by checking whether f_out is 1 only when the inputs of the circuit corresponds to one of the minterms in the on-set of the K-map. In addition, since the gates are implemented with no delays, the circuit does not have any timing problem as it happened in homework 1.

1.25 Verilog Programming

Consider the following Verilog `mux2to1` module which implements a 2:1 mux. It is similar to the version presented in lecture 6 slide 3.

```
module mux2to1 (f, a, b, sel);
    output f;
    input a, b, sel;

    and g1(f1, a, nsel),
        g2(f2, b, sel);
    or  g3(f, f1, f2);
    not g4(nsel, sel);
endmodule
```
a. Implement the following 4:1 mux circuit by writing a **Structural Model** in Verilog using the concept of **instantiation**. You can call this new module `mux4to1`. You **must** use the above `mux2to1` module three times to produce the new module. Moreover, you are not allowed to use any other gate-level primitives in writing your module.

![MUX Circuit Diagram]

The new module should look like the following. It is similar to the `mux2to1` module except it instantiates a previously defined module, instead of primitive gates. Because of this, you should explicitly declare *wires* which connect module ports. (However, Verilog will let you get away without the declaration for wires of width 1.) In `mux2to1` module, you do not need to do that since gate instantiations implicitly declare wires for their outputs.

```verilog
module mux4to1 (f, a, b, c, d, sel0, sel1);
  output f;
  input a, b, c, d, sel0, sel1;
  wire w1, w2;
  mux2to1 m1(w1, a, b, sel0),
         m2(w2, c, d, sel0),
         m3(f, w1, w2, sel1);
endmodule
```

In order to verify that the Verilog module really describes the above circuit correctly, we
label the circuit diagram using the instance names we used in the module:

Notice that the way to feed signals to inputs of muxes is different from primitive gates (e.g. 2-input AND gates) in the way that the order of input lines do matter. When you describe the structural connections of m3, for instance, the order of w1 and w2 affects the functionality of the module significantly!

b. Complete the Verilog program by writing a test module called test_mux4to1 and a system module called testbench. In the test module, simulate two sets of data inputs \((a, b, c, d) = (1, 0, 1, 0)\) and \((0, 0, 1, 1)\) with all possible combinations of select inputs to verify that your description is correct. Hand in both source code for the above test and system modules as well as the simulation output.

The test module is a Behavioral Model which generates the specified input combinations and displays changes in output.
The system module is a simple **Structural Model** which connects the design (**mux4to1**) and the test modules (**test_mux4to1**) together. It does not have any inputs and outputs. Make sure the select lines are correctly connected.

```verilog
module test_mux4to1 (a, b, c, d, sel0, sel1, f);
    input f;
    output a, b, c, d, sel0, sel1;
    reg a, b, c, d, sel0, sel1;

    initial begin
        $monitor($time,, "a = %b, b = %b, c = %b, d = %b, sel1 = %b, sel0 = %b, f = %b",
            a, b, c, d, sel1, sel0, f);
        a = 1; b = 0; c = 1; d = 0;
        sel1 = 0; sel0 = 0;
        #10 sel1 = 0; sel0 = 1;
        #10 sel1 = 1; sel0 = 0;
        #10 sel1 = 1; sel0 = 1;

        #10 a = 0; b = 0; c = 1; d = 1;
        sel1 = 0; sel0 = 0;
        #10 sel1 = 0; sel0 = 1;
        #10 sel1 = 1; sel0 = 0;
        #10 sel1 = 1; sel0 = 1;

        #10 $finish;
    end
endmodule
```

The simulation output should look similar to the following. The first line is formatted to improve readability.

```verilog
module testbench;
    wire a, b, c, d, sel0, sel1, f;

    test_mux4to1 my_tester(a, b, c, d, sel0, sel1, f);
    mux4to1 my_design(f, a, b, c, d, sel0, sel1);
endmodule
```
<table>
<thead>
<tr>
<th></th>
<th>a = 1, b = 0, c = 1, d = 0, sel1 = 0, sel0 = 0, f = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>a = 1, b = 0, c = 1, d = 0, sel1 = 0, sel0 = 1, f = 0</td>
</tr>
<tr>
<td>20</td>
<td>a = 1, b = 0, c = 1, d = 0, sel1 = 1, sel0 = 0, f = 1</td>
</tr>
<tr>
<td>30</td>
<td>a = 1, b = 0, c = 1, d = 0, sel1 = 1, sel0 = 1, f = 0</td>
</tr>
<tr>
<td>40</td>
<td>a = 0, b = 0, c = 1, d = 1, sel1 = 0, sel0 = 0, f = 0</td>
</tr>
<tr>
<td>50</td>
<td>a = 0, b = 0, c = 1, d = 1, sel1 = 0, sel0 = 1, f = 0</td>
</tr>
<tr>
<td>60</td>
<td>a = 0, b = 0, c = 1, d = 1, sel1 = 1, sel0 = 0, f = 1</td>
</tr>
<tr>
<td>70</td>
<td>a = 0, b = 0, c = 1, d = 1, sel1 = 1, sel0 = 1, f = 1</td>
</tr>
</tbody>
</table>