Delay Minimal Decomposition of Multiplexers in Technology Mapping

Shashidhar Thakur*
Synopsys Inc.,
700 E. Middlefield Road,
Mountainview, CA 94043

D.F. Wong*
Department of Computer Sciences,
University of Texas at Austin,
Austin, TX 78712

Shankar Krishnamoorthy
Synopsys Inc.,
700 E. Middlefield Road,
Mountainview, CA 94043

Abstract

Technology mapping requires the unmapped logic network to be represented in terms of base functions, usually two-input NORs and inverters. Technology decomposition is the step that transforms arbitrary networks to this form. Typically, such decomposition schemes ignore the fact that certain circuit elements can be mapped more efficiently by treating them separately during decomposition. Multiplexers are one such category of circuit elements. They appear very naturally in circuits, in the form of datapath elements and as a result of synthesis of CASE statements in HDL specifications of control logic. Mapping them using multiplexers in technology libraries has many advantages. In this paper, we give an algorithm for optimally decomposing multiplexers, so as to minimize the delay of the network, and demonstrate its effectiveness in improving the quality of mapped circuits.

1 Introduction

Technology mapping involves the tasks of decomposition, matching, and covering. The first of these steps, decomposition, involves expressing every node in the logic network in terms of base functions. This step will be the focus of this paper.

The goal of the decomposition step is twofold; to create a good initial structure for the mapping algorithm to work on and to ensure that the logic function of each node belongs to a set of base functions, every member of which exists in the library. Most available technology mappers employ the base function set consisting of two-input NOR and NOT functions.

The structural tree-based mapping algorithm [1, 3, 5] is a very popular algorithm, and forms the core of many academic and commercial technology mapping tools. This algorithm partitions the unmapped logic network into a collection of trees. Each component tree is then optimally mapped to the library using graph matching techniques. When using such a mapper, it is of an advantage to decompose the unmapped network in such a way that the resulting network has fewer component trees. Each library cell is represented as a pattern tree, in which every node is one of the base functions. Such mappers can utilize library cells that can be represented by tree patterns only.

Circuits usually contain a large number of multiplexers (MUXes). This is especially true for circuits that are automatically synthesized from high-level descriptions. MUXes exist in the data-paths of circuits, where they are used to route operands to operators. Also, the control logic is frequently specified as a CASE statement in HDL descriptions. MUXess arise as a result of a direct translation of CASE statements in HDLs into a logic-level description. Figure 1 illustrates the occurrence of MUXes in circuits.

Cell libraries too contain various choices of MUXes. Cell implementations make use of the fact that a pass gate implementation of a MUX is both, faster and smaller. In the case of MUX-based FPGAs like Actel, there is a natural presence of MUX in the virtual library. Thus, a method for mapping

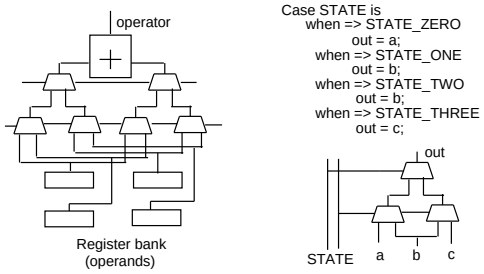

Figure 1: Occurrence of MUXes in synthesized circuits.

MUXes in the unmapped network to those in the library is desirable.

There are some problems that can result from decomposing such MUX nodes in terms of NOR and NOT base functions. These are illustrated in Figure 2. It shows the structure

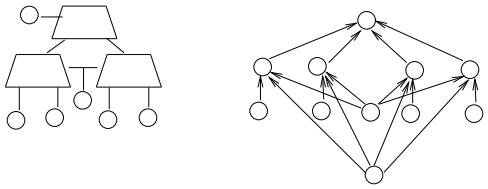

Figure 2: Result of decomposing a four-to-one MUX.

that results if a four-to-one MUX is represented in a two-level form, and then decomposed in terms of NOR and NOT gates. The immediate observation is that such a decomposition causes multiple fanouts in the network. As mentioned earlier, this is harmful when using a tree-based mapper. Another, and more important, point to note is that the mapper will not be able to match this circuit to three two-to-one MUXes in the library, the reason being that the two-to-one MUX is not a subgraph of the decomposed network. This illustrates the importance of finding a decomposition of a large MUX that preserves its structure, allowing the mapper to find matches with MUX cells in the library. An additional motivation for doing this is that the multiple fanouts, resulting for decomposing a MUX, get hidden inside a cell when such a match is chosen. This improves the routability of the circuit.

Rudell [5] suggested that an additional base function, in the form of a two-to-one MUX, be added to create an extended base function set. This allows structural mappers to make a better use of MUX library cells. Clearly, a scheme that identifies MUXes in the circuits, and decomposes them in terms of two-to-one MUX base functions, is needed to take advantage of this.

In this paper we show how large MUX nodes in the circuit can be decomposed into a tree of two-to-one MUX nodes, under the constant delay model (delays of gates are independent of drives and loads). The objective will be to minimize the depth of the tree, thus minimizing the propagation delay to the root of the decomposition tree. We give an algorithm that computes the optimal solution to this problem.

Some algorithms for technology mapping for MUX-based FPGAs [4] involve decomposing both, the network and the library

*Partially supported by the Texas Advanced Research Program under Grant No. 003658459, by a DAC Design Automation Scholarship, and by a grant from the AT&T Bell Laboratories.

DAC 96 - 06/96 Las Vegas, NV, USA ©1996 ACM, Inc. 0-89791-833-9/96/0006..$3.50

cells, in terms of two-to-one MUX base functions and then applying structural mapping algorithms. Here the aim is to get a representation using a small number of base function nodes, and the objective of meeting performance goals is left to the covering step. Other work on the problem of synthesizing Boolean networks using multiplexers involves trying to express each node in the circuit as a network of multiplexers [2, 6, 7]. The aim is to minimize the number of multiplexers used in the decomposition. In contrast, we do the decomposition step with the aim of getting good performance after mapping.

2 Definitions

Let $S = \{s_1, s_2, \ldots, s_\sigma\}$ and $D = \{d_1, d_2, \ldots, d_\delta\}$ be sets of Boolean variables and let $V = S \bigcup D$. Let f be a $\sigma + \delta$ input Boolean function, over the set of variables V. Below we will assume that i, j are variables that can take values from the set $\{1, 2, \ldots, \delta\}$. We assume that the input format for f is a Boolean expression.

For $i \leq \delta$, define $C_{d_i} = f_{d'_1 d'_2 \ldots d_i \ldots d'_\delta}$. Thus, C_{d_i} is the cofactor of f with respect to the positive phase of d_i, and the negative phases of the rest of the variables in D.

Definition 1: Function f is called a (δ, σ) *MUX* with *selector inputs* $s_1, s_2, \ldots, s_\sigma$ and *data inputs* $d_1, d_2, \ldots, d_\delta$ if the following conditions hold:

1. $s_1, s_2, \ldots, s_\sigma$ are binate in f and $d_1, d_2, \ldots, d_\delta$ are unate in f. We assume, without loss of generality, that the unate inputs of f are positive unate.
2. $(\bigvee_{i \leq \delta} C_{d_i} d_i) = f$.
3. For $i, j \leq \delta$, $C_{d_i} \bigwedge C_{d_j} = 0$.

□

For a variable d_i, C_{d_i} is called its *selector code*. Thus, the selector code of a data input is the disjunction of minterms of selector inputs that select it.

A *full MUX* is one in which each data signal gets selected by one and only one selector minterm, and each selector minterm selects some data signal. Thus, a full MUX has $\delta = 2^\sigma$. A *priority MUX* is one in which each data signal can get selected by more than one selector minterm. If each selector minterm selects some data signal the MUX is a *priority complete MUX*, else it is a *priority incomplete MUX*. Priority MUXes have $\delta < 2^\sigma$.

Definition 2: For a (δ, σ) MUX f, the function $(\bigvee_i C_{d_i})'$ is called the *don't care selector codes function*, and is denoted $(DC)_f$. □

Note that $(DC)_f$ is a function over variables in S only. Informally, this function is the sum of minterms of variables in S that do not select any data input in f. This is a non-zero function only for priority incomplete MUXes.

Definition 3: A *multiplexer decomposition* of a (δ, σ) MUX f, defined over the variables in V, is a tree of (2,1) MUXes that is functionally equivalent to f. The selector inputs of each (2,1) MUX is taken from S, and the data inputs of the MUXes are either the outputs of other (2,1) MUXes in the tree or elements of D. □

We denote the propagation delays of a (2,1) MUX and a two-input NOR by Δ_m and Δ_n, respectively. The delay of a NOT gate is assumed to be zero. We will use the *unit delay model* in all delay computations. The problem we address in this paper is stated below.

Problem 1 (Multiplexer Decomposition Problem): Given Δ_m, a (δ, σ) MUX f, and arrival times for all inputs of f, find a decomposition of f in terms of (2,1) MUXes such that the arrival time of the signal at the output of the root (2,1) MUX is minimized. □

3 Decomposing the Network

The algorithm for decomposing the network essentially does a topological walk over the DAG representing the input logic network. Clauses 1-3 in Definition 1 are checked to determine if the node represents a MUX. The nodes that are identified as MUXes are decomposed into a tree of (2,1) MUXes using our decomposition algorithm. Other nodes are decomposed in terms of NOR and NOT base functions using the Huffman tree-based Algorithm [8]. We call the corresponding algorithms *mux_decomp* and *huff_decomp*, respectively. The algorithm *mux_decomp* will be the subject of the next section.

4 Multiplexer Decomposition

We are now ready to present the multiplexer decomposition algorithm. We will analyze the complexity of the problem in Section 4.1. We will develop the algorithm in Sections 4.2, 4.3, and 4.4.

4.1 Complexity of the Problem

We claim that any algorithm for the MUX Decomposition Problem will have a worst case running time that is exponential in the size of the input. Recall that we assume that the input MUX, f, is specified as a Boolean expression. The above claim is stated in the following lemma:

Lemma 1: *The Multiplexer Decomposition Problem has time and space complexities that are exponential in the size of the input MUX, the number of selector inputs to the MUX, and the number of data inputs to the MUX.*

This implies that any algorithm for the problem has to have a running time of $O(2^n)$, for input f_n. Hence, any algorithm has to have a worst case running time that is exponential in the size of the input. As a consequence of this, any algorithm has to have a worst case running time that is exponential in the number of selector inputs and the number of data inputs. The exponential time complexity is not a disaster for problems of practical importance. Our experiments show that, for most MCNC circuits, the average number of selectors in the MUXes identified is less than 4. Thus, algorithms that have a worst case running time of $O(k^\sigma)$, where σ is the number of selectors and k is a small constant, will be of practical importance (as well as the best possible, theoretically).

4.2 Intuitive Algorithm

In this section, we present an intuitive algorithm that correctly decomposes a MUX node f. This algorithm finds the delay optimal decomposition of a full MUX. But for priority MUXes it is sub-optimal.

We first state a lemma that makes a recursive solution of the problem possible. The proof of this claim follows immediately from the definition of a MUX.

Lemma 2: *If f is a MUX and s is a selector input of f then f_s and $f_{s'}$ are MUXes too.*

The algorithm is easy to describe. The selectors are sorted according to their arrival times. The latest arriving selector s is chosen, and the cofactors f_s and $f_{s'}$ are computed. By Lemma 2, both these cofactors are MUXes. The algorithm then is to decompose each of the cofactors recursively. Doing a Shannon expansion of f, with respect to s, we get $f = sf_s + s'f_{s'}$. Hence, the outputs of the decomposition trees of f_s and $f_{s'}$ are used as data inputs to a new (2,1) MUX node, with s as selector input. Note that the MUX needs to be constructed only if $f_s \neq f_{s'}$, as otherwise s is not in the true support of f.

The algorithm described above is called *mux_decomp_selsort* If the input node is a full MUX, then the decomposition done by this algorithm minimizes the arrival time at the output of the root of the decomposition tree. This is because all the selectors have to occur on every path from the root to the leaves. Thus, arranging the selectors in the decreasing order of arrival times from the root to the leaves ensures a delay optimal decomposition, independent of the arrival times of the data signals. This algorithm is suboptimal for priority MUXes. This result is stated in the following lemma. The complexity of Algorithm *mux_decomp_selsort* is also stated.

Lemma 3: *If the input node is a MUX then Algorithm* mux_decomp_selsort *creates a decomposition tree of (2,1) MUX nodes, that is functionally equivalent to the input node. If the input MUX node is a full MUX then it creates a decomposition tree of (2,1) MUX nodes that minimizes the signal arrival time at the output of the root of the tree. The running time for this algorithm is $O(2^\sigma)$ and it uses $O(2^\sigma)$ space, where σ is the number of selector signals of the input node.*

4.3 Optimal Algorithm

We now present the optimal MUX decomposition algorithm. We first restrict ourselves to full MUXes and priority complete MUXes, i.e., the don't care selector codes function is zero. The decomposition of priority incomplete MUXes will be done by a slight extension to be described later.

For priority MUXes Algorithm *mux_decomp_selsort* may perform sub-optimally as is illustrated by the following example.

Example 1: Let $\Delta_m = 1$. Consider the following function:

$$f = s(tua + tu'b + t'uc + t'u'd) + s'(ue + tu'f + t'u'g)$$

This is a (7,3) priority complete MUX with selector inputs s, t, u and data inputs a, b, c, d, e, f, g. Assume the arrival times of s, t, u are 1,2 and 3, respectively. The arrival time of all data signals except e is 1, and that of e is 4. Figure 3(a) shows the result of applying Algorithm *mux_decomp_selsort* and Figure 3(b) shows a better decomposition.

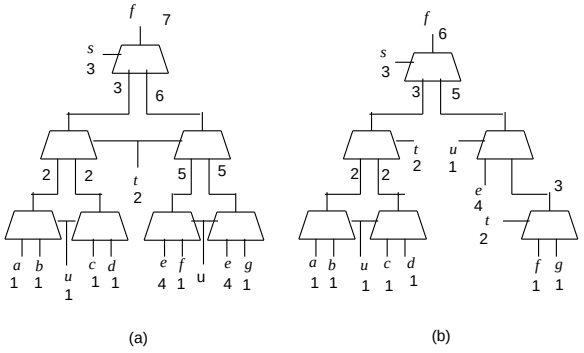

Figure 3: Illustration of sub-optimality of Algorithm *mux_decomp_selsort*.

The above example brings out two important points that lead to an optimal algorithm. It shows that the order of selectors on the two branches, starting from an internal MUX in the decomposition, need not be the same. In the decomposition shown in Figure 3(b), the left branches of the root MUX have the selector order t, u and the right branch has the selector order u, t. The order of selectors on the right branch is not the same as the order according to decreasing arrival times. This allows the late arriving data signal, e, to be pushed towards the root. Also, the example illustrates the fact that saving a MUX along a critical path (in this case the path from e to the root) can offset the disadvantage of pushing a late arriving selector signal away from the root of the decomposition tree.

The straightforward way then, to allow different selector orders along different branches starting at an internal MUX, is to modify Algorithm *mux_decomp_selsort* so that every selector is tried as the selector input to the root MUX. The chosen solution is the best among all the decompositions corresponding to each possible assignment of the selector input to the root MUX. The possibilities are tried out in a brute force manner, a source for inefficiency that will be corrected later in this section.

The following lemma states the optimality and the complexity of the above algorithm, which we call *mux_decomp_opt_ineff*.

Lemma 4: *If the input MUX node is a full or priority complete MUX then Algorithm* mux_decomp_opt_ineff *creates a decomposition tree of (2,1) MUX nodes that minimizes the signal arrival time at the output of the root of the tree. The running time of this algorithm is $O(2^\sigma \sigma!)$ and it uses $O(2^\sigma)$ space, where σ is the number of selectors signals of the the input node.*

Thus, this algorithm causes a blowup in the running time by a factor of $\sigma!$, where σ is the number of selectors, when compared to Algorithm *mux_decomp_selsort*. The reason for this inefficiency lies in the top-down approach to constructing the decomposition tree. This approach results in repeating the computation of the optimal decomposition tree for the same function. For example, if s and t are two selector signals of a MUX f, then the optimal decomposition for f_{st} is computed twice, once when s is chosen at the top recursion level and t at the second level, and once for the reversed order of selectors.

The final efficient algorithm uses a bottom-up, non-recursive, dynamic programming approach in building the tree. Partial solutions are stored, and re-used when the optimal decomposition of a previously processed function is desired. The intuition behind this is illustrated in Figure 4. Figures 4(a) and (b) show the exploration of two possible decompositions of a function f. The triangles represent MUX decomposition trees for the corresponding functions. We observe that the optimal decomposition of $f_{s'u}(= f_{us'})$ is required to evaluate the delay at the root node in both the possibilities. It is exactly this repeated computation that is avoided if the optimal decomposition of $f_{s'u}$ is stored the first time it is computed. We call the modified algorithm *mux_decomp_opt*.

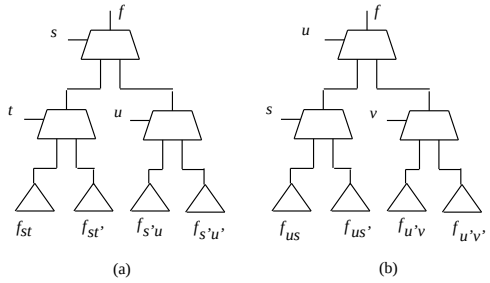

Figure 4: Illustration of efficient optimal MUX decomposition algorithm.

The following lemma states the optimality and the complexity of the above algorithm, which we call *mux_decomp_opt*.

Lemma 5: *If the input MUX node is a full or priority complete MUX then Algorithm* mux_decomp_opt *creates a decomposition tree of (2,1) MUX nodes that minimizes the signal arrival time at the output of the root of the tree. The running time of this algorithm is $O(\sigma 3^\sigma)$ and it uses $O(3^\sigma)$ space, where σ is the number of selectors signals of the the input node.*

4.4 Priority Incomplete Multiplexers

We now describe how the MUX decomposition algorithms can be extended to handle priority incomplete MUXes. By definition, for a priority incomplete MUX f, $(DC)_f \neq 0$. Introduce a new variable d_0, such that $d_0 \notin D$. Let $D' = D \bigcup \{d_0\}$, and $arr(d_0) = 0$. Consider the function g_f defined as follows:

$$g_f = f + d_0(DC)_f$$

We observe that g_f is a $(\delta + 1, \sigma)$ priority complete MUX, with selector signals being the members of S, and data signals being the members of D'. Either of Algorithms *mux_decomp_opt_ineff* or *mux_decomp_opt* can be used to decompose g_f optimally. A post-processing step is applied to extract the decomposition for f from this. In this step, all occurrences of d_0 in the decomposition are replaced by the constant 0. Then constant propagation is done, where every MUX with one data input 0 is replaced by a combination of a two-input NOR and a NOT node. Alternatively, the constants could be left in the network, so that the MUX structure is preserved.

Circuit Name	Area			Delay			Number of Nets			Pins/Net		
	huff	selsort	opt	huff	selsort	opt	huff	selsort	opt	huff	selsort	opt
C2670	1525	1519	1519	29.6	28.2	28.2	1142	1111	1111	2.92	2.87	2.87
C5315	3802	3671	3672	25.2	23.0	23.0	2540	2366	2370	3.26	3.24	3.23
apex7	451	456	456	9.3	9.3	9.3	330	329	329	3.08	3.05	3.05
cm150a	116	59	59	5.4	2.9	2.9	101	50	50	2.54	2.32	2.32
cm151a	61	50	50	4.3	3.9	3.9	52	41	41	2.60	2.46	2.46
cm152a	36	29	29	2.9	2.0	2.0	31	26	26	2.65	2.12	2.12
dalu	3463	3342	3342	41.6	35.3	35.3	2043	1906	1906	3.55	3.48	3.48
des	7714	7571	7571	69.7	58.7	58.7	4502	4299	4299	3.65	3.70	3.70
eric	8313	12025	7694	48.4	149.5	41.1	4521	4784	3893	3.77	4.03	3.82
i10	6756	6779	6777	75.7	75.6	75.4	3899	3904	3904	3.53	3.52	3.52
i8	3610	3603	3592	43.9	43.9	43.9	2182	2154	2150	3.50	3.51	3.51
mux	157	73	73	6.7	3.7	3.7	120	54	54	3.03	2.72	2.72
sdmux	213	87	87	5.1	2.6	2.6	124	62	62	3.24	2.34	2.34
tcon	40	32	32	1.0	1.1	1.1	49	25	25	2.47	1.96	1.96
normalized average	1.0	0.87	0.83	1.0	0.99	0.84	1.0	0.82	0.80	1.0	0.94	0.93

Table 1: Comparison of quality of mapped circuits - lsi_10K.

5 Experimental Results

We implemented the network decomposition algorithm based on the decomposition algorithms developed in this paper. We implemented it so as to offer three choices for the decomposition method.

1. **huff**: All nodes decomposed by Algorithm *huff_decomp*.
2. **selsort**: MUX nodes decomposed by Algorithm *mux_decomp_selsort*.
3. **opt**: MUX nodes decomposed by Algorithm *mux_decomp_opt*.

This was integrated with SIS as the command `decomp_network`.

We used two libraries, Actel and lsi_10K, which were selected because of the presence of large multiplexers in them (the largest MUX in the Actel2 library is a (4,2) full MUX and the largest in lsi_10K is a (8,3) full MUX). We just show the results for the lsi_10K library for brevity. The following sequence of operations were performed on each circuit.

```
sweep; eliminate 5;
```

The first operation removes single input and constant nodes from the circuit. The second operation, `eliminate`, was just a straightforward way to collapse small nodes to form larger ones. This was one way to recover MUX structures in the circuit that might have been destroyed in the process of translating from HDLs.

Following this, we did the decomposition and then mapped the circuits using the commands,

```
decomp_network -choice; map;
```

We did our experiments on a Sun Sparcstation 5. It was observed that the deviation in the CPU time for the three algorithms is only about 25%. Thus, the run-time penalty paid for doing multiplexer decomposition is very low.

Finally, we did the technology mapping with the aim of minimizing the delay of the mapped circuit as a primary aim and minimizing the area as the secondary aim. The technology mapper was a tree-based structural mapper that allowed a (2,1) MUX as a base function, in addition to a two-input NOR and NOT. We used the libraries for Actel2 and lsi_10K. The results of these experiments are tabulated in Table 1.

We make following conclusions from this experiment:

1. The intuitive selector sorting based MUX decomposition method of Algorithm *mux_decomp_selsort* does not cause any improvement in delay, on an average. This can be directly attributed to its sub-optimality for priority MUXes. This justifies the use of the more complicated Algorithm *mux_decomp_opt*.
2. Both the multiplexer decomposition algorithms cause substantial reductions in area. This can be attributed to the the fact that reducing multiple fanouts in the decomposed network improves the performance of tree-based technology mappers. Also, a better use is made of the efficient multiplexer implementations in the libraries.
3. For the larger circuits, despite the presence of many MUXes, the improvement in delay is small. One possible explanation for this is that the MUXes are along non-critical paths. Hence, a better decomposition for them does not have much of an influence on the delay of the mapped circuit. In contrast, an area improvement is observed for most of the examples.
4. As mentioned in the introduction, the decomposition of MUXes in terms of the base MUX function has the added advantage of reducing the number of multiple fanouts. This is confirmed by the reduction in number of nets and the number of pins per net.

A point to be noted is that the delay optimality of Algorithm *mux_decomp_opt* is with respect to the decision to decompose MUX nodes identified in terms of (2,1) MUXes. The possibility exists that a better mapping might be obtained by decomposing such nodes in terms of AND/OR gates. For example, in Table 1, the circuit tcon has a better delay after mapping when the Huffman tree-based algorithm is used.

References

[1] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proceedings of the International Conference on Computer Aided Design*, pages 116–119. IEEE/ACM, 1987.

[2] R. K. Gorai and A. Pal. Automated synthesis of combinational circuits by tree networks of multiplexers. In *Proc. 3rd Intl. Conf. VLSI Design*, pages 300–305. IEEE, January 1990.

[3] K. Keutzer. Dagon: Technology binding and local optimization by DAG matching. In *Proceedings of the Design Automation Conference*, pages 617–623. ACM/IEEE, 1987.

[4] R. Murgai, R.K. Brayton, and A. L. Sangiovanni-Vincentelli. An improved synthesis algorithm for multiplexer-based PGAs. In *Proceedings of the Design Automation Conference*, pages 380–386. IEEE/ACM, 1992.

[5] R. L. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U.C. Berkeley, April 1989.

[6] I. Schafer and M. Perkowski. Synthesis of multilevel multiplexer circuits for incompletely specified multioutput Boolean functions with mapping to multiplexer based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(11):1655–1664, November 1993.

[7] A. J. Tossner and D. Aoulad-Syad. Cascade networks of logic functions built in multiplexer units. *Proceedings of IEE, Pt. E*, 127(2):64–67, March 1980.

[8] A. R. R. Wang. *Algorithms for Multi-level Logic Optimization*. PhD thesis, U.C. Berkeley, April 1989.