Problem 1: CLD2 Problems.

(a) 4.3

The functions for the 7-segment display decoder given in Section 4.3 are:

\[
\begin{align*}
C_0 &= A + BD + C + \overline{BD} \\
C_1 &= A + \overline{CD} + CD + \overline{B} \\
C_2 &= A + B + \overline{C} + D \\
C_3 &= \overline{BD} + C \overline{D} + B\overline{CD} + \overline{BC} \\
C_4 &= \overline{BD} + CD \\
C_5 &= A + \overline{CD} + B\overline{D} + BC \\
C_6 &= A + C\overline{D} + B\overline{C} + \overline{BC}
\end{align*}
\]

Refer to Figures 4.8 -4.10 in CLD2, which show the block diagram and internals of a ROM. A ROM is basically a hard-wired truth table, where the inputs are the Address bits and the outputs are the Data bits. There are 4 inputs (A, B, C, D), so the ROM will require a minimum of 4 address bits. Each one of the functions (C₀-C₆) corresponds to a separate bit line, meaning it is one bit of the Data output. 7 functions means we need 7 output bits, giving us 7 bits per word.

To program the ROM, we use the truth table for these functions given in Figure 2.33 in CLD2. Each 0 in the truth table will correspond to a pull-down transistor in the ROM. The programmed ROM is shown in Figure 1 below. A red circle at the intersection of a word line and a bit line indicates that a pull-down transistor should be placed in that location. Each of the bit lines is tied to logic 1 (as shown in CLD2 Figure 4.9), and a bit line will be pulled to 0 if the word line that is asserted activates a pull-down on that bit line. For example, if the input is ABCD = 0001, the word line 0001 is asserted. All the pull-down transistors driven by that word line are turned on, which pulls the bit lines C₀, C₃, C₄, C₅, and C₆ to 0. The Data output is thus 011000, which matches the 0001 row of the truth table.

The personality matrix for the ROM is shown in Figure 2. Note that since a ROM uses a decoder it creates all the minterms for a 4-input function.
Figure 1: ROM implementation of 7-seg display functions

<table>
<thead>
<tr>
<th>Product Term (minterm)</th>
<th>Inputs ABCD</th>
<th>Outputs C₀C₁C₂C₃C₄C₅C₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>A'B'C'D'</td>
<td>0000</td>
<td>1111110</td>
</tr>
<tr>
<td>A'B'C'D</td>
<td>0001</td>
<td>0110000</td>
</tr>
<tr>
<td>A'B'CD'</td>
<td>0010</td>
<td>1101101</td>
</tr>
<tr>
<td>A'B'CD</td>
<td>0011</td>
<td>1111001</td>
</tr>
<tr>
<td>A'BC'D'</td>
<td>0100</td>
<td>0110011</td>
</tr>
<tr>
<td>A'BC'D</td>
<td>0101</td>
<td>1011011</td>
</tr>
<tr>
<td>A'BCD'</td>
<td>0110</td>
<td>1011111</td>
</tr>
<tr>
<td>A'BCD</td>
<td>0111</td>
<td>1100000</td>
</tr>
<tr>
<td>AB'C'D</td>
<td>1000</td>
<td>1111111</td>
</tr>
<tr>
<td>AB'C'D</td>
<td>1001</td>
<td>1110011</td>
</tr>
<tr>
<td>AB'CD'</td>
<td>1010</td>
<td>1111111</td>
</tr>
<tr>
<td>AB'CD</td>
<td>1011</td>
<td>1111111</td>
</tr>
<tr>
<td>ABC'D</td>
<td>1100</td>
<td>1111111</td>
</tr>
<tr>
<td>ABC'D</td>
<td>1101</td>
<td>1111111</td>
</tr>
<tr>
<td>ABCD'</td>
<td>1110</td>
<td>1111111</td>
</tr>
<tr>
<td>ABCD</td>
<td>1111</td>
<td>1111111</td>
</tr>
</tbody>
</table>

Figure 2: Personality Matrix
(b) 4.8
(a) A multiplexer with $n$ control bits takes $2^n$ inputs, and has 1 output. The output is connected to the input whose binary code matches the control bits. A demultiplexer takes a single input and has $2^n$ outputs. Based on the binary value of the $n$ control bits, it will pass the input value into the output bit whose binary code matches the control bits. A decoder is the same as a demultiplexer except that in general the input bit is viewed more as an enable signal in this case.
(b) The function below implements a 2:4 demultiplexer.

(c) 4.10
Start by making a truth table for the function $F$, shown in Figure 3.
(a) We use AB as the select bits for the 4:1 mux. Thus we have 4 functions of the 3 variables CDE: \( F_{00} \), \( F_{01} \), \( F_{10} \), and \( F_{11} \), where the subscripts show the value of the select bits AB. Use a K-map to find a minimal expression for each function of 3 variables. The following minimal expressions are obtained:

\[
\begin{align*}
F_{00} &= D + CE \\
F_{01} &= \overline{C} + \overline{D} \\
F_{10} &= 1 \\
F_{11} &= 1
\end{align*}
\]

\( F_{00} \) is the input logic for the input of the mux corresponding to AB=00, and so on for each of the 4 expressions. The complete implementation is shown in Figure 4.
(b) Now we’ll use B and C as the control bits for the 4:1 mux. Since F is true anytime A is true, we can simplify the circuitry by OR’ing the output of the mux with A to get the final output F. Then we only have to find expressions $F_{00}$ – $F_{11}$ as a function of two variables, D and E. To do this, use the portion of the truth table where A=0, and separate it again into 4 sections: BC=00, BC=01, BC=10, and BC=11. We get:

$$F_{00} = D + CE$$
$$F_{01} = \overline{C} + D$$
$$F_{10} = 1$$
$$F_{11} = 1$$

(c) Using the same process as we did in part (b), we get the following expressions and circuit:

$$F_{00} = CE$$
$$F_{01} = 1$$
$$F_{10} = 1$$
$$F_{11} = \overline{C}$$
(d) Again using the same process:

\[ F_{00} = B \]
\[ F_{01} = 1 \]
\[ F_{10} = B + E \]
\[ F_{11} = \overline{B} \]

The implementations in parts b-d each use 2 gates and 1 inverter. This is significantly less than the implementation in part a, which required 3 gates and 2 inverters. Since \( F \) is true whenever \( A \) is true, using \( A \) as a control bit means two of the inputs to the mux are tied directly high. That only leaves two inputs with which to implement the rest of the functionality. Using the OR gate at the output of the mux is much more efficient for this function.

(d) 9.17

The number of gates that are “replaced” by any programmable logic chip completely depends on the design. It’s entirely possible that the 4-input function is something very simple – i.e., \( AB + CD \) – and would only take a few gates to implement. The equivalent gate count is largely just a marketing term used by the FPGA sales guys to convince you that their parts will save you money. With that in mind, they would probably give you the worst-case numbers. So what 4-input function requires the highest number of 2-input gates to implement? (Assume you can only use 2-input AND, OR, NAND, and NOR gates to implement the function – no 2-input XOR gates can be used, since it takes multiple gates to make an XOR.)

It turns out that the worst-case function is \( A \oplus B \oplus C \oplus D \). Since we can’t use XOR gates, we’ll need to implement this with AND and OR gates. Let’s look at a single XOR gate first. \( A \oplus B = A\overline{B} + \overline{A}B \), which requires two 2-input
AND gates and one 2-input OR gate. Thus we need three 2-input gates per XOR gate. Three XOR gates is therefore equivalent to 9 2-input gates, so the function generator is equivalent to 9 gates.

**9.18 (e)**

(a) Apologies from the course staff... There is actually an error in the book, and you can’t implement a 25 input parity function with a two-level LUT. If you had a five input LUT, you could implement this as follows:

A LUT (Look Up Table) stores the output value of a function for each input combination in a table. A two-level structure of 5-input LUTs can implement a 25-input parity function as shown in Figure 8. The block diagram for a 5-input parity function is shown on the left, where CL indicates combinational logic inside the block that implements the parity function. On the right, 5 of the 5-input parity function blocks make up the first level of the 25-input parity function. Those outputs are fed into another 5-input parity block to give the final output.

![Figure 8: 25-input parity function](image)

With a 4-input LUT as stated in the problem, you could implement a 16 input parity function with a two-level implementation.

(b) A single 4-input LUT can implement $4^1 = 4$ input parity. In part (a) we saw that a two level structure can do $4^2 = 16$ input parity. Similarly, a three-level LUT structure could implement a $4^3 = 64$ input parity function.

With 5-input LUTs, a three level implementation would give you $5^3 = 125$ input parity.

**9.19 (f)**

(a) The highest order bit (in this case, A) is used as the select bit of the 2:1 MUX. The two 8:1 multiplexers select the value of BCD, and their outputs feed the inputs of the 2:1 mux. This is shown below in Figure 9.
(b) To implement $F = A \oplus B \oplus C \oplus D$, we need to draw the truth table (see Figure 10 below). For each place where we find $F=1$ in the truth table, the corresponding input would be tied to $V_{DD}$. For $F=0$, tie the corresponding input to ground. The column in the truth table labeled $A=0$ goes with inputs 0-7, and the column labeled $A=1$ goes with inputs 8-15.

<table>
<thead>
<tr>
<th>BCD</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A=0$</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 10: 9.19 part b, input settings for $F = A \text{ xor } B \text{ xor } C \text{ xor } D$

(c) See the truth table in Figure 11. Again, $A=0$ corresponds to the top 8:1 MUX in Figure 9, while $A=1$ corresponds to the bottom 8:1 MUX.
Problem 2: Atmel ATF16V8B datasheet

a) The security fuse prevents someone from changing how the device is programmed. It also prevents reverse engineering by not allowing someone to see how the internals of the device are programmed.

b) The device can be operated in the following 3 modes. The maximum number of inputs, outputs, and flip-flops for each mode are listed below.
- **Registered mode**: 8 inputs (pins 2-9), 8 outputs (pins 12-19), 8 flip-flops (1 per output logic block)
- **Complex mode**: 10 inputs (pins 1-9, 11), 8 outputs (pins 12-19), 0 flip-flops
- **Simple mode**: 16 inputs (pins 1-9, 11-14, 17-19), 8 outputs (pins 12-19), 0 flip-flops. Note that pins 12-14 and 17-19 can inputs OR outputs, but not both simultaneously. Thus, you cannot get the maximum 16 inputs and 8 outputs with a single configuration.

c) A maximum of 8 product terms (implicants) can be used in the next state logic, and each product term can have a maximum of 16 literals. (This would mean we are in Simple Mode, using 16 inputs.)

d) Input to non-registered output: t_{PD} = 10ns max.
   Input to registered output: t_s + t_{CO} = 7.5ns + 7ns = 14.5ns
   To determine the maximum frequency of an FSM implemented using this part, consider that the inputs go through the next state logic (10ns), then the state register (14.5ns) to the output. The total time from input to output is 24.5ns. If the clock period was shorter than 24.5ns, then the output of the state register (current state) might not make it through the next state logic and to the state register before the next rising edge. Thus the maximum clock frequency for a FSM is 1/24.5ns = 40.8 MHz.

e) The power consumption at 40.8MHz, 5V, 25C can be found by looking at the Test Characterization Data section. We can get the supply current from the graph of Supply Current vs Input Frequency for the ATF16V8B device. At 40MHz, the device draws ~55mA from a 5V supply. Power = Current*Voltage (P = IV), so (5V)*(55mA) = 275mW.

Problem 3: Implement Fig. 7.15 and Fig. 7.17 using ATF16V8B logic diagram.

To implement the 163 counter using the ATF16V8B logic diagram, we need to find the logic expression for the next states QA+, QB+, QC+, QD+, and the output RCO in
terms of the current states QA, QB, QC, QD, and the inputs ABCD, P, T, /LOAD, and /CLEAR.

Rather than making a truth table with 12 inputs, we need to simplify the next state logic by recognizing the function of the control inputs:

<table>
<thead>
<tr>
<th>PT</th>
<th>/LOAD</th>
<th>/CLR</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>Clear</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Load</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Inhibit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Count</td>
</tr>
</tbody>
</table>

Figure 12: Control signals for 163

(For the 4-bit counter, we’ll treat PT as a single enable input. Also assume that this is a clear-dominant device.)

**Clear mode:** Set all outputs to 0.

**Load mode:** Set QA = A, QB = B, QC = C, and QD = D. RCO = 0.

**Count mode:** The next state functions of QA*, QB*, QC*, QD* can be found using a 4-variable K-map. We find that:

\[
Q_A^* = \overline{Q_A} \overline{Q_B} Q_C Q_D + Q_A \overline{Q_D} + Q_A \overline{Q_B} + Q_A \overline{Q_C}
\]

\[
Q_B^* = Q_B \overline{Q_C} + Q_B \overline{Q_D} + \overline{Q_B} Q_C Q_D
\]

\[
Q_C^* = Q_C \overline{Q_D} + \overline{Q_C} Q_D
\]

\[
Q_D^* = \overline{Q_D}
\]

And to make RCO go high during the same clock cycle that the output is 1111:

\[
RCO = Q_A Q_B Q_C Q_D
\]

Implement each of the expressions above, including PT, /LOAD, and /CLR in each product term.

**Inhibit mode:** This is essentially a “hold” mode; QA* = QA, etc.

To implement this functionality using the ATF16V8B logic diagram, label the input pins and output pins, and indicate which bits in the AND plane should be connected to create the product terms for each output. Each horizontal line going into an AND gate represents a product term. See Figure 13.

The 8-bit counter shown in Figure 7.17 of CLD2 cannot be implemented with the ATF16V8B – it requires at least 8 inputs and 8 state bits, which does not leave enough pins for the control inputs.
Figure 13: 4-bit counter implementation
Problem 4:
Compare the number of inputs, outputs, and flip-flops in an ATF16V8B chip vs. a single Xilinx Virtex-II CLB:

<table>
<thead>
<tr>
<th></th>
<th>ATF16V8B</th>
<th>Virtex-II CLB</th>
<th>Virtex-II CLB from CLD2 (WRONG!)</th>
</tr>
</thead>
<tbody>
<tr>
<td># inputs</td>
<td>16 (max)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td># outputs</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td># flip flops</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 14: Comparison of ATF16V8B and Virtex CLB

NOTE: The description of the CLB in CLD2 is wrong! There are 2 slices per CLB, not 4 as it says in the text.

In addition to the number of inputs and outputs, another important function to consider is the ability to use the outputs as feedback inputs. This feature on the ATF16V8B allows you to feed your current state (D-FF output) back into the next state logic! This is a very powerful feature. As a stand-alone PLD, the Atmel part is more powerful than a single CLB. However, the CLB has features such as the SOPIn/SOPOut and the Cin/COut make it easily expandable, which makes it more useful for large systems.

Problem 5:
The block diagram for a FSM implemented using a ROM and a register looks like this:

Figure 15: ROM + FF implementation of a FSM
The ROM implements the Next State logic. (Remember, a ROM is just like a hard-wired truth table!) The number of address bits for the ROM is the sum of the inputs and state bits \( (n = n_i + n_s) \). The number of bits per word is the sum of the state bits and the output bits \( (k = n_o + n_s) \). For a Moore machine, \( n_{o,Moore} = n_s = k \) since the output is only a function of the state.

For the 4-bit counter in problem 3, there are 8 inputs and 4 state bits. This FSM would require a ROM with \( 8 + 4 = 12 \) address bits and a 4 bit wide register.

For the 8-bit counter, there are 3 common input signals: P, /CLR, and /LOAD. Thus there are 13 inputs and 8 state bits. This FSM would require a ROM with 21 address bits and an 8 bit wide register.