February 11, 2019

Only Problem Set Part B will be graded. Turn in only Problem Set Part B which will be due on February 26, 2019 (Tuesday) at 3:00pm.

Submit homework via gradescope <gradescope.com> as a PDF. Other formats may or may not be viewed and evaluated accurately.

1 Problem Set Part A

All questions in this part are from Roth&Kinney, 7th Edition.

- 7.1, 7.4, 7.5, 7.6, 7.8, 7.9, 7.10, 7.11, 7.14, 7.26, 7.27, 7.28, 7.41, 7.42
- 8.2, 8.6, 8.7, 8.8
- 13.3(a)(b), 13.4(a), 13.8(a), 13.11(a)
- 14.4, 14.12, 14.13, 14.17, 14.23
2 Problem Set Part B

I. (Taming Multiple Output Functions)

In your book, you saw some examples of how we can optimize the implementation of multiple output functions by maximizing the number of prime implicants that are shared between the two functions, thereby reducing the total number of logic gates.

You might remember that for this, we need to see “essentiality” of the prime implicants in the context of multiple functions. Another thing we need to consider is that in certain cases, including a sub-prime implicant in the final cover may actually save us some gates.

In this question, we try to develop an algorithm to find out the optimal implementation for multiple output functions in terms of minimizing the number of gates. We have learned how Quine McCluskey works on a single function and this question will help you explore how this method can be extended to multiple output functions, perhaps with some considered modifications in the setup stage, which we introduce below.

Firstly, we start by realizing that the “shared” implicants between two functions $f_1$ and $f_2$ are nothing but the prime implicants of $f_1 \cdot f_2$. This leads us to deduce that for obtaining an optimal cover for $f_1$ and $f_2$ together, we need to consider the prime implicants of $f_1$, $f_2$ and $f_1 \cdot f_2$.

The next step is to set up a prime implicant chart which would list all the prime implicants of $f_1$, $f_2$ and $f_1 \cdot f_2$. Since we are dealing with two functions now, the columns will contain minterms of $f_1$, followed by minterms of $f_2$. The rows would contain prime implicants from $f_1$, $f_2$, $f_1 \cdot f_2$.

In each of the parts of this question, you would work with the modified prime implicant chart described above. Since we list minterms of both $f_1$ and $f_2$, it gives us two sub-charts which we separate with a double line to help you visualize things clearly. As $f_1$ and $f_2$ may have common minterms, we duplicate them in each sub-chart. Throughout, we will call the prime implicants of only $f_1$ as $P_i$’s, of only $f_2$ as $Q_i$’s, and the implicants shared between $f_1$ and $f_2$, i.e. the prime implicants of $f_1 \cdot f_2$, as $R_i$’s.

The hope is that with the prime implicant chart set up properly, we can go ahead and apply the Quine McCluskey method for reduction on the chart. This question invites you to consider the ramifications needed to possibly make this proposal work out for the case of multiple output functions.
(Part A) Below are given K-maps for 2 functions $f_1$ and $f_2$, and an outline of the combined prime implicant chart. As you can see, the chart now has 2 sub-charts, left for $f_1$ and right for $f_2$.

While placing X’s in this chart for each $P_i$ and $Q_i$ (i.e. the implicants belonging to $f_1$ and $f_2$, respectively), how would you treat the minterms that are common between $f_1$ and $f_2$ and therefore duplicated (e.g. $m_{15}$)? Would you place an X in both sub-charts for a common minterm, or only in one sub-chart? Give a reasoning for your answer.

\[
\begin{array}{cccc}
xy \backslash w & 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 & 0 \\
01 & 1 & 1 & 1 & 0 \\
11 & 0 & 1 & 1 & 0 \\
10 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
xy \backslash w & 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 & 0 \\
01 & 0 & 0 & 0 & 1 \\
11 & 0 & 0 & 1 & 1 \\
10 & 0 & 0 & 0 & 1 \\
\end{array}
\]
(Part B) Having figured out how to lay out the prime implicants that belong to only one of the functions, we are now ready to think how to treat the “shared” implicants.

Below are given K-maps for 2 functions $g_1 \cdot g_2$ as well as the K-map for $g_1 \cdot g_2$. As before, let us call the prime implicants exclusive to either function (i.e. appearing only in either $g_1$ or $g_2$ and not in $g_1 \cdot g_2$) as \( P_i \)'s and \( Q_i \)'s. The shared implicants, i.e. prime implicants of $g_1 \cdot g_2$ (which could turn out to be sub-prime to $g_1$ or $g_2$) are called \( R_i \)'s.

Now, while placing X’s in this chart for each \( R_i \), how would you treat the minterms that are common between $g_1$ and $g_2$? Would you place an X on both sides for a common minterm, or only on one side of the chart? Give a reasoning for your answer.
(Part C) Having the basics sorted out, now we are ready to find the optimal implementation of multiple output functions using our algorithm. For the K-maps below, use your learnings in (Part A) and (Part B) to fill out the given prime implicant chart, clearly naming the implicants as $P_i$ (for the K-map on the left), $Q_i$ (for the K-map on the right) and $R_i$. Then reduce it using the Quine McCluskey method. Does it give us the optimal implementation?

**Note:** The blank chart given below may contain more number of rows and columns than needed. Also be sure to show which sub-chart is for which function and separate the sub-charts with a double line.

\[
\begin{array}{cccc}
xy \backslash wz & 00 & 01 & 11 & 10 \\
00 & 0 & 1 & 0 & 0 \\
01 & 0 & 1 & 1 & 1 \\
11 & 0 & 1 & 1 & 0 \\
10 & 0 & 0 & 0 & 0 \\
\end{array}
\]  

\[
\begin{array}{cccc}
xy \backslash wz & 00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 & 0 \\
01 & 0 & 0 & 1 & 0 \\
11 & 0 & 1 & 1 & 1 \\
10 & 0 & 0 & 0 & 1 \\
\end{array}
\]
(Part D) Now we want to find the implementation again, this time by using Petrick’s method instead of the Quine McCluskey method.

(i). Let us think about finding the logic function for the optimal multiple output implementation of \( g_1 \) and \( g_2 \) together. While forming the logic function \( P \), should you consider the duplicate minterms (i.e. minterms present for both the functions), only once or twice?

(ii). Based on your answer for (i)., please use Petrick’s method to find which implicants should be included in the multiple output implementation for \( g_1 \) and \( g_2 \). Does it give us the optimal implementation?
II. (Deciphering Characteristic Equations)

As you have by now done many times in your homework, you can easily go from a function in the form of a list of minterms or in a K-Map to a sum-of-products or product-of-sums. Likewise, you can create a list of minterms or a K-Map from the product-of-sums or sum-of-products. However, any don't cares in the original function will either be assigned a fixed value of 1 or 0 when coming up with the product-of-sums or sum-of-products. As a result, much of the information about what input combinations to the original function result in a don't care gets lost. However, if all of the terms are minimal implicants or minimal implicates, it is possible to figure out which outputs we are certain about, and which ones could be a don't care.

To illustrate how this can be done, let’s take the simple example sum-of-products $X + Y$ containing minimal implicants. If we assume that there are no don't cares in the original function, we would end up with a K-Map that looks like the following:

\[
\begin{array}{c|c|c}
X \backslash Y & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

As can be seen from the K-Map, the minterm for the input combination $(X = 1, Y = 1)$ is covered by both implicants. However, if the value for the input combination $(X = 1, Y = 1)$ was originally a don't care, in the minimal cover we would end up with the exact same set of implicants. That means that for the input combination $(X = 1, Y = 1)$, the original K-Map could have contained either a 1 or a don't care. Now, let’s see what happens if the input combination $(X = 1, Y = 0)$ is instead replaced with a don't care. With this minor alteration, the K-Map would become:

\[
\begin{array}{c|c|c}
X \backslash Y & 0 & 1 \\
0 & 0 & 1 \\
1 & dc & 1 \\
\end{array}
\]

Since we are going for minimal implicants, we would get the implicant $Y$ to cover the right-most column, and then we are done, since our newly introduced don't care does not need to be covered. The resulting sum-of-products is therefore just $Y$, which is different from the sum-of-products we were originally given; since the original sum-of-products was reportedly in minimal form, we know that an input combination of $(X = 1, Y = 0)$ can’t be a don't care, as that would have led to the $X + Y$ sum-of-products not to be the minimal implicant cover. By similar reasoning, an input of $(X = 0, Y = 1)$ can’t be a don't care either.

Now, what happens if the input $(X = 0, Y = 0)$ were a don't care? In this case, the resulting K-Map would look like:

\[
\begin{array}{c|c|c}
X \backslash Y & 0 & 1 \\
0 & dc & 1 \\
1 & 1 & 1 or dc \\
\end{array}
\]

The minimal implicant in this case would thus contain all of the entries in the function (regardless of whether $(X = 1, Y = 1)$ is a 1 or a don't care), so for all input combinations the output will be a 1, resulting in a minimal sum-of-products representation of 1 instead of $X + Y$. It should thus be evident that the original K-Map that gave rise to the minimal implementation of $X + Y$ could be either of the two K-Maps shown summarily below:

\[
\begin{array}{c|c|c}
X \backslash Y & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 or dc \\
\end{array}
\]
Now, after all this talk about the sum-of-products with minimal implicants you may be wondering what happens if the product-of-sums with the minimal implicates formulation of a function is given instead. Well, if we replace all of the 1’s with 0’s and vice versa in the previous examples, you will notice that the same lines of reasoning can be applied. For the minimal product-of-sums derivation of \((X')(Y')\), the possible designations (i.e. 0, 1, and/or \(dc\)) for each entry in the original function are shown in the K-Map below. As can be seen, the entry for \((X = 1, Y = 1)\) could be either a 0 or a \(don't care\).

\[
\begin{array}{c|c|c}
X \backslash Y & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 or dc \\
\end{array}
\]

(Part A) With your newfound understanding of how \(don't cares\) can disappear when writing a function in its product-of-sums or sum-of-products formulation, please indicate which of the K-Maps shown below could be the original function that results in the minimal sum-of-products form of \(Y' + Z\) by circling all of the ones that are possible.

(Part B) Now that you have figured out how to identify which observed outputs could potentially be \(don't cares\), let’s consider what additional information can be gleaned when a product-of-sums and a sum-of-products for the same function are given, both of which contain only their minimal implicants and implicates. Whereas with only the sum-of-products or product-of-sums form of a function given you are restricted to idle speculation on whether or not a particular output is a \(don't care\) or a fixed value, when both forms are provided you can tell with certainty that certain outputs are definitely a \(don't care\) when the two forms of the same original function have different outputs. For the minimal sum-of-products and minimal product-of-sums forms of the same function given below, please fill in each entry of the K-Map with all possible values that entry could have. (Remember that for some entries that appear to have a value of 1 or 0, a \(don't care\) could also be a possible value in the original function.)

\[
\begin{array}{c|c|c|c|c}
X \backslash Y Z & 00 & 01 & 11 & 10 \\
0 & dc & 1 & 1 & 0 \\
1 & dc & 1 & dc & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
X \backslash Y Z & 00 & 01 & 11 & 10 \\
0 & 1 & dc & 1 & dc \\
1 & dc & dc & 1 & 0 \\
\end{array}
\]

SOP: \(Y' + XZ\)
POS: \((X' + Z')(Y' + Z)\)

\[
\begin{array}{c|c|c|c}
X \backslash Y Z & 00 & 01 & 11 & 10 \\
0 & 0 & 1 & 1 \\
1 & 1 & dc & dc \\
\end{array}
\]
(Part C) After hearing of your expertise with reconstructing the original functions from sets of equations containing minimal implicants and implicates, two of your classmates come to you seeking assistance with their tale of woe. They had been designing a new form of flip-flop; however, after a late night studying for an exam they woke up only to discover that their cat was eating the notes detailing how the flip-flop state changes for different combinations of inputs. After salvaging the remainder of the notes from the cat, they discover to their despair that the only partially readable portion of their notes is the characteristic equation for their flip-flop design, in both minimal sum-of-products ($C_1$) and minimal product-of-sums ($C_2$) form.

\[ C_1 : Q_{next} = I'_0 + Q_{curr}I_1 \]
\[ C_2 : Q_{next} = (I_1)(Q_{curr} + I'_0) \]

To help identify for what input combinations the output of the characteristic equations will differ, your classmates ask you to draw out a separate K-Map for $C_1$ and $C_2$.

\[
\begin{array}{c|cc|cc|cc}
 Q_{curr} & \bar{I}_0 & I_1 & 00 & 01 & 11 & 10 \\
 \hline
 0 & & & & & & \\
 1 & & & & & & \\
\end{array}
\]  

\[
\begin{array}{c|cc|cc|cc}
 Q_{curr} & \bar{I}_0 & I_1 & 00 & 01 & 11 & 10 \\
 \hline
 0 & & & & & & \\
 1 & & & & & & \\
\end{array}
\]

(Part D) After seeing the discrepancies between the two forms of the original function, your classmates ask about which of the entries could have been don’t cares. To help them out, they ask you to fill in a K-Map that shows in each entry the possible values that entry could contain. (Remember that for some entries that appear to have a value of 1 or 0, a don’t care could also be a possible value in the original function.)

\[
\begin{array}{c|cc|cc|cc}
 Q_{curr} & \bar{I}_0 & I_1 & 00 & 01 & 11 & 10 \\
 \hline
 0 & & & & & & \\
 1 & & & & & & \\
\end{array}
\]

(Part E) After recovering from the trauma of studying for a midterm, your classmates recall that their flip-flop design only had 2 don’t care conditions. From this additional information you quickly realize which of the don’t cares you’ve identified are part of their flip-flop design, and write the updated K-Map with the correct values for your classmates.

\[
\begin{array}{c|cc|cc|cc}
 Q_{curr} & \bar{I}_0 & I_1 & 00 & 01 & 11 & 10 \\
 \hline
 0 & & & & & & \\
 1 & & & & & & \\
\end{array}
\]

(Part F) Unfortunately, your classmates are still unable to recall how their flip-flop was supposed to operate, and ask you to fill out the characteristic table for their flip-flop, and to give descriptive names to the two inputs, $I_0$ and $I_1$. (Note that entries for $Q_{next}$ are not limited to the values of 0 and 1. They may be written in terms of a variable such as $Q_{curr}$; certain entries may also be invalid if the behavior involves only don’t cares.)

\[
\begin{array}{c|c|c|c}
 I_0 & I_1 & Q_{next} \\
 \hline
 0 & 0 & \text{ } \\
 0 & 1 & \text{ } \\
 1 & 0 & \text{ } \\
 1 & 1 & \text{ } \\
\end{array}
\]
III. (Sequence Sequence Detectors)

Chapter 14 outlined a number of techniques that we studied for building a class of FSMs, namely, sequence detectors. The purpose of this question is to challenge you to think about the hierarchical construction of Mealy FSMs by using the basic sequence detector FSMs as building blocks.

We will work with two lower level sequences in this question, namely, $\alpha = 110$ and $\beta = 100$. Therefore, we can reduce a sequence such as 11010000100 to the recognition of $\alpha \beta \beta$. Similarly, 11001110 will map to $\alpha \beta \alpha$.

(Part A) (2 points) We have reproduced the bubble diagram for the sequence detector that recognizes $\alpha$ and $\beta$ below, but it is missing two crucial transitions. You will see that the missing transitions are exactly the ones that complete the final recognition step for each of the sequences. As you add them, please be mindful of sequence reuse.

(Part B) (4 points) Create a sequence detector that recognizes $\alpha$ and $\beta$, but once it sees the first $\beta$, the machine switches to a new mode in which it only recognizes $\alpha$.

We are asking you to hone your skills of FSM composition by hooking 2 copies of the basic building block FSM you constructed in (Part A). The only modifications to the basic building block you completed in (Part A) you are allowed to do are: a) Change the transition destinations of the 1-outputting transitions b) Change an output value from 1 to 0.
(Part C) (4 points) After seeing the wonderful feats of composition that you accomplished by combining sequence detectors together, you embark on a more difficult specification. Create a sequence detector that recognizes nothing until two $\beta$ sequences occur a row: once it sees two $\beta$ sequences it switches behaviors and recognizes both $\alpha$ and $\beta$ sequences. The two $\beta$ sequences must occur back-to-back with no intervening $\alpha$ sequences; however, junk bits that don’t match a pattern can occur without disturbing the machine’s state. You should be able to do it by using the 3 basic building blocks of (Part A). The same rules on modification as in (Part B) apply.

(Part D) (4 points) You have decided to implement one more FSM before calling it quits for the day. This sequence detector won’t recognize anything until two $\alpha$ sequences occur back-to-back or two $\beta$ sequences occur back-to-back; once either of those two sequences occurs the machine switches to a state where both the $\alpha$ and $\beta$ sequences are recognized. You should be able to do this using only 4 building blocks of (Part A). The same rules on modification as in (Part B) apply.
(Part E) (16 points) Now that we figured out how to hierarchically compose FSMs, let us try to carry this knowledge to the next lower level, that of FSM implementations. Since the underlying 4 building blocks of (Part D) are essentially the same, we can contemplate the possibility of reusing them as long as a higher level FSM keeps track of which lower level machine we are currently at (and thus can prod to do the small changes that you laboriously identified in the previous parts.) The higher level sequence detector that is keeping track of this state information will be called the *meta sequence detector*.

The lower level FSM is simply the 110 & 100 sequence detector that you filled out in (Part A). The meta sequence detector is most of what you will build during the course of this part. You may need to have the meta sequence detector chime in a bit to moderate the output behavior of the lower level FSM from (Part A), which outputs a 1 for both $\alpha$ and $\beta$, even though the specifications of (Part D) call for different output behavior depending on the state of the meta sequence detector.

(i) Now that you understand the meta sequence detector, please draw the bubble diagram for the meta sequence detector that detects the sequence from (Part D). Recall that the meta sequence detector listens for the $\alpha$ and $\beta$ events from the lower FSM and you can consequently use these two sequences as transition inputs in this high-level 4-state Mealy machine.

(ii) Randomly assign state encoding to the 4 states in your *meta sequence detector*.

(iii) It should be observed that we could run the regular sequence detector on the same bit inputs as before and it would act normally by detecting the $\alpha$ and $\beta$ sequences. Once it sees either $\alpha$ or $\beta$, the meta sequence detector picks up on that event and transitions to a new state (or possibly remains in the same state).
Therefore, at all times there are 3 possible events that the lower sequence detector could give to the meta sequence detector: an $\alpha$ sequence has been recognized, a $\beta$ sequence has been recognized, or neither has been recognized. These 3 events can be encoded using 2 bits of information that we will call $I_0$ and $I_1$. A bare boned block-level diagram of these two FSMs is given below.

The input of $I_1$ encodes if we have seen any valid sequences. In other words, $I_1 = 0$ means that neither $\alpha$ nor $\beta$ have been detected and $I_1 = 1$ means that either $\alpha$ or $\beta$ have been detected. In order to differentiate the sequence we’ve seen, we use input $I_0$. $I_0 = 0$ means that we have seen the $\beta$ sequence and $I_0 = 1$ means we have seen the $\alpha$ sequence. The outputs of $M_1$ and $M_0$ are the two bits that represent the current state of the meta sequence detector. We have provided the truth table for the encoding of $I_0$ and $I_1$ below.

<table>
<thead>
<tr>
<th>$I_1I_0$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>∅</td>
</tr>
<tr>
<td>01</td>
<td>∅</td>
</tr>
<tr>
<td>10</td>
<td>$\beta$</td>
</tr>
<tr>
<td>11</td>
<td>$\alpha$</td>
</tr>
</tbody>
</table>

As it turns out, these two input bits ($I_1$ and $I_0$) can be efficiently generated from the inputs and the output of the lower FSM. Please write compact Boolean expressions that generate these two bits of information. The state encodings for the lower level FSM of (Part A) are given in the truth table below.

<table>
<thead>
<tr>
<th>State</th>
<th>$Q_1Q_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>11</td>
</tr>
<tr>
<td>S3</td>
<td>10</td>
</tr>
</tbody>
</table>

$I_0 =$
$I_1 =$

(iv) Since the output function of the lower FSM no longer matches the specifications for the problem description (they need to be customized for each of the 4 metastates after all), we can no longer use it directly as the output to the sequence detector. As a result, please write the output logic of this new sequence detector by combining the output of the lower level FSM with some information from the higher level FSM so that it matches the problem specifications from (Part D).

$O =$