Center for Reliable Computing

TECHNICAL REPORT

# Efficient Multiplexer Synthesis

Subhasish Mitra, LaNae J. Avra and Edward J. McCluskey

**00-3**

(CSL TR # ??)

March 2000

**Center for Reliable Computing**
Gates Building 2A, Room 236
Computer Systems Laboratory
Dept. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-9020

**Abstract:**

The multiplexer is a common standard sub-circuit used frequently in the datapath logic of complex designs, typically to provide a path for routing operands to operations and operation results to destination registers. During RTL synthesis, multiplexers are used for realizing if-then-else and case statements in the RTL design description. In this paper, we describe a new heuristic algorithm for synthesizing efficient multiplexers consisting of a tree of multiplexer components from a library. Area minimization is our primary goal. Hence, we first generate an area-minimal implementation of a multiplexer, using multiplexer components from the library. Subsequently, we minimize the delay of the area-minimal implementation. We have implemented the algorithm in a high-level synthesis tool. Experimental results show that our algorithm almost always generates the minimum-area multiplexer, and consistently generates smaller multiplexers than commercial tools that synthesize multiplexers. Moreover, experimental results show that the multiplexers generated by our technique are efficient in terms of propagation delay when compared to commercial tools.

**Funding:**

This work was supported by the Advanced Research Projects Agency under prime contract No. DABT63-94-C-0045 and DABT-97-C-0024.

**Imprimatur:** Samy Makar and Jonathan T. Y. Chang

# Efficient Multiplexer Synthesis

Subhasish Mitra, LaNae J. Avra and Edward J. McCluskey

CRC Technical Report No. 00-3
(CSL TR No. ??)
March 2000

**Center for Reliable Computing**
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

**Abstract**

The multiplexer is a common standard sub-circuit used frequently in the datapath logic of complex designs, typically to provide a path for routing operands to operations and operation results to destination registers. During RTL synthesis, multiplexers are used for realizing if-then-else and case statements in the RTL design description. In this paper, we describe a new heuristic algorithm for synthesizing efficient multiplexers consisting of a tree of multiplexer components from a library. Area minimization is our primary goal. Hence, we first generate an area-minimal implementation of a multiplexer, using multiplexer components from the library. Subsequently, we minimize the delay of the area-minimal implementation. We have implemented the algorithm in a high-level synthesis tool. Experimental results show that our algorithm almost always generates the minimum-area multiplexer, and consistently generates smaller multiplexers than commercial tools that synthesize multiplexers. Moreover, experimental results show that the multiplexers generated by our technique are efficient in terms of propagation delay when compared to commercial tools.

## TABLE OF CONTENTS

## 1. INTRODUCTION

A multiplexer is a standard sub-circuit which is often used in datapath logic to provide multiple connections between computation units. Typically, multiplexers are required for routing the operands to the operators in the datapath. In CAD tools for high-level synthesis, multiplexers are used to enable register sharing among several variables having non-overlapping lifetimes. During RTL synthesis, multiplexers are generated corresponding to the *if-then-else* and the *case* statements in Verilog or VHDL RTL design descriptions. Moreover, multiplexers are present in the libraries of multiplexer-based FPGAs, such as those produced by Actel. As a result, different aspects of multiplexers have been studied extensively in [Makar 88][Murgai 92][Thakur 96]. However, existing logic synthesis tools typically do not handle multiplexers efficiently during technology mapping. Technology mapping consists of three parts, decomposition, matching and covering. One technology mapping algorithm, the structural tree based algorithm [Detjens 87][Keutzer 87], partitions the unmapped logic network into a collection of trees. Each component tree is then optimally mapped to the elements in the technology library through graph matching techniques, where each library cell is represented as a set of pattern trees. However, the technology mapping algorithms based on graph matching typically cannot effectively utilize a rich technology library. This is because complex library cells, such as a 4-to-1 multiplexer, have many distinct tree representations, and generation, storage, and search of all possible tree representations is often difficult.

In this paper, we describe new techniques for synthesizing multiplexers, which are efficient in terms of area and delay, using multiplexer components available in a technology library. Our primary objective is area minimization. We do not detect multiplexers in random logic; a technique for this is described in [Thakur 96]. After performing high-level synthesis, we consider each multiplexer operation required and determine its implementation using component multiplexers available in the technology library so that the area of the synthesized multiplexer is minimized. In the past, synthesis of decoders as a tree of component decoders with a goal of minimizing the number of nets in the final switching circuit has been described in [Burks 54]. However, the multiplexer synthesis problem is more complex because after obtaining a minimum cost decomposition, we have to assign signals to the address inputs of the component multiplexers — this problem is non-trivial. Once the component multiplexers are determined, the next step in our algorithm is to assign a minimum number of signals to the address inputs of the component multiplexers. In this paper, we also describe our algorithm for address signal assignment and propose several schemes for generating efficient multiplexers by combining the area minimization and address signal assignment algorithm into one step. For a wide range of multiplexers, we have obtained results that validate the effectiveness of the cost functions we have chosen for our search-

based algorithm for generating area-efficient multiplexers presented in this paper. Once the area-efficient implementation of a multiplexer is obtained, we minimize the delay of that implementation using some techniques described in this paper. It may be noted that the general multiplexer synthesis problem has exponential complexity because the problem of delay minimal decomposition of a multiplexer into 2-to-1 multiplexers is exponential [Thakur 96].

Section 2 introduces the nomenclature used in this paper. Section 3 describes our area minimization algorithm. In Sec. 4, we present an algorithm for assigning signals to the address inputs of the component multiplexers. In Sec. 5, we compare the areas of the multiplexers generated by our scheme with those generated by an internal tool of an ASIC vendor and commercial tools from Synopsys [Synopsys 98] and Ambit [Ambit 97]. Section 6 describes techniques for minimizing the delay of the area-minimal implementation and provides results to compare the delays of the multiplexers generated by our scheme and a commercial tool. Finally, we conclude in Sec. 7.

## 2. NOMENCLATURE

In this section, we describe and define the terms related to multiplexers. "A **multiplexer** (mux) is a circuit that can select information from one of the several input terminals and route that input to a single output bit" [McCluskey 86].

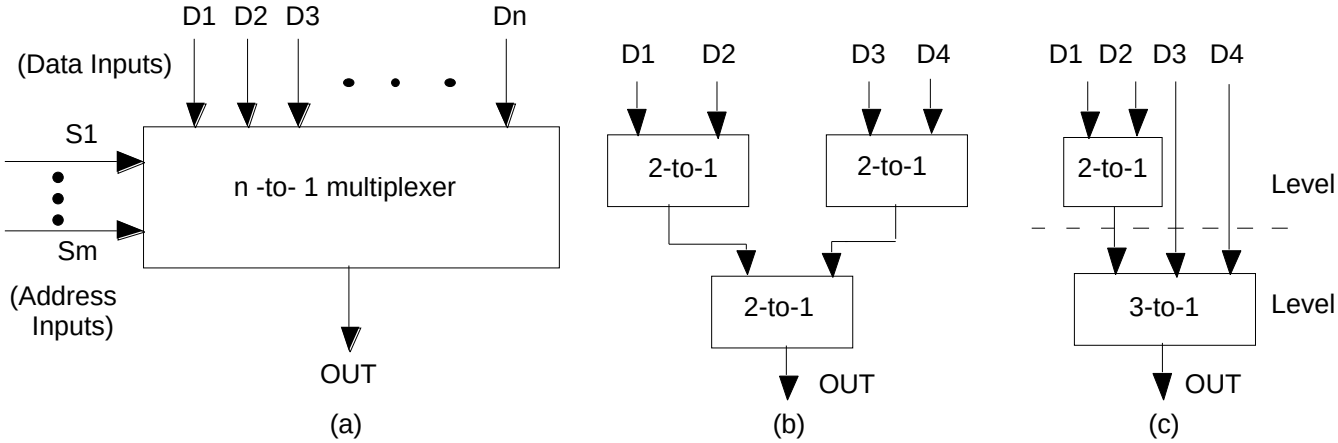



Figure 1. Multiplexers; (a) Block Diagram of a Multiplexer; (b) and (c) Implementation of a 4-to-1 mux from smaller multiplexers

The multiplexer has two sets of inputs as shown in Fig. 1(a): the *data* inputs (D1, D2 ,..., Dn) and the *address* inputs (S1, S2, ... , Sm), where $m = \lceil \log_2 n \rceil$. The binary code on the address inputs determines which data input is routed to the output. A *full (complete) multipexer* is one for which $n = 2^m$. Each data input of a full multiplexer is selected by one and only one binary code on the address inputs. An *incomplete multiplexer* is one for which $2^{m-1} < n < 2^m$. An *n-to-1*

multiplexer can be implemented using a tree of smaller multiplexers. Figures 1(b) and 1(c) show two possible implementations of a 4-to-1 multiplexer using component multiplexers. A multiplexer which has all its inputs pins connected to primary inputs is said to be at *level* 1. The *level* of any other multiplexer is defined as one plus the maximum level of a multiplexer whose output is connected to the input of the current multiplexer.

## 3. MULTIPLEXER AREA MINIMIZATION ALGORITHM

Given a library of multiplexers, the problem we consider here is to synthesize a larger multiplexer function using a tree of multiplexer components from the library such that the total area of the resulting multiplexer is the minimum. Techniques for minimizing the delay of the minimum-area implementation are described in Sec. 6. The information corresponding to each *n-to-1* multiplexer in the library that is required by our algorithm can be represented by a pair *<n, area>* where *n* is the number of data inputs. Two example libraries are shown in Table 1.

Table 1. Component multiplexers in Library 1 and Library 2

| n | 2 | 3 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| Library 1 Area | 8 | 14 | 19 | 33 | 42 |
| Library 2 Area | 8 | 14 | 21 | 33 | 48 |

The multiplexer area minimization problem is computationally intensive. The general problem can be proved to be NP-complete. An exhaustive technique enumerates all possible realizations of a larger multiplexer using smaller multiplexers from the library, computes the area of each realization and selects the one with the smallest area.

Table 2. Exhaustive Search for 5-to-1 multiplexer

| Level 1 | Level 2 | Level 3 | Level 4 |
|---|---|---|---|
| 2-to-1 | 2-to-1 | 2-to-1 | 2-to-1 |
| | | 3-to-1 | — |
| | 2-to-1, 2-to-1 | 2-to-1 | — |
| | 3-to-1 | 2-to-1 | — |
| | 4-to-1 | — | — |
| 2-to-1, 2-to-1 | 3-to-1 | — | — |
| | 2-to-1 | 2-to-1 | — |
| 3-to-1 | 3-to-1 | — | — |
| | 2-to-1 | 2-to-1 | — |
| 2-to-1, 3-to-1 | 2-to-1 | — | — |
| 4-to-1 | 2-to-1 | — | — |

For example, in order to implement a 5-to-1 multiplexer using Library 1, an exhaustive technique will consider individually each of the following cases in the first step (first level of multiplexers). The possibilities are: (a) one 2-to-1 mux (b) two 2-to-1 muxes, (c) one 3-to-1 mux, (d) one 2-to-1 and one 3-to-1 mux and (e) one 4-to-1 mux. For case (a), we need to implement a 4-to-1 in the next level. For cases (b) and (c) we have to implement a 3-to-1 in the second level

while for cases (d) and (e) we must implement a 2-to-1 mux. Note that, the 3-to-1 mux to be implemented for cases (b) and (c) can either be implemented by using a 3-to-1 mux directly from the library or using a tree of 2-to-1 muxes. All these cases are shown in Table 2.

To reduce the complexity of exhaustive search, we use a search technique based on heuristic cost functions. We start with a node representing *n*, the data input count of the multiplexer to be implemented. Let us call this an OR node with label *n*. Each descendent of the OR-node with label *n* represents a partition of *n* into *n1* and *n2*, $n = n1 + n2$ where $0 \le n1, n2 \le n$ and $n1 \ge n2$. We call the descendants of the OR node AND nodes. There are $\lfloor n/2 \rfloor + 1$ AND descendants of an OR node with label *n*. The children of an AND node, corresponding to the partition $<n1, n2>$, are OR nodes with labels *n1* and *n2*. This nomenclature is similar to that of AND-OR graphs [Nilsson 80]. The descendants of OR nodes represent different architectural choices. At a particular OR node we have freedom to choose one of the AND nodes. However, once an AND node is chosen, we have to build structures corresponding to both of its children. In our algorithm, we guide our choice of the AND node on the basis of cost estimates that represent the area of the final implementation of the multiplexer. Experimental results demonstrate the accuracy of the cost estimates in generating minimal-area implementations of multiplexers.

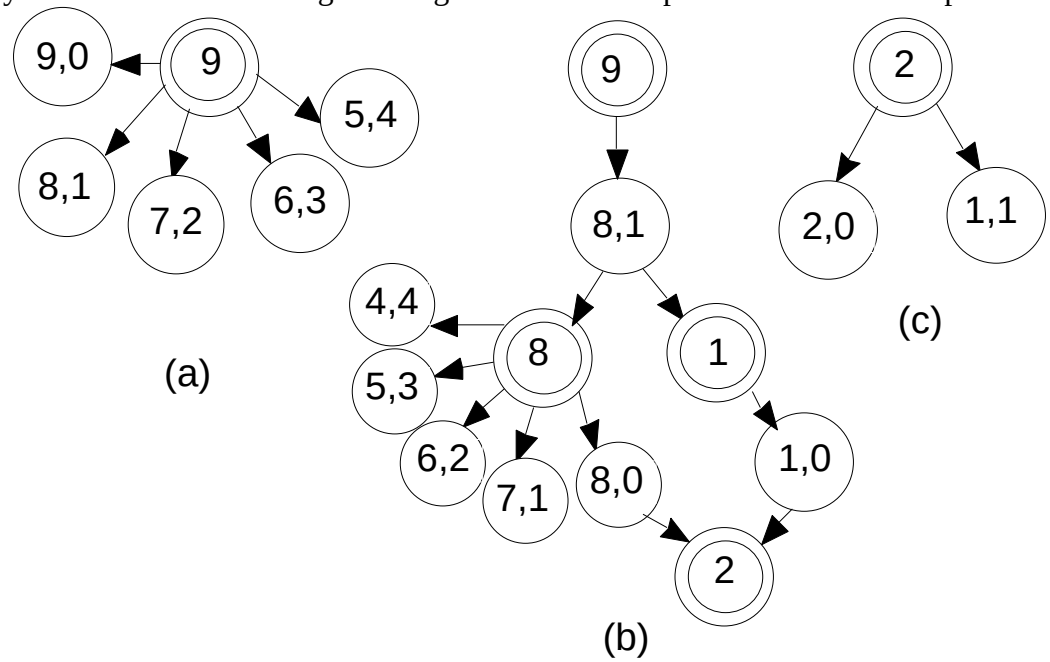



Figure 2. Synthesis of a 9-to-1 multiplexer using Library 1 (Table 1) (a) AND descendants of OR-node with label 9; (b) AND descendants of OR-nodes; (c) AND descendants of OR node 2

We first illustrate our algorithm with a simple example. In Fig. 2, we illustrate the synthesis of a 9-to-1 mux using Library 1. In Fig. 2(a), we start with an OR node representing 9, the data input size of the multiplexer to be implemented. The set of nine data inputs can be partitioned into 2-partitions in *5* ways *(9, 0), (8,1), (7, 2), (6, 3)* and *(5, 4)*. Figure 2(a) shows the AND nodes corresponding to each of these partitions. Since there is no 9-to-1 mux in the

library, we can immediately eliminate the (9, 0) case. For each AND node, we compute a cost estimate and select the AND node with the minimum cost. We describe the calculation of cost estimates later in this section. For this example, the partition *(8, 1)* yields the lowest cost and is selected. In Fig. 2(b), we show the two OR children of the node *(8, 1)*. The possible partitions for the OR node with label 8 are *(8, 0), (7, 1), (6, 2), (5, 3)* and *(4, 4)*. Since there is an 8-to-1 mux in the library, we keep *(8, 0)* as a possible partition. The partition *(8, 0)* yields the lowest cost and is selected. Since (8, 0) corresponds to the 8-to-1 mux in the library we choose that in the implementation of the 9-to-1 mux. For the OR node with label 1 (i.e. a single signal line), the only possible partition is (1, 0); this means, in the next level, we have to consider two signal lines — the output of the 8-to-1 mux just chosen and the single signal line corresponding to the OR node with label 1. Hence, we have to implement a structure corresponding to a 2-to-1 mux. As shown in Fig. 2(c), the possible partitions of the OR node with label 2 are *(2, 0)* and *(1, 1)*. The partition *(1, 1)* means two signal lines. Since the smallest multiplexer available in the library is a 2-to-1 mux, partition *(2, 0)* is our only choice. Thus, for Library 1, we implement the 9-to-1 mux using an 8-to-1 mux and a 2-to-1 mux, the total area being *50* units.

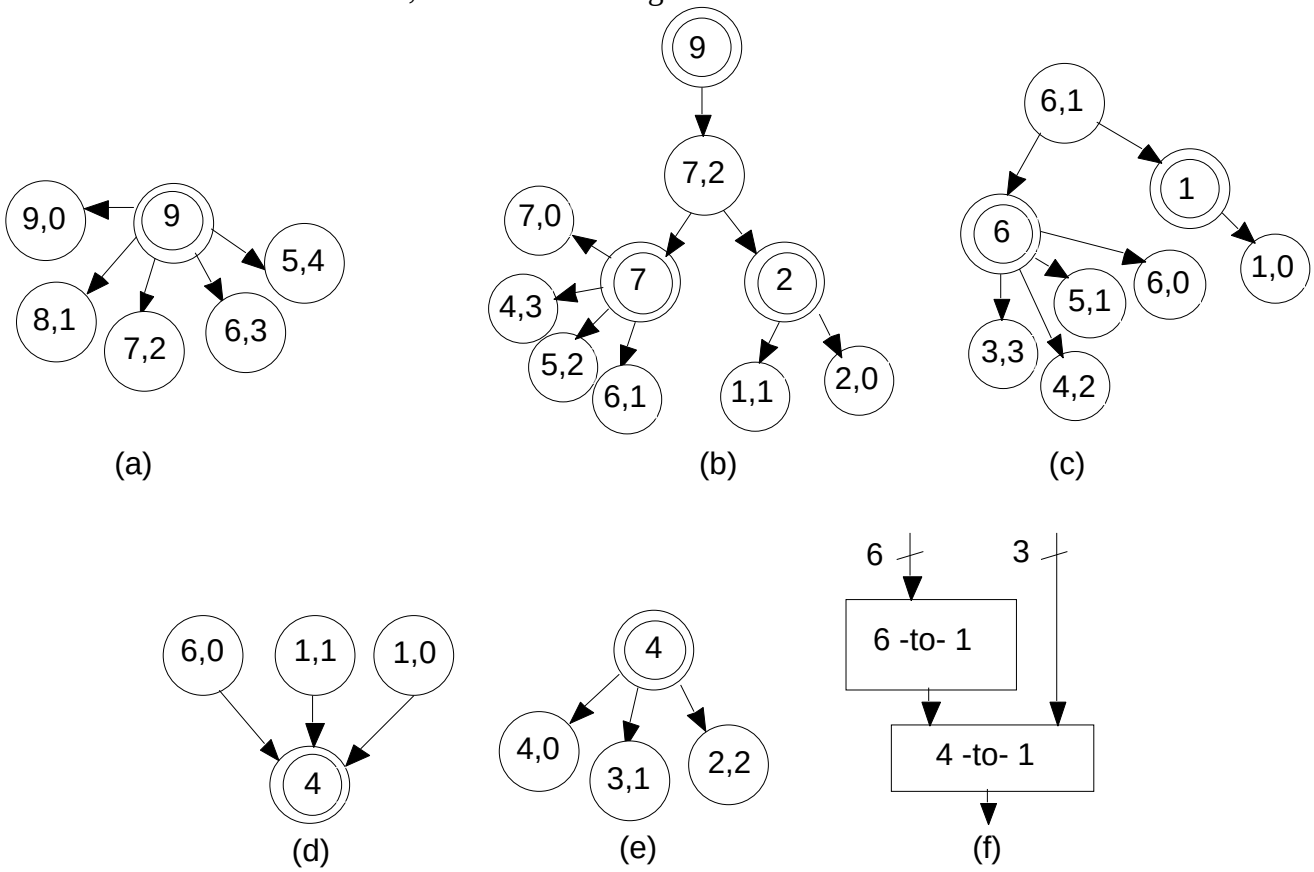



Figure 3. Synthesis of a 9-to-1 multiplexer using Library 2 (Table 1) (a) AND descendants of OR-node 9; (b) Descendants of OR-nodes 7 and 2; (c) Tree after choosing partition (8, 1); (d) Output of 8-to-1 mux and three signal lines form OR node 4; (e) Partitions of OR-node 4; (f) Final implementation

```
Algorithm syn_min_area_mux :

Inputs :
Multiplexer Library represented as a set of pairs {<n, area>};
m, for the m-to-1 multiplexer to be synthesized

Output : Minimal-area m-to-1 multiplexer using tree of component multiplexers

begin
        Determine_structure(m)
end syn_min_area_mux

Procedure Determine_structure (n):
begin
        if ( n  == 1) return;
        if ( n == 2)  {
                Find a 2-to-1 MUX from the library;
                return;
        }
        Determine_structure (Determine_next_level (n, 0));
end Determine_structure;

Procedure Determine_next_level (n , n_sibling)

Inputs :
n : current number of inputs;
n_sibling : the other part of the partition from where n was derived

Output : number of outputs of the next level

begin
        D = {(n-1, 1), (n-2, 2), …, (n-⌈n/2⌉, ⌈n/2⌉)}
        if  (there is an n-to-1 multiplexer in the input library) D = D     {n, 0}
        for each {j, n - j}     D,
                Compute the  estimated cost Cj using the information about n_sibling
        Choose the partition {n1, n2} with minimum Cj.
        if (n2 == 0) {
                implement n1-to-1 multiplexer from the library;
                num_outputs = 1
        }
        else {
                num_ouputs = Determine_next_level (n1, n2) + Determine_next_level (n2, n1);
        }
        return (num_outputs)
end Determine_next_level;
```

Figure 4. Pseudo-code of the area minimization algorithm

Figure 3 illustrates the synthesis of a 9-to-1 mux using Library 2. The decomposition into a 6-to-1 mux and a 4-to-1 mux gives the minimal area implementation of 54 units. A simple greedy strategy would choose an 8-to-1 and a 2-to-1 mux requiring an area of 56 units. However, this is not an optimal solution. Our algorithm works in the following way. We start with the OR node with label 9 as shown in Fig. 3(a). Out of the five partitions (AND nodes), the partition (7, 2) is chosen since it yields the lowest cost. The OR node with label 7 has three possible partitions out

of which (6, 1) yields the lowest cost. For the OR node 2, we have two partitions out of which (1, 1) yields the lowest cost. The OR node with label 6 has four children AND nodes out of which (6, 0) corresponds to the lowest computed cost. Corresponding to (6, 0), we implement the 6-to-1 from library 2. The children of the AND node (1, 1) are two OR nodes each of label 1, which are actually two stand-alone signal lines. A similar situation happens for the OR node with label 1 that is a child of the AND node (6, 1). We combine these three signal lines together with the output of the 6-to-1 mux to obtain an OR node with label 4 (Fig. 3(e)). Cost computations tell us to implement a 4-to-1 mux corresponding to this node. The implementation is shown in Fig. 3(f).

Figure 4 is the pseudo-code for our area minimization algorithm. Next, we describe the method of calculating the estimated costs of the partitions. If we examine the area values of the different multiplexers in Table 1, we find that the area required to implement a 3-to-1 mux, for example, using a 3-to-1 mux from the library is less than the area required if we implement the 3-to-1 mux using two 2-to-1 multiplexers in the library. This is the basic philosophy that we use to generate the different cost functions.

The cost function used to guide the heuristic algorithm consists of two components: the *local* cost of a particular partition and the *global impact* cost that may arise in the subsequent levels of the design if the current partition is chosen for implementation. Suppose we have an OR node with label *n* and we partition *n* into *(n1, n2)*. If there are *n1-to-1* and *n2-to-1* mux in the library, then the local cost will be *Area (n1-to-1) + Area (n2-to-1).* If there is no *n1-to-1* mux (or *n2-to-1* mux) in the library, we estimate the area corresponding to *n1* (*n2*) by calculating the area of the first level of multiplexers in an implementation of an *n1-to-1* (*n2-to-1*) mux using a *best-fit* strategy. The best-fit strategy selects multiplexers from the library with the largest number of data inputs, *m*, where $m \leq n1\ (n2)$. For example, for Library 1 (Table 1), for *n1* = 5, the local cost is equal to the cost of a 4-to-1 mux (*m* = 4), i.e., 19. If *n1* = 12, for Table 1, the local cost is equal to the sum of areas of an 8-to-1 and a 4-to-1 mux, i.e., 19 + 42 = 61 units.

The local cost gives us a *local* picture of the cost of a particular partition. Next, we consider the global impact cost. This cost estimate is used to evaluate the effect of a particular choice of partition on the subsequent levels of the multiplexer implementation. Suppose we choose *(n1, n2)* as the partition of *n*. If we implement the first level of *n1-to-1* and *n2-to-1* mux using the best-fit strategy, then we can calculate the number of inputs (derived from the outputs of the multiplexers and standalone signal lines) *(n1, n2)* will contribute to the next level. Let us call it *n3.* For example, for Library 1, for a partition (5, 2), the best-fit strategy would select a 4-to-1 mux corresponding to 5 and a 2-to-1 mux corresponding to 2. Thus, n3 = 3, corresponding to the outputs of the 4-to-1 and the 2-to-1 mux and the remaining signal from the partition 5 of (5, 2). Now, suppose that node *n* is a child of an AND node *(m, n)*, i.e., node *n* has a sibling (another child of *(m, n)*) OR node with label *m*. The node *m* also contributes to the global impact cost,

which is estimated by the number of inputs that will be there in the next level if the node *m* is implemented using the best-fit strategy. Let this contribution be *m3*. For example suppose we want to estimate the global impact cost of a partition (AND node) (5, 2). Its parent is an OR node with label 7. Now, suppose that the parent of this OR node is an AND node representing the partition (7, 4). Thus, node 7 has a sibling which is an OR node with label 4. If this OR-node is implemented using the best-fit strategy, then we will use a 4-to-1 mux (since such a multiplexer exists in the library). Therefore, m3 = 1 that represents the output of that 4-to-1 mux. Now, the global impact cost of the partition of *n* into *n1* and *n2* is computed as the cost of implementing the first level of an *(n3 + m3)-to-1* mux using the best-fit strategy. Thus, we estimate the global impact cost of implementing the next (second) level of (5, 2) as the cost of implementing a 4-to-1 mux (since n3 = 3 and m3 = 1) using the best fit strategy.

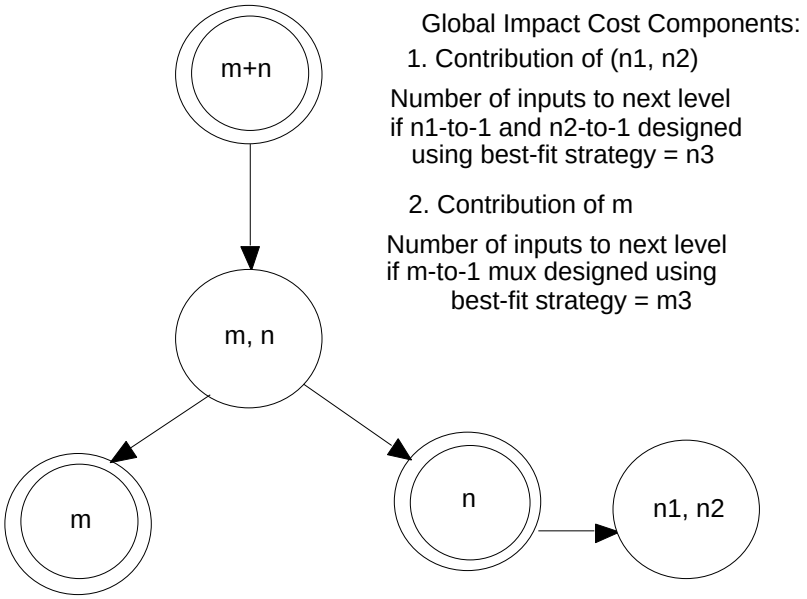



Figure 5. Global Impact Cost Computation

Figure 5 shows the different components of the global impact cost if we choose the AND node (*n1, n2*) during our search process. In a similar way, we can estimate the impact of the choice of *(n1, n2)* on subsequent levels. Our algorithm has the flexibility of considering additional levels of the tree. The sum of the *global impact* cost and the *local* cost gives the total *estimated* cost. We illustrate the cost calculation method using some examples.

The cost of the partition *(8, 1)* of Fig. 2 is computed as follows:

- The *local* cost = Area of the 8-to-1 mux = 42 units
- The *global impact* cost = Area of a 2-to-1 mux in the next level (the output of the 8-to-1 mux and the signal line corresponding to the child of (8,1) with label 1) = 8 units
- *Total estimated* cost = 42 + 8 = 50 units

Now, we calculate the cost of the *(7, 2)* partition in Fig. 2:

- The *local* cost = Area of a 6-to-1 mux + Area of a 2-to-1 mux = 33 + 8 = 41 units
- The *global impact* cost = Area of a 3-to-1 mux (n3 = 3) having inputs from the output of 6-to-1 mux (since best-fit strategy chooses a 6-to-1 to implement the 7 of (7, 2)), output of the 2-to-1 (best-fit strategy chooses a 2-to-1 mux to implement the 2 of (7, 2)) and the remaining signal line = 14 units
- *Total estimated* cost = 41 + 14 = 55 units

Now, we show how to compute the cost of the partition (8, 0) of Fig. 2.

- The *local* cost = Area of the 8 -to- 1 mux = 42 units
- The *global impact* cost: n3 = 1, output of the 8-to-1 mux; m3 = 1, contributed by the OR node with label 1 which is a child of the AND node (8, 1). Global impact cost = area of 2-to-1 mux = 8 units
- *Total estimated* cost = 42 + 8 = 50 units

For Fig. 3 (Library 2), we compute the cost of the AND node (2, 0) as follows:

- The *local* cost = Area of the 2-to-1 multiplexer = 8 units
- The *global impact* cost: n3 = 1, output of the 2-to-1 mux; m3 = 2 (output of the 6-to-1 mux and the remaining signal if the OR node 7 is implemented using the best-fit strategy). Global impact cost = Area of 3-to-1 mux = 14 units.
- *Total estimated* cost = 8 + 14 = 22 units

For Fig. 3 (Library 2), we the cost of the AND node (1, 1) is:

- The *local* cost = 0 units
- The *global impact* cost: n3 = 2; m3 = 2 (output of the 6-to-1 mux and the remaining signal if the OR node 7 is implemented using the best-fit strategy). Global impact cost = Area of 4-to-1 = 21 units.
- *Total estimated* cost = 0 + 21 = 21 units

Figure 6 shows the algorithm for computing the estimated costs of a particular partition. It may be noted that our implemented algorithm has the flexibility of computing the total estimated cost after considering the effects of additional levels of the search tree.

We have experimented with different libraries containing component multiplexers with different areas. Our algorithm generates multiplexers with minimal areas for these cases. The results are reported in Sec. 5.

```
Cost Function Calculation

Input:
A partition <m, n>, which is an OR node in the search tree

Output:
Estimated Cost of<m, n>

Local Cost (m)
        If (m == 0) return (0);
        If (m == 1) return (0);
        Find the largest k, k ≤ m such that, there is a k-to-1 multiplexer in the library
        return (Area (k-to-1) + Local Cost (m-k));
end Local Cost;

Next-level Input (m):
        If (m == 0) return (0); If (m == 1) return (1);
        Find the largest k, k ≤ m such that, k-to-1 multiplexer in the library
        return (1 + Next-level Input (m-k));
end Next-level Input;

Estimated Cost (<m, n>):
        Net Local Cost = Local Cost(m) + Local Cost(n);
        First-level Input = Next-level Input(m) + Next-level Input(n);
        Node_(m+n) = Parent of (<m, n>);
        Node_p = sibling of (Node_(m+n));
        Second-level Input = Next-level Input(p);
        Net Global Impact Cost = Local Cost(First-level Input + Second-level Input);
        return(Net Local Cost + Net Global Impact Cost);
end Estimated Cost;
```

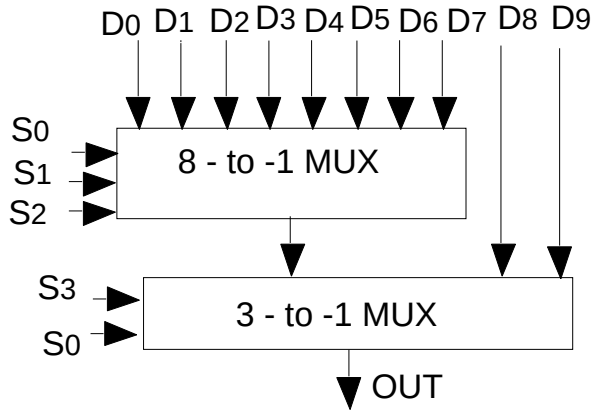
Figure 6. Pseudo-code for the cost computation algorithm

## 4. SYNTHESIS OF MULTIPLEXER ADDRESS SIGNALS

In this section, we consider the problem of generating address signals for the component multiplexers that implement the synthesized multiplexer. The decomposition of a given multiplexer with $n$ data inputs into a tree of multiplexers of different sizes may require greater than $\lceil \log_2 n \rceil$ address signals. If we assume that the generated multiplexers have fully encoded address inputs, i.e., $\lceil \log_2 n \rceil$ address inputs, then extra logic may be required to translate the input address signals into the component address signals.

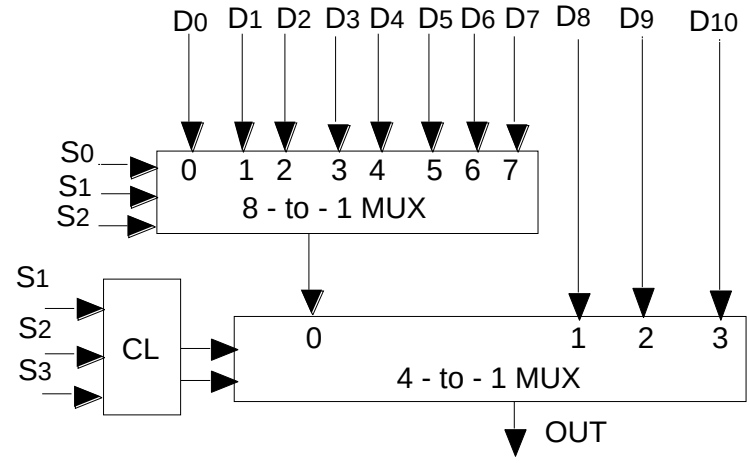
Figures 7 and 8 illustrate the synthesis of the component multiplexer address signals for a 10-to-1 mux and an 11-to-1 mux, respectively. The 10-to-1 mux, utilizing one 8-to-1 mux and a 3-to-1 mux, does not require extra logic or extra signals for the address inputs because signal S0 is assigned to the address input of both component multiplexers. However, the implementation of an 11-to-1 mux with an 8-to-1 mux and a 4-to-1 mux requires extra logic (shown in Fig. 8) if we require that the number of address signals of our implemented multiplexer is minimum (4); otherwise we require extra signals (5, in this case). In this section, we present an algorithm that, given a tree of multiplexer components and the desired number of address signals, tells us whether

or not extra logic is necessary for the address signals. Next, we introduce the concept of *use-sets*.




| S0 | S1 | S2 | S3 | OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | D0 |
| 0 | 0 | 1 | 0 | D1 |
| 0 | 1 | 0 | 0 | D2 |
| 0 | 1 | 1 | 0 | D3 |
| 1 | 0 | 0 | 0 | D4 |
| 1 | 0 | 1 | 0 | D5 |
| 1 | 1 | 0 | 0 | D6 |
| 1 | 1 | 1 | 0 | D7 |
| 0 | d | d | 1 | D8 |
| 1 | d | d | 1 | D9 |

Figure 7. 10-to-1 Multiplexer Implementation




| S0 | S1 | S2 | S3 | OUT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | D0 |
| 0 | 0 | 1 | 0 | D1 |
| 0 | 1 | 0 | 0 | D2 |
| 0 | 1 | 1 | 0 | D3 |
| 1 | 0 | 0 | 0 | D4 |
| 1 | 0 | 1 | 0 | D5 |
| 1 | 1 | 0 | 0 | D6 |
| 1 | 1 | 1 | 0 | D7 |
| d | 0 | d | 1 | D8 |
| d | 1 | 0 | 1 | D9 |
| d | 1 | 1 | 1 | D10 |

Figure 8. 11-to-1 Mux requiring extra address signal logic

**Definition:** Given a tree of component multiplexers, the *use-set* of a particular data signal is defined as the union of the address signals of all the multiplexers lying on all paths from the primary data inputs to that signal. The use-set of a primary data input signal is *NULL*.

For example, in Fig. 7, the *use-set* of the output signal of the 8-to-1 multiplexer is {S0, S1, S2} and that of the signal OUT is {S0, S1, S2, S3}. The use-set of the output signal of multiplexer $M_i$ is the union of use-sets of its data input signals and the set of address signals of $M_i$. We use the concept of use-sets to formulate Rule 1, described next.

**Rule 1:** The signal assigned to the address input of a 2-to-1 multiplexer cannot belong to the *use-sets* of any of its data inputs.

**Proof:** The proof is straightforward. Suppose $s_i$ is a signal belonging to the *use-set* of the input signal $l_i$ of the 2-to-1 multiplexer. This implies that there is a multiplexer, $M_j$, on the path from the primary inputs to $l_i$ which has $s_i$ assigned to one of its address inputs. This means that with $s_i$ =

0, we choose a particular input $l_{j1}$ of $M_j$ and with $s_i = 1$, we choose another input $l_{j2}$ of $M_j$. However, if $s_i$ is assigned to an address input of $M_i$ such that $l_i$ is chosen when $s_i = 0$, then we will always choose $l_{j1}$ and never $l_{j2}$. The case of choosing $l_i$ with $s_i = 1$ is symmetric. Q.E.D.

Rule 1 forms the basis for our algorithm, shown in Fig. 9, which determines whether a given tree of multiplexers can be implemented with the specified number of address signals connected directly to the address inputs of the components. The algorithm returns either an assignment of address signals or an indication that an assignment is impossible. Incomplete multiplexers have multiple decompositions into trees of 2-to-1's depending on the truth table of a specific implementation. We explain the working principle of our algorithm using the examples in Figs. 7 and 8.

```
Algorithm Determine_control

Input :
A tree of component multiplexers
Select signals S = {s1 , s2 ,..., sm}.

Output :
Assignment of control signals to multiplexer select inputs, if possible
-1, otherwise  /*** indicating failure ***/

begin
        for each primary input signal p_i ,
                use_set[ p_i ] = NULL;
        Order the select signals as s1 , s2 ,..., sm.
        done = FALSE;
        while not done {
                Select the next multiplexer M_j so that
                        (i) select signals are unassigned and
                        (ii) use-sets of all data inputs computed
                if no such M_j is present, return(assignment of control signals)
                Sort the data inputs of M_j in ascending order of the cardinality of their use-sets.
                Decompose M_j into a tree of 2-to-1 multiplexers to generate decomp_j.
                Assign the data inputs of M_j to the data inputs of the 2-to-1 multiplexers of the tree such
that:
                Data input k in the sorted list is not assigned to a lower level 2-to-1 mux than data input l, k
< l.
                While decompj is not empty {
                        Select the 2-to-1 m_k   decomp_j such that
                                (i) address input is unassigned
                                (ii) use-sets of all data inputs computed
                        Assign the first select signal to the address input of m_k so that Rule 1 is satisfied.
                        If the union of the use-sets of the two data inputs of m_k is S return(-1);
                        Compute the use-set of the output signal of m_k;
                        Remove m_k from decomp_j
                endwhile;
                Compute the use-set of the output signal of M.
        endwhile;
end Determine_Control
```

Figure 9. Pseudo-code of Determine_Control

We apply our algorithm, Determine_control, to the 10-to-1 mux in Fig. 7 and obtain the decomposition into 2-to-1 muxes shown in Fig. 10. The use-sets of the data inputs D0-D9 are all *NULL*. The use-sets of the other signals and the assignment of signals to the address inputs are shown in Fig. 10. Since the use-sets of D8 and D9 are *NULL*, we can assign s0 to the address input of the 2-to-1 multiplexer having D8 and D9 as data inputs. So, we do not need an extra address signal. Figure 11 shows the result of applying our algorithm Determine_control to the 11-to-1 mux of Fig. 8. The minimum number of address signals required by an 11-to-1 mux is 4. Our algorithm determines that for the implementation shown in Fig. 8, additional logic is required to implement the multiplexer with 4 address signals.

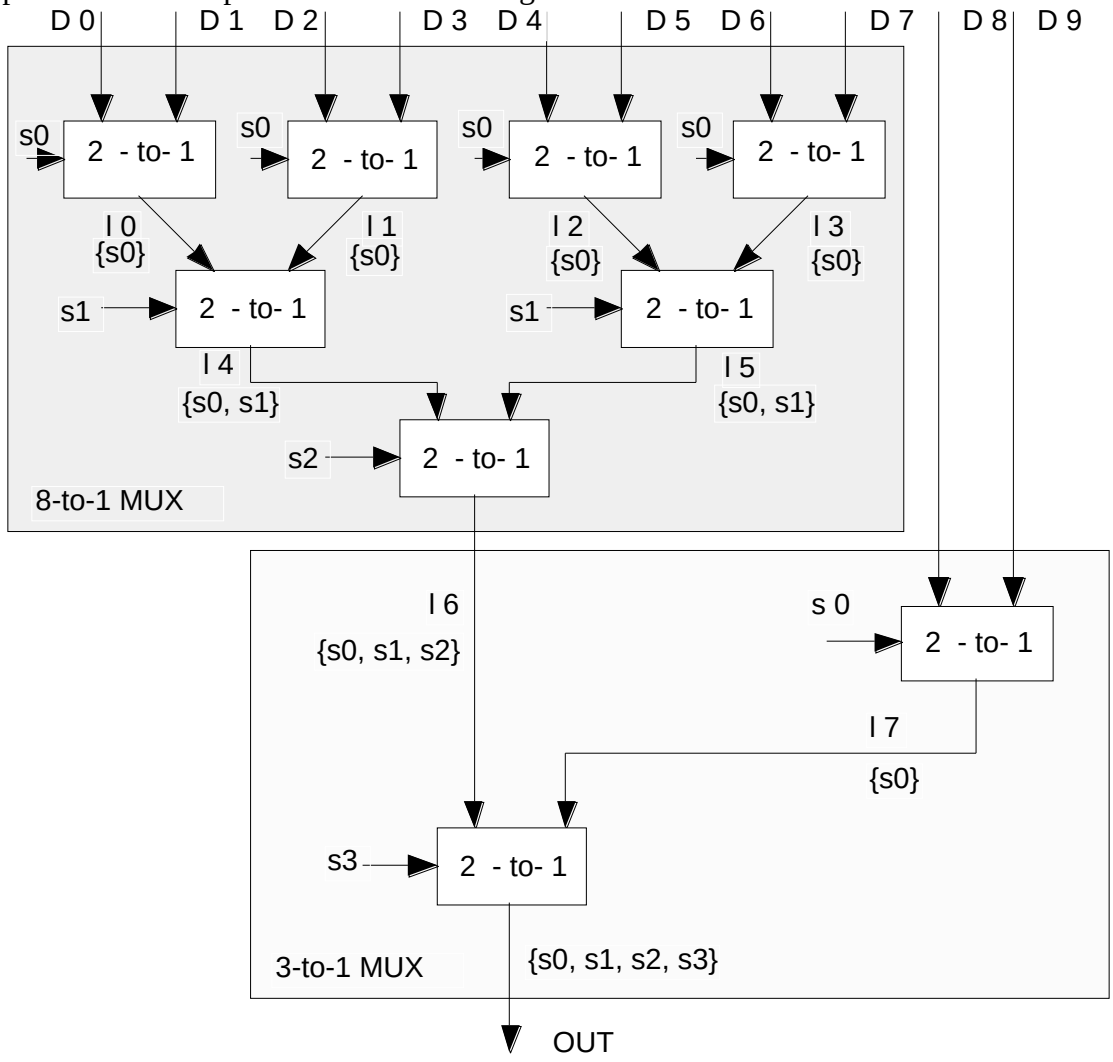



Figure 10. Assignment of address signals of 10-to-1 MUX

Different strategies may be adopted when Determine_Control reports failure. The simplest solution is to allocate extra address signals. Another strategy is to consider a set of minimal cost decompositions at each step of the algorithm. For example, as shown later in Table 3, for the 14-to-1 mux, the minimum multiplexer generated by our algorithm needs 80 units of area and more

than 4 address signals. If we consider a set of minimal cost decompositions at each step of the algorithm, then we can implement it using 83 units of area and 4 address signals. The third option is to add translation logic to generate the extra address signals required. The first approach, which transfers the responsibility of the generation of extra address signals to the control logic, is suitable for a high-level synthesis tool where address logic for multiple multiplexers may be shared. Tables 3 and 4 show that the second strategy of choosing an alternative implementation produces good results.

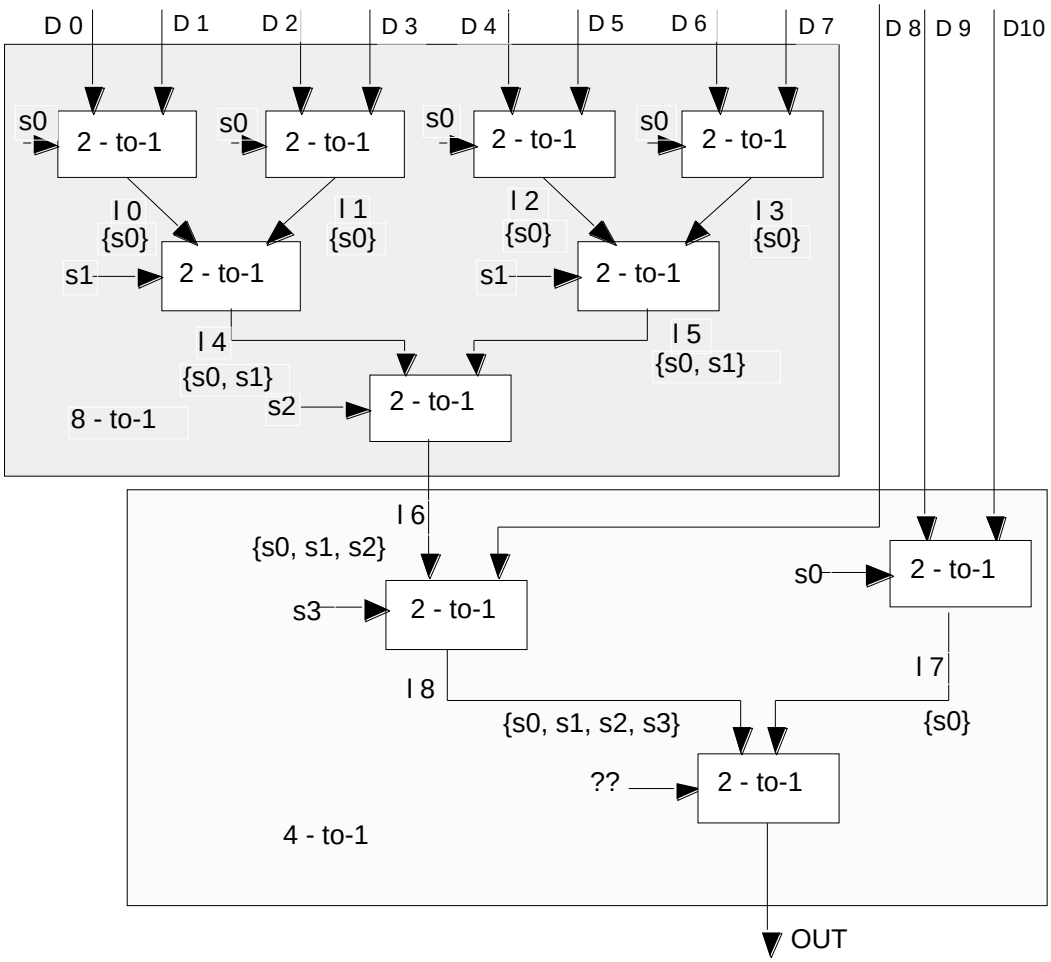



Figure 11. Assignment of address signals of 11-to-1 MUX

## 5. EXPERIMENTAL RESULTS

In this section, we present a comparison of the areas of the multiplexers generated by our algorithm presented in this paper with those synthesized by existing synthesis tools. First, we show the results of our algorithm (decomposition into component multiplexers) in Table 3. For our algorithm, we first generated the minimum-area decomposition (Component Muxes (1) column). Next, if Determine_Control returned a failure (marked by asteriks), we iterated to find a decomposition by considering a set of minimal decompositions at each step of our algorithm

(Component Muxes (2) column). For all cases, our algorithm could generate the minimum-area implementation, which we validated by running an exhaustive algorithm for synthesizing multiplexers using a tree of multiplexer components. The comparison with an exhaustive technique is shown in Appendix A. The CPU time required to execute our algorithm is extremely small — it takes around 0.9 microseconds on a Sun Ultra2 workstation to synthesize a 20-to-1 multiplexer.

We compare the area results of the multiplexers generated by our technique with those generated by commercial tools referred to as Tool A, B and C in Table 4. For Tool A, a commercial RTL synthesis tool, we wrote verilog code with *case* statements to specify the multiplexer operation. We specified an incomplete truth table (corresponding to the incomplete multiplexer) using case statements with *full_case* annotation. This is shown in Tool A(1) column of Table 4. We ran Tool A with some special multiplexer components instantiated. The results are shown under column Tool A(2) of Table 4. The LSI Logic G10-p library [LSI 96] was used as the mapping library.

Table 3. Component multiplexers generated by our algorithm

| # Data inputs | Component MUXes (1) | Component MUXes (2) |
|---|---|---|
| 9 | 8, 2 † % & | — |
| 10 | 8, 3 † % & | — |
| 11 | 8, 4* † % & | 8, 3, 2 † % & |
| 12 | 8, 4, 2 † % & | — |
| 13 | 8, 4, 3 † | — |
| | 8, 6 * % | 8, 4, 3 % |
| | 8, 2, 4, 2 & | — |
| 14 | 8, 4, 4* † % | 8, 6, 2 † % |
| | 8, 2, 4, 2, 2 & | — |
| 15 | 2, 4, 8, 4* † % | 4, 4, 4, 3, 4 † % |
| | 8, 3, 2, 4, 2 & | — |
| 16 | 8, 8, 2 † % & | — |
| 17 | 8, 8, 3 † & | — |
| | 4, 8, 4, 4 % | — |
| 18 | 8, 8, 4 † % & | — |
| 19 | 8, 8, 4, 2* † % & | 8, 8, 3, 3 † % & |
| 20 | 8, 8, 6 † & | — |
| | 8, 8, 3, 4 % | — |

† G10p Library, % LCB500K Library, & LCA300K Library, * Requires extra address signals

Tool B is used to generate regular structures, such as multiplexers and counters, and it always generates multiplexers out of component multiplexers in the library. However, it cannot generate multiplexers with more than 16 data inputs. The version of Tool B available to us cannot be used for the LSI Logic G10-p library [LSI 96] but can be used for LSI Logic LCB500K library

[LSI 95]. The area result comparison with Tool B is shown in Table 4.

Tool C is another commercial RTL synthesis tool. However, our version of Tool C supports LCA300K [LSI 93] library. For Tool C also, we gave as input, case statements written in Verilog. For incomplete multiplexers, we used *full_case* annotation. The comparisons of the areas are shown in Table 4.

As shown in Table 4, Tool A (in case 1) always generates bigger multiplexers than our algorithm. In fact, our algorithm could always generate the minimum decomposition. Tool A, for case 2, can generate the minimum decomposition only for the 16-to-1 mux case. It can also be seen from Table 4 that Tool B performs well. Our tool always performs as well as, or better than, Tool B. However, Tool B cannot generate multiplexers with more than 16 data inputs. Table 4 also shows that we consistently generate multiplexers requiring less area compared to Tool C. We found that our algorithm generated minimum area multiplexers for all the cases. Note that, for all these cases we ran the tools aiming at area optimization and the highest effort option for mapping. In the tables, we have reported in bold the area figures for our algorithm and the tools A, B and C whenever they produce the minimum area implementation. The experimental results report areas for multiplexers with up to 20 data inputs. This is because, multiplexers with more than 20 data inputs are typically implemented using tristate gates.

Table 4. Area Comparisons with Tool A, B and C

| # Data inputs | G10-p Library | | | LCB500K Library | | LCA300K Library | |
|---|---|---|---|---|---|---|---|
| | New Algorithm | Tool $A_1$ | Tool $A_2$ | New Algorithm | Tool B | New Algorithm | Tool C |
| 9 | **50** | 56 | 60 | **48** | 50 | **14** | 25 |
| 10 | **56** | 70 | 62 | **54** | 56 | **16** | 27 |
| 11 | **64** | 90 | 74 | **62** | **62** | **18** | 27.5 |
| 12 | **69** | 91 | 76 | **66** | 68 | **20** | 29.5 |
| 13 | **75** | 107 | 85 | **72** | **72** | **22** | 34.5 |
| 14 | **83** | 104 | 88 | **79** | 80 | **24** | 32 |
| 15 | **90** | 101 | 97 | **86** | **86** | **26** | 30 |
| 16 | **92** | 111 | **92** | **88** | 90 | **26** | **26** |
| 17 | **98** | 121 | 111 | **94** | — | **28** | 47 |
| 18 | **103** | 130 | 113 | **98** | — | **30** | 48 |
| 19 | **112** | 139 | 123 | **108** | — | **32** | 52 |
| 20 | **117** | 145 | 125 | **112** | — | **34** | 55 |

## 6. Delay Minimization Algorithm

In this section, we minimize the delay of the multiplexers generated by our area-minimization algorithm. Unlike [Thakur 96], which considers decomposition of a given multiplexer into a tree of 2-to-1 multiplexers, we consider the components determined by our area minimization algorithm. There are two basic approaches to delay minimization:

1. Delay minimization maintaining the level information generated by the area minimization algorithm

2. Delay minimization on the components generated by the area minimization algorithm without maintaining the level information generated by the algorithm

The inputs to our algorithm are the *arrival times* of the data and the address input signals and the *worst case* delay of each library multiplexer component. The library [LSI 96] shows that different data inputs have different propagation delays through the multiplexer. We use this information in our algorithm. As mentioned earlier, the first approach retains the level information generated by the area minimization algorithm. The second approach, while producing lesser delay implementation, sometimes adds an extra overhead because the structure may require extra address signals or extra logic to generate these signals. We illustrate both of these approaches using Fig. 12 and Fig. 13.

As shown in Fig. 12, we have built a 6-to-1 multiplexer using a 4-to-1 mux and two 2-to-1 muxes. The data inputs D0, D1, D2 and D3 arrive at time 0; D4 and D5 arrive at times 6 and 8, respectively. c0 and c1 arrive at time 0. c2 arrives at time 4. The area minimization algorithm generated the information that there is a 4-to-1 mux and a 2-to-1 mux in the first level and a 2-to-1 mux in the second level. The *worst case* delay through a 4-to-1 mux is 5 units and that through a 2-to-1 mux is 3 units. However, when data input D2 or D3 is chosen, the delay through a 4-to-1 mux is 4 units; when data input D4 (or D6) is chosen, the delay through a 2-to-1 mux is 3 units.

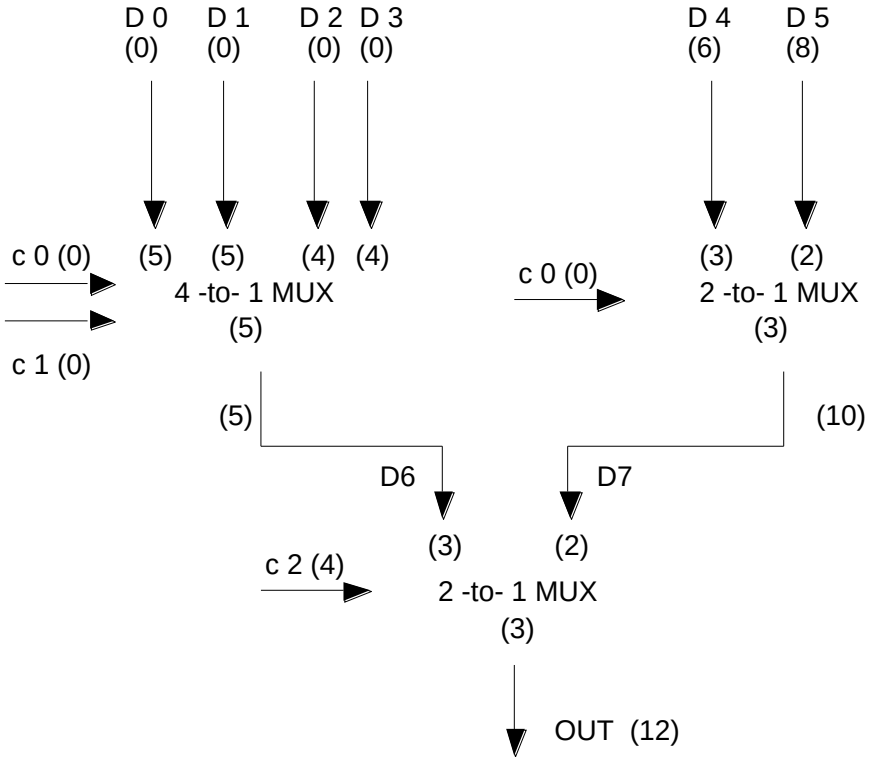



Figure 12. Delay Minimization of a 6-to-1 MUX implementation keeping the level information

According to Fig. 12, at each level, we sort the signal lines, to be connected to the

multiplexer data inputs in that level, in ascending order of their arrival times. Now, we assign the next earliest arriving signal to the currently available multiplexer (in that level) with the highest delay. Thus, we know that D0, D1, D2 and D3 will be assigned to the inputs of the 4-to-1 mux while D5 and D6 will be assigned to the inputs of the 2-to-1 mux. For each multiplexer, we assign the earliest arriving input signal (assigned to its data inputs) to the highest delay input pin. Since D0, D1, D2 and D3 have same arrival times, any ordering among them works for the 4-to-1 mux. For the 2-to-1 mux, D4 arrives at time 6 and D5 arrives at time 8. Hence, D4 and D5 are assigned to data inputs with delays 3 and 8, respectively. Hence, the arrival time of the output of the 2-to-1 mux in the first level is 10. A similar strategy is taken for the 2-to-1 mux in the second level. The address signal assignment can be done in a similar way.

The implementation generated using this approach is not guaranteed to have the minimum delay. A smaller delay implementation can be obtained if we do not consider the level information generated by our area minimization algorithm. This is illustrated in Fig. 13.

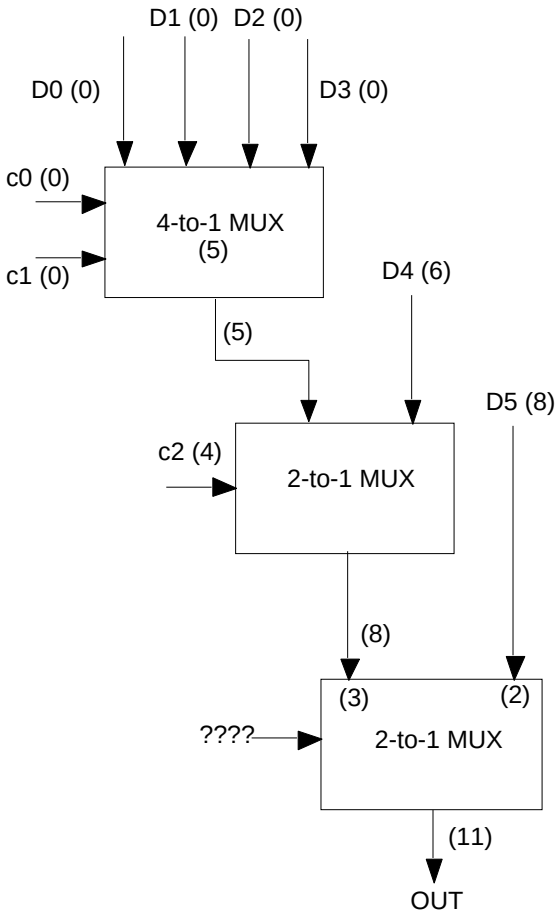



Figure 13. Implementation of a 6-to-1 MUX using a given set of component multiplexers (no level information)

For the implementation in Fig. 13, we sort the set of inputs in ascending order of their

arrival times. Then, we assign the earliest arriving input signals to the currently unassigned multiplexer with the highest delay. The arrival time of the output signal of a multiplexer can be calculated from the arrival times of the data inputs of that multiplexer and the delay of that multiplexer. Next, we add the output signal of that component to the set of currently unassigned input signals and repeat the above procedure. However, when assigning inputs to a given component, we use the same strategy that we used in our first approach. Thus, D0, D1, D2 and D3 are assigned to the 4-to-1 mux. Its output signal has arrival time 5. Hence, the output signal is paired up with D4 to go to the input of the 2-to-1 mux. The output signal of the 2-to-1 mux and D6 are the inputs to the remaining 2-to-1 mux. The arrival time of OUT and hence, the worst-case delay of the 6-to-1 mux is 11 units. However, as shown in Fig. 13, such a structure cannot be implemented without extra logic for address signals. Thus, in our effort to minimize the delay of a multiplexer implementation, we have increased its effective area.

Table 5. Comparison of multiplexer delays (G10p Library)

| # Data Inputs | New Algorithm (ns) | Tool A[1] (ns) | Tool A[2] (ns) |
|---|---|---|---|
| 9 | 1.05 | **1.00** | 1.04 |
| 10 | 1.10 | **0.99** | 1.05 |
| 11 | **1.05** | 1.33 | **1.05** |
| 12 | **1.05** | 1.16 | 1.08 |
| 13 | 1.10 | 1.20 | **1.07** |
| 14 | **1.05** | 1.26 | 1.07 |
| 15 | **0.90** | 1.55 | 1.08 |
| 16 | **1.06** | 1.27 | **1.06** |
| 17 | **1.13** | 1.39 | 1.17 |
| 18 | **1.17** | 1.40 | 1.19 |
| 19 | **1.14** | 1.64 | **1.14** |
| 20 | **1.13** | 1.85 | 1.17 |

The delay values obtained by applying the above algorithms to the minimal area multiplexer implementation, generated by our area-minimization algorithm for the G10p library (Table 1), are shown in Table 5. The delay values have been computed by Tool A after we input our implementations to Tool A using structural Verilog. Then, using Tool A (in cases 1 and 2, as described in the experimental results section), we obtained the delay values of the implementations of the different multiplexers generated by Tool A (reported in Table 3). The results in Table 5 indicate that the multiplexer implementations generated using our technique are efficient in terms of both area and delay.

## 7. CONCLUSIONS

In this paper, we have presented a new algorithm for synthesizing area-efficient multiplexers consisting of a tree of component multiplexers from a given library. Our technique first generates area-efficient multiplexers using component multiplexers from the given library. Next, we assign signals to the address inputs of the multiplexers. In this paper, we have presented

a novel approach to assigning signals to the address inputs of the component multiplexers. Finally, our technique minimizes the delay of the area-efficient multiplexer synthesized in the previous step. The delay minimization algorithm takes into account the delays of the component multiplexers and the arrival times of the data and address input signals in order to minimize the delay of the resulting multiplexer implementation. Experimental results show that our technique generates multiplexers that are more area-efficient than those generated by commercial tools that are used for multiplexer synthesis. As shown in Table 5, the multiplexers generated by our algorithm are efficient in terms of delay also. One big advantage of our technique is that our algorithm does not require a full truth table as input for synthesizing incomplete multiplexers — it has the capability of deciding the most efficient area implementation and hence the truth table. The procedure is well-suited for use in high-level synthesis tools and has been implemented in TOPS, Stanford CRC's Totally-Optimized-Synthesis for test tool [TOPS 97]. In this work, area minimization is our primary objective; however, an interesting extension of this work will be to consider both area and delay efficiency in a single algorithm rather than following a two-step technique of generating a minimal area implementation and subsequently minimizing the delay of that implementation. We are further looking into techniques for generating efficient multiplexers using multiplexer components and some other basic gates (like AND, OR, NANDs, etc) in the library. Another interesting extension of this problem is to obtain a minimum delay decomposition of a multiplexer of given number of inputs into smaller multiplexers in the library, not necessarily into 2-to-1 multiplexers only.

## 8. ACKNOWLEDGEMENTS

The authors would like to thank Dr. Jonathan T. Y. Chang, Dr. Santiago Fernandez-Gomez, Dr. Samy Makar and Dr. Robert B. Norwood of Stanford Center For Reliable Computing for their helpful comments regarding this work. This work was supported by the Advanced Research Projects Agency under Contract No. DABT-94-C-0045 and DABT63-97-C-0024..

## 9. REFERENCES

[Ambit 97] *BuildGates™ User Guide Release 2.0,* Ambit Design Systems, December, 1997.

[Burks 54] Burks, A. W., *et. al.*, "Complete Decoding Nets: General Theory and Minimality," *Journal of SIAM*, Vol. 2, No. 4, pp. 201-243, 1954.

[Detjens 87] Detjens, E., G. Gannot, R. Rudell, A Sangiovanni Vincentelli and A. Wang, "Technology mapping in MIS," *Intl. Conf. on Computer Aided Design Proc.*, pp. 116-119, 1987.

[Keutzer 87] Keutzer, K., "Dagon: Technology binding and local optimization by DAG matching," *Design Automation Conference Proc.*, pp. 617-623, 1987.

[LSI 93] *LCA300K Gate Array 5 Volt Series Products Databook*, LSI Logic Corporation, October 1993.

[LSI 95] *LCB500K Preliminary Design Manual*, LSI Logic Corporation, USA, June 1995.

[LSI 96] *G10-p Cell-Based ASIC Products Databook*, LSI Logic Corporation, USA, May 1996.

[Makar 88] Makar, S.R. and E. J. McCluskey, "On The Testing of Multiplexers," *Intl. Test Conference Proc.*, pp. 669-679, 1988.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, Eaglewood Cliffs, NJ, USA, 1986.

[Murgai 92] Murgai, R., R. K. Brayton and A. L. Sangiovanni Vincentelli, "An improved synthesis algorithm for multiplexer based PGAs," *Design Automation Conference Proc.*, pp. 380-386, 1992.

[Nilsson 80] Nisson, Nils J., *Principles of Artificial Intelligence*, Tioga Publishing Company, CA, USA, 1980.

[Synopsys 98] *DesignWare Components Quick Reference Guide*, *Version 1998.02*, Synopsys, February, 1998.

[Thakur 96] Thakur, S., D. F. Wong and S. Krishnamoorthy, "Delay Minimal Decomposition of Multiplexers in Technology Mapping," *Design Automation Conference Proc.*, pp. 254-257, 1996.

[TOPS 97] TOPS, Web address*: http://www-crc.stanford.edu/ projects/topsSummary.html*

## Appendix A

Table A.1: Comparison of our results with an exhaustive technique

| MUX data input | Our Algorithm | | | Our Algorithm iterated | | | Min Area Exh. |
|---|---|---|---|---|---|---|---|
| | Comp. MUXes | area | # select signals | Comp.MUXes | area | # select signals | |
| 9 | 8, 2 | 50 | 4 | — | — | — | 50 |
| 10 | 8, 3 | 56 | 4 | — | — | — | 56 |
| 11 | 8, 4 | 61 | 5 | 8, 3, 2 | 64 | 4 | 64 |
| 12 | 8, 4, 2 | 69 | 4 | — | — | — | 69 |
| 13 | 8, 4, 3 | 75 | 4 | — | — | — | 75 |
| 14 | 8, 4, 4 | 80 | 5 | 8, 6, 2 | 83 | 4 | 83 |
| 15 | 2, 4, 8, 4 | 88 | 5 | 8, 6, 3 | 89 | 4 | 89 |
| 16 | 8, 8, 2 | 92 | 4 | — | — | — | 92 |
| 17 | 8, 8, 3 | 98 | 5 | — | — | — | 98 |
| 18 | 8, 8, 4 | 103 | 5 | — | — | — | 103 |
| 19 | 8, 8, 4, 2 | 111 | 6 | 8, 8, 3, 3 | 112 | 5 | 112 |
| 20 | 8, 8, 6 | 117 | 5 | | — | — | 117 |

In this section, for Library 1, we show a comparison of the areas of multiplexers generated

by our algorithm with those generated by an exhaustive search (Min Area Exh. column) For our algorithm, we first generated the minimum-area decomposition (Component Muxes (1) column). Next, if Determine_Control returned a failure (marked by asteriks), we iterated to find a decomposition such that the implementation requires the minimum number of address signals. For example, while generating a 12 input multiplexer, our area algorithm first generated a tree of 8-to-1 and a 4-to-1 multiplexer. Although this implementation requires 61 units of area, we need 5 address signals. So we iterated and generated a tree consisting of 8-to-1, 3-to-1 and 2-to-1 multiplexers that requires 64 units of area and 4 address signals — an exhaustive search procedure also generates an implementation of 64 units when restricted to 4 address signals. It may be noted that, although the general multiplexer technology mapping problem is NP-complete, the problem can be solved using an exhaustive technique if the number of data inputs of the synthesized multiplexer is small.