Multi-output Functional Decomposition with Exploitation of Don’t Cares

Christoph Scholl

Institute of Computer Science, Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
email: scholl@informatik.uni-freiburg.de

Abstract

Functional decomposition is an important technique in logic synthesis, especially for the design of lookup table based FPGA architectures.

We present a method for functional decomposition with a novel concept for the exploitation of don’t cares thereby combining two essential goals: the minimization of the numbers of decomposition functions in the current decomposition step and the extraction of common subfunctions for multi-output Boolean functions.

The exploitation of symmetries of Boolean functions plays an important role in our algorithm as a means to minimize the number of decomposition functions not only for the current decomposition step but also for the (recursive) decomposition algorithm as a whole.

Experimental results prove the effectiveness of our approach.

1 Introduction

Functional decomposition was introduced by Ashenhurst [1], Curtis [4], Roth and Karp [16, 7]. During last years functional decomposition attracted a lot of interest especially in connection with the design of lookup table based FPGA architectures (see e.g. [14, 8, 9, 10, 18, 24, 21]). Efficient functional decomposition methods based on Binary Decision Diagrams (BDDs) were proposed; there were improvements on the basic decomposition techniques with respect to the extraction of common sublogic in the decomposition of multi-output Boolean functions [11, 24, 21]. In [3, 2] a BDD based method was presented which computes extensions of incompletely specified single-output functions with a minimal number of decomposition functions in the current decomposition step.

The exploitation of don’t cares is an important step in functional decomposition even for completely specified functions, since decomposition is applied recursively and at least at higher levels of the recursion we usually obtain incompletely specified functions.

In our decomposition procedure we use an improved method to exploit don’t cares which does not only minimize communication complexity in the current decomposition step but has also an effect on the (recursive) decomposition of the decomposition functions in our procedure. We assign don’t cares to maximize the number of symmetries to achieve this ‘global’ effect (see also [20]).

In addition we developed a new method for don’t care assignment with respect to the computation of common decomposition functions of multi-output functions. This method is intended to increase the potential to share decomposition functions for several single-output functions.

Finally we apply the method of Chang and Marek-Sadowska [3, 2] to minimize the number of decomposition functions for single-output functions in the current decomposition step.

The crucial point in our don’t care assignment concept is the fact that all steps in this procedure are compatible in the sense that one step does not destroy the results of the previous one.

The paper is organized as follows: In Section 2 we define basic notations. In Section 3 we briefly review our method to compute common decomposition functions for multi-output functions [21]. The role of symmetries in logic synthesis is investigated in Section 4 and our concept for don’t care assignment is given in Section 5. In Section 6 experimental results are given and Section 7 concludes the paper.

2 Preliminaries

We restate some well-known definitions for decomposition of Boolean functions.

A single-output Boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) with input variables \( x_1, \ldots, x_n \) is decomposed with respect to a subset \( \{x_1, \ldots, x_p\} \) \((1 < p < n)\) of the input variables according to Figure 1. The functions \( a_i : \{0, 1\}^p \rightarrow \{0, 1\} \) \((1 \leq i \leq r)\) are called decomposition functions and the function \( g : \{0, 1\}^{n-p} \rightarrow \{0, 1\} \) is called composition function. \( \{x_1, \ldots, x_p\} \) is called bound set and \( \{x_{p+1}, \ldots, x_n\} \) is called free set. \( f \) is said to be decomposable with respect to \( \{x_1, \ldots, x_p\} \), if there is a decomposition with \( r < p \) decomposition functions.

If the Boolean function is to be realized by an FPGA with \( n_{LUT}\)-input lookup tables and if \( p \leq n_{LUT} \), \( \alpha = (\alpha_1, \ldots, \alpha_r) \) can be realized by \( r \) lookup tables (similarly for \( g \)). If the number of inputs of \( \alpha \) or \( g \) is still too large. decomposition has to be applied recursively to \( \alpha \) and \( g \).
In this section we will briefly review our method to decompose multi-output functions [21].

Given an m-output function \( f = (f_1, \ldots, f_m) : \{0, 1\}^n \rightarrow \{0, 1\}^m \) we have to compute m decompositions with respect to bound set \( \{x_1, \ldots, x_p\} \)

\[
f_i(x_1, \ldots, x_n) = g_i(\alpha_i(1)(x_1, \ldots, x_p), \ldots, \alpha_i(n)(x_1, \ldots, x_p), x_{p+1}, \ldots, x_n).
\]

Unlike [10] we choose decompositions with minimal numbers of decomposition functions \( r_i = \lfloor \log_2(ncc(f_i, \{x_1, \ldots, x_p\}) \rfloor \) for each \( f_i \) (1 \( \leq i \leq m \)), since our goal is to minimize the number of inputs of decomposition functions and composition functions, such that they can be realized by one LUT as soon as possible. (In [10] the total number of decomposition functions for \( f_1, \ldots, f_m \) is minimized, but the number of inputs of \( g_i \) can be (much) larger than \( n - p + \lfloor \log_2(ncc(f_i, \{x_1, \ldots, x_p\}) \rfloor \).)

Thus, under the condition \( r_i = \lfloor \log_2(ncc(f_i, \{x_1, \ldots, x_p\}) \rfloor \) we minimize the number of decomposition functions \( \bigcup_{i=1}^m \{\alpha_i(1), \ldots, \alpha_i(n)\} \).

This is done by a BDD based computation of common decomposition functions for subsets \( \{f_1, \ldots, f_i\} \) [21]. The computation is significantly sped up by a restriction of the search space to the so-called strict decomposition functions. A decomposition function \( \alpha_i^{(j)} \) is called strict, iff \( \forall e^{(1)}, e^{(2)} \in \{0, 1\}^p: e^{(1)} \equiv e^{(2)} \Rightarrow \alpha_i^{(j)}(e^{(1)}) = \alpha_i^{(j)}(e^{(2)}) \). \( \equiv \) is the compatibility relation for \( f_i \).

However, the restriction to strict decomposition functions has not only this ‘technical’ reason. It can be shown, that strict decomposition functions preserve structural properties of the functions \( f_i \), which is crucial for our decomposition algorithm (see Section 4).

4 Symmetries

Logic synthesis can take advantage of symmetries of Boolean functions. In the decomposition approach, e.g., symmetries in the set of bound variables lead to smaller numbers of decomposition functions. For the extreme case of \( f \) being symmetric in the bound set \( \{x_1, \ldots, x_p\} \) (i.e. \( f \) does not change, if any pair of variables from \( \{x_1, \ldots, x_p\} \) is exchanged), it is easy to see that the number of decomposition functions needed in a decomposition with respect to \( \{x_1, \ldots, x_p\} \) is not larger than \( \lfloor \log(p+1) \rfloor \). Analogous results hold when \( f \) is symmetric in not all pairs of variables from \( \{x_1, \ldots, x_p\} \).

Strict decomposition functions have the property that they preserve symmetry properties: If \( f \) is symmetric in a pair \( x_i, x_j \) of variables from \( \{x_1, \ldots, x_p\} \), then all strict decomposition functions of \( f \) are symmetric in \( x_i \) and \( x_j \). This fact is also true for more general types of symmetry like \( G \)-symmetries in the bound set \( \{x_1, \ldots, x_p\} \).

5 Incompletely Specified Functions

If in a decomposition \( f(x_1, \ldots, x_n) = g(\alpha(x_1, \ldots, x_p), \ldots, \alpha_r(x_1, \ldots, x_p), x_{p+1}, \ldots, x_n) \) some codes \( (a_1, r_1) \ldots, (a_r, r_r) \) do not occur in the image of \( \alpha \), then \( (a_1, r_1, \delta_1, \ldots, \delta_{n-p}) \) is a don’t care of \( g \) for all \( (\delta_1, \ldots, \delta_{n-p}) \in \{0, 1\}^n \). Because of that we

\*\( G \)-symmetries in the bound set \( \{x_1, \ldots, x_p\} \) consist of all possible combinations of exchanges and negations of variables from \( \{x_1, \ldots, x_p\} \). Various types of symmetries can be expressed as \( G \)-symmetries: Equivalence symmetry \( G \) in \( x_i \) and \( x_j \), e.g., means that \( f \) does not change under application of the following sequence: negation of \( x_i \), exchange of \( x_i \) and \( x_j \), negation of \( x_i \).
have to deal with incompletely specified functions during the recursive decomposition procedure, even if we start with completely specified functions.

Our concept to assign values to don’t cares consists of three steps:

1. First of all, we assign don’t cares in order to obtain as many symmetries as possible for the resulting function. As mentioned in the previous section this will lead to an reduction of the number of decomposition functions. There is not only an effect on the current decomposition step, but also on later (recursive) decompositions.

Don’t care assignment to obtain symmetries can be done before the selection of a bound set for the decomposition. We then use symmetric sifting [12, 15] to determine a starting point of our search for good candidates for bound sets. During the search for a good bound set we exchange groups of symmetric variables.

The difficulty in the don’t care assignment consists of the fact, that an assignment to obtain symmetry in a pair \((x_i, x_j)\) can destroy symmetry in another pair \((x_k, x_l)\). Theory and an algorithm to solve this don’t care assignment problem heuristically are given more detailed in [20]. We treat as well nonequivalence symmetry as equivalence symmetry [5].

2. In general, the functions to be decomposed have still don’t cares after step 1. These remaining don’t cares are assigned with respect to logic sharing. The don’t care assignment takes into account that for a multi-output function \(f = (f_1, \ldots, f_m)\) the functions \(f_1, \ldots, f_m\) are decomposed with computation of common decomposition functions. We minimize not only the number of decomposition functions for the single-output functions \(f_i\), but also the total number of decomposition functions to obtain as much logic sharing as possible.

To achieve this goal we propose to minimize a lower bound on the total number of decomposition functions of \(f_1, \ldots, f_m\).

If \(f_1, \ldots, f_m\) are completely specified, a lower bound on the total number of decomposition functions is computed as follows:

Now we call two bound set vertices \(e^{(i)}\) and \(e^{(2)}\in \{0,1\}^p\) compatible \((e^{(i)} \equiv e^{(2)})\), iff for all \(1 \leq i \leq m\) and \(\forall \delta \in \{0,1\}^{n-p}: f_i(e^{(1)}, \delta) = f_i(e^{(2)}, \delta)\). If \(ncc(f, \{x_1, \ldots, x_p\})\) is the number of different compatible classes according to this definition of compatibility, a lower bound on the total number of decomposition functions of \(f_1, \ldots, f_m\) is given by \(\lceil \log_2(ncc(f, \{x_1, \ldots, x_p\})) \rceil\). Let \((\alpha_1^{(i)}, \ldots, \alpha_{r_i}^{(i)})\) be the set of decomposition functions for \(f_i\) \((r_i = \min_{x^{(i)}}(\alpha_1^{(i)}, \ldots, \alpha_{r_i}^{(i)})\)) as defined in Section 2) and let \(r = \lceil \sum_{i=1}^{m} (\alpha_1^{(i)}, \ldots, \alpha_{r_i}^{(i)})\rceil\) be the total number of decomposition functions in the decomposition of \(f = (f_1, \ldots, f_m)\).

Then we have \(\lceil \log_2(ncc(f, \{x_1, \ldots, x_p\})) \rceil \leq r \leq \sum_{i=1}^{m} \lceil \log_2(ncc(f, \{x_1, \ldots, x_p\})) \rceil = \sum_{i=1}^{m} r_i\).

Thus, \(\lceil \log_2(ncc(f, \{x_1, \ldots, x_p\})) \rceil\) is not only a lower bound on \(\sum_{i=1}^{m} r_i\) but it also provides an estimation to what extent we can expect to find common decomposition functions in the decomposition of \(f_1, \ldots, f_m\). If \(\lceil \log_2(ncc(f, \{x_1, \ldots, x_p\})) \rceil\) is small whereas \(\sum_{i=1}^{m} r_i\) is large, then we can hope that there is a large potential to share decomposition functions in the decomposition of the single-output functions \(f_i\).

The minimization of this lower bound can be reduced to the computation of a graph (based on BDD representations of incompletely specified functions \(f_i\)) and a solution of the minimum clique cover problem. All further details are omitted due to page limitation.

3. Finally we exploit remaining don’t cares after step 2 to further minimize the number of decomposition functions for single-output functions \(f_i\) using the method of Chang and Marek-Sadowska [3].

We can prove that the don’t care assignment of step 3 can not increase the lower bound from step 2. Moreover we can prove that the procedure does not destroy symmetries, if each group of symmetric variables is completely contained in the bound set or in the free set [19].

6 Experimental Results

6.1 Arithmetic Functions

First of all we demonstrate that our automatic logic synthesis tool is able to produce competitive designs even for arithmetic functions which were already studied intensively using human intelligence.

Applied to adders of various operand lengths our tool automatically produces realizations which are very similar to the well-known conditional-sum adder [22]. Figure 2 shows the example of a two-input gate realization of an 8-bit adder generated by our tool. Differences in details even lead to a smaller number of gates for our realization (in the example of Figure 2 49 two-input gates compared to 90 two-input gates for the conditional-sum adder.)

We also applied our synthesis tool to partial multipliers, i.e. functions \(pm_n: \{0,1\}^{n^2} \rightarrow \{0,1\}^{2n}\) where the inputs are given by the bits of the \(n\) partial products\(^1\) and the outputs are given by the \(2n\) product bits.

Figure 3 shows the result for \(pm_4\). The result can be interpreted as a new multiplier scheme with a ‘column-wise’ addition of the bits of the multiplication matrix. Note that the don’t care assignment concept from Section 5 is essential for these results. A realization with-

\(^1\)i.e. conjuctions \(p_{i,j} = a_i b_j\) of bits of the operands \((a_1, \ldots, a_n)\) and \((b_1, \ldots, b_n)\).
out this don’t care assignment leads to a circuit with 75% more gates for pm, e.g.
A generalization of the principle to various operand lengths leads to a multiplier with \( 8 \cdot n^2 + O(n \log^2 n) \) two-input gates and depth \( 5.13 \cdot \log n + O(\log^2 n \log \log n) \) (compared to \( 10n^2 - 20n \) gates for the Wallace tree multiplier [23] with depth \( 5 \log n - 5 \)).

### 6.2 Benchmark Circuits

We applied the decomposition procedure described above to several MCNC and ISCAS benchmarks to compute FPGA realizations for Xilinx XC3000 device (where the number of inputs of the lookup tables is \( n_{LUT} = 5 \)).

We compared the numbers of CLBs for our new algorithm \texttt{mulop-dc} to the results of \texttt{mulopII} [21], where we didn’t use any don’t care assignment procedure. The results of Table 1 show a considerable reduction of CLB counts for our new algorithm. There are

\[
\log^* n := \min \{ m | \log^{(m)}(n) \leq 1 \} \text{ with } \log^{(0)}(n) := n \text{ and } \\
\log^{(i)}(n) := \log(\log^{(i-1)}(n)) \text{ for } i \in \mathbb{N}
\]

\footnote{\( \log^* n := \min \{ m | \log^{(m)}(n) \leq 1 \} \text{ with } \log^{(0)}(n) := n \text{ and } \\
\log^{(i)}(n) := \log(\log^{(i-1)}(n)) \text{ for } i \in \mathbb{N} \).}

 reductions of CLB counts of up to 35% for \texttt{alu2} and the overall reduction is more than 10%. Note that the benchmark functions are all completely specified functions and don’t cares occur only at higher levels of the recursion. For this reason it is clear that improvements can be obtained only for larger benchmarks.

Finally, Table 2 shows a comparison between our tool \texttt{mulop-dcII}, \texttt{FGMap} [8], mis-pgu(new) [14, 17] and \texttt{IMODEC} [24] proving the advantages of our procedure.

### 7 Conclusions

We presented a method for functional decomposition which combines the exploitation of don’t cares with the exploitation of symmetries of Boolean functions and the extraction of common subfunctions for multi-output Boolean functions.

Applied to FPGA synthesis, our methods to exploit don’t cares lead to considerable reductions of CLB counts of up to 35% for \texttt{alu2} and the overall reduction is more than 10%. Note that the benchmark functions are all completely specified functions and don’t cares occur only at higher levels of the recursion. For this reason it is clear that improvements can be obtained only for larger benchmarks.

Finally, Table 2 shows a comparison between our tool \texttt{mulop-dcII}, \texttt{FGMap} [8], mis-pgu(new) [14, 17] and \texttt{IMODEC} [24] proving the advantages of our procedure.
Figure 3: Automatically generated circuit for a partial 4-bit multiplier.
Table 1: Comparison of CLB counts for XC3000 device without and with don’t care exploitation.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>( i )</th>
<th>( o )</th>
<th>Number of CLBs</th>
<th>( \text{mulop-I} )</th>
<th>( \text{mulop-de} )</th>
<th>( \text{IMODEC} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5x1p</td>
<td>5</td>
<td>15</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>9ym</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>alu2</td>
<td>10</td>
<td>6</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>apex7</td>
<td>49</td>
<td>37</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>b9</td>
<td>41</td>
<td>21</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>C499</td>
<td>41</td>
<td>32</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>C880</td>
<td>60</td>
<td>26</td>
<td>87</td>
<td>87</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>clip</td>
<td>9</td>
<td>5</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>count</td>
<td>35</td>
<td>16</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>duke2</td>
<td>22</td>
<td>29</td>
<td>114</td>
<td>114</td>
<td>114</td>
<td>114</td>
</tr>
<tr>
<td>rd4</td>
<td>65</td>
<td>63</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>misex2</td>
<td>25</td>
<td>18</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>rd84</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>rot</td>
<td>135</td>
<td>107</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
</tr>
<tr>
<td>sao2</td>
<td>10</td>
<td>4</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>vg2</td>
<td>25</td>
<td>8</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>z4ml</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(total)</td>
<td></td>
<td></td>
<td>745</td>
<td>669</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison of CLB counts for XC3000 device between \( \text{mulop-deII} \), \( \text{FGMap} \), \( \text{mis-gga(new)} \), and \( \text{IMODEC} \).

CLB counts even for completely specified benchmark functions, since incompletely specified functions arise during the recursive application of the decomposition procedure.

References


