Fun Exercises for Logic with EXORs and Recursive Programming

Takashi Hirayama

Ver. 2019-04-09

Abstract

This manuscript is to do fun exercises for PPRMs and recursive programming. You will level up with your programming skill after finishing the exercises.

1 Recursive Algorithm

The following principle[1, 2] formulates the basic classes of AND-EXOR expressions.

Theorem 1 (Expansion Theorem). An arbitrary logic function $f(x_n, x_{n-1}, \ldots, x_1)$ can be expanded as

\[
\begin{align*}
  f(x_n, x_{n-1}, \ldots, x_1) &= f_0 \oplus x_n f_2 \\
  f(x_n, x_{n-1}, \ldots, x_1) &= \overline{x_n} f_2 \oplus f_1 \\
  f(x_n, x_{n-1}, \ldots, x_1) &= \overline{x_n} f_0 \oplus x_n f_1,
\end{align*}
\]

where $f_0 = f(0, x_{n-1}, \ldots, x_1)$, $f_0 = f(1, x_{n-1}, \ldots, x_1)$, and $f_2 = f_0 \oplus f_1$.

Equations (1)–(3) are the positive Davio expansion, the negative Davio expansion, and the Shannon expansion, respectively. By expanding a function $f$ using Equation (1) recursively, we have a logical expression with only un-complemented literals, which is called a PPRM. Another definition of PPRM is given below.

Definition 1. Let $x^0$ and $x^1$ denote 1 and $x$, respectively. The logical expression in the form:

\[
\bigoplus_{0 \leq i \leq 2^n - 1} a_i \cdot x_{i_n}^{i_{n-1}} \cdots x_1^{i_1}
\]

is a positive polarity Reed-Muller expression (PPRM), where $(i_n, i_{n-1}, \ldots, i_1)$ is the binary representation of integer $i$, and $a_i \in \{0, 1\}$ is a constant.

Equation (4) defines the general form of PPRMs, and Equation (1) explains how to obtain the PPRM of $f$. A PPRM is also called a Reed-Muller canonical expression. For a logic function, the PPRM is unique and is a canonical expression. From Definition 1, a PPRM is specified by the bit sequence $a_0, a_1, \ldots, a_{2^n-1}$. In this manuscript, we represent a PPRM as a bit sequence like Example 1.
1: function GetPPRM1(b, n): Integer
   ▷ Input: b is the truth table of a logic function, and n is the number of variables.
   ▷ Output: the binary PPRM of b.
2: var f0, f1, f2, p0, p2: Integer;
3: if n = 0 then
   ▷ Terminal condition.
4:   return b;
5: end if
6: f1 ← ShiftRight(b, 2^{n-1});
   ▷ f1 is the upper half of bit sequence of b
7: f0 ← b ⊕ ShiftLeft(f1, 2^{n-1});
   ▷ f0 is the lower half of bit sequence of b
8: f2 ← f0 ⊕ f1;
9: p2 ← GetPPRM1(f2, n - 1);
   ▷ p2 is the upper half of binary PPRM
10: p0 ← GetPPRM1(f0, n - 1);
   ▷ p0 is the lower half of binary PPRM
11: return ShiftLeft(p2, 2^{n-1}) ⊕ p0;
12: end function

Figure 1: GetPPRM1: an algorithm with the Davio expansion

Example 1. A logic function is often represented by the truth table like Table 1, which corresponds to the minterm expansion: \( \bar{x}_3 \bar{x}_2 \bar{x}_1 \oplus \bar{x}_3 \bar{x}_2 x_1 \oplus \bar{x}_3 x_2 \bar{x}_1 \oplus x_3 \bar{x}_2 \bar{x}_1 \oplus x_3 x_2 \bar{x}_1 \). Hereafter, we refer to the output sequence 10011011 as the data structure of truth table of \( f \). Note that, in this manuscript, the output for the input (0,0,0) is the least-significant bit and one for (1,1,1) is the most-significant bit. The same function \( f \) can also be represented by the PPRM: \( 1 \oplus x_3 \oplus x_2 x_1 \oplus x_3 x_1 \oplus x_3 x_2 x_1 \). Table 2 shows the corresponding \( a_i \)'s in Definition 1. We call the bit sequence 10101101 of Table 2 the binary PPRM of \( f \), in which \( a_{2^{n-1}} \) is the most-significant bit.

<table>
<thead>
<tr>
<th>Table 1: Truth table</th>
<th>Table 2: PPRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_3 )</td>
<td>( x_2 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

From Theorem 1, a conversion algorithm from a truth table to the PPRM can be built with recursive applications of Equation (1). GetPPRM1 in Figure 1 is the pseudo-code of the algorithm. In this pseudo-code, ‘⊕’ is the bit-wise EXOR operation of binary integers. The function ShiftRight(b, k) takes integers b and k, then returns the integer of shifting b to the right by k bits. ShiftLeft is the left shift version of ShiftRight.

Exercise 1. Analyze the time complexity of GetPPRM1. For simplicity, assume that ShiftLeft and ShiftRight can be executed in constant time. First give the recurrence relation for the running time \( T(n) \) of GetPPRM1(b, n). Then solve the recurrence. Finally, give the order of complexity using big O notation.
Exercise 2. Implement GetPPRM1 with your favorite programming language. To deal with big integers, a high-level programming language that supports big integers is recommended. Check your program with the test cases given in Table 3.

Table 3: Test cases for PPRMs

<table>
<thead>
<tr>
<th>Truth table</th>
<th>n</th>
<th>Binary PPRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10011011</td>
<td>3</td>
<td>0b10101101</td>
</tr>
<tr>
<td>0b00000001</td>
<td>3</td>
<td>0b11111111</td>
</tr>
<tr>
<td>0b10000001</td>
<td>3</td>
<td>0b01111111</td>
</tr>
<tr>
<td>0b10110110</td>
<td>3</td>
<td>0b10110110</td>
</tr>
<tr>
<td>0b11111111</td>
<td>3</td>
<td>0b00000001</td>
</tr>
<tr>
<td>0xD6A3</td>
<td>4</td>
<td>0xA375</td>
</tr>
<tr>
<td>0xEC48</td>
<td>4</td>
<td>0xEC48</td>
</tr>
<tr>
<td>0x8000</td>
<td>4</td>
<td>0x8000</td>
</tr>
<tr>
<td>0xCC006D5F</td>
<td>5</td>
<td>0xFE21FA21</td>
</tr>
<tr>
<td>0x5EA3E55C1F1816E4</td>
<td>6</td>
<td>0x2F551D582354BA2C</td>
</tr>
<tr>
<td>0xF5EBDD6D7DA16552933A12A2D878C4B3</td>
<td>7</td>
<td>0x5DC6C04E14D8C01FAE65D90F299D0985</td>
</tr>
</tbody>
</table>

2 Iterative Algorithm

Example 2. \( \bar{x} = x \oplus 1 \) is a basic property of the EXOR operation. By substituting negative literals with this equation, any minterm expansion can be converted to the PPRM. For example, \( f \) given in Example 1 is converted to the PPRM by substituting \( \bar{x_3} = x_3 \oplus 1 \), \( \bar{x_2} = x_2 \oplus 1 \), and \( \bar{x_1} = x_1 \oplus 1 \) as follows.

\[
\begin{align*}
f & = \bar{x_3}\bar{x_2}x_1 \oplus \bar{x_3}x_2x_1 \oplus \bar{x_3}x_2\bar{x_1} \oplus x_3x_2x_1 \\
& = (x_3 \oplus 1)(x_2 \oplus 1)(x_1 \oplus 1) \oplus (x_3 \oplus 1)(x_2 \oplus 1)x_1 \oplus (x_3 \oplus 1)x_2x_1 \oplus x_3x_2x_1 \\
& = 1 \oplus x_2 \oplus x_2x_1 \oplus x_3x_1 \oplus x_3x_2x_1
\end{align*}
\]

Based on the idea of Example 2, we can convert a truth table to the binary PPRM. This allows us to make another conversion algorithm without recursive calls. We name the algorithm GetPPRM2 in this manuscript.

Exercise 3. Implement GetPPRM2, and check your program with the test cases in Table 3. (Hint: you may introduce another data structure suitable for the substitution \( \bar{x} = x \oplus 1 \), and use it as a temporary and intermediate representation during the conversion from a truth table to the binary PPRM.)

Exercise 4. Make an automatic test program to check if GetPPRM1 equals to GetPPRM2 for randomly-generated truth tables. The program should test at least ten random truth tables for each \( n = 4, 5, \ldots, 10 \).

Exercise 5. Give the pseudo-code of your GetPPRM2 to express your algorithm clearly, like GetPPRM1 in Figure 1.
3 Conclusion

Through these exercises, you have noticed that an algorithm that looks simpler for human beings (like Example 2) does not always result in easier programming. Actually, GetPRM1 with recursion is far easier for programming than GetPPRM2 even though the conversion process of GetPPRM1 is hardly traced by humans. Recursion is a powerful technique to make programming easier. It is especially effective in the case where the given problems or data structures are defined in recursive forms. Keep it in mind and enjoy programming with recursive techniques!

References
