Efficient Multiplexer Synthesis Techniques

Subhasish Mitra

LaNae J. Avra

Edward J. McCluskey
Center for Reliable Computing, Stanford University

A heuristic algorithm synthesizes efficient multiplexers consisting of a tree of multiplexer components from a technology library. The synthesized multiplexer structures are more area- and delay-efficient than those generated by commercial tools.

A MULTIPLEXER (MUX) IS A STANDARD complex gate often used in data-path logic to provide multiple connections among computation units. Typically, multiplexers (muxes) are required for routing operands to operators in the data path. In CAD tools for high-level synthesis, muxes serve to implement register sharing among several variables having nonoverlapping lifetimes. During register-transfer-level (RTL) synthesis, muxes are generated corresponding to the if-then-else and the case statements in Verilog or VHDL RTL design descriptions. Existing logic-synthesis tools do not handle muxes efficiently during technology-mapping because conventional technology mapping algorithms based on graph matching[1,2] typically cannot effectively utilize a rich technology library. This is because complex library cells, such as a 4-to-1 mux, have many distinct tree representations, and generation, storage, and search of all possible tree representations are often difficult.

In this article we describe new techniques for synthesizing muxes that are efficient in terms of area and delay, using mux components available in a technology library. Our primary objective is area minimization. Once the component muxes are determined, the next step in our algorithm is to minimize the number of signals used by the address inputs of the component muxes. For a wide range of muxes, we have obtained results that validate the effectiveness of the cost functions we have chosen for our search-based algorithm for generating area-efficient muxes. After obtaining the area-efficient implementation, we minimize the delay of that implementation using techniques described in this article.

Nomenclature

McCluskey defines a multiplexer as "a circuit that can select information from one of the several input terminals and route that input to a single output bit."[3] The mux has two sets of inputs, as shown in Figure 1a: data inputs (D_1, D_2, . . . , D_n) and address inputs (S_1, S_2, . . . , S_p), where $p = \lceil \log_2 n \rceil$. The binary code on the address inputs determines which data input is routed to the output. A *full (complete) mux* is one for which $n = 2^p$. Each data input of a full mux is selected by one and only one binary

0740-7475/00/$10.00 © 2000 IEEE

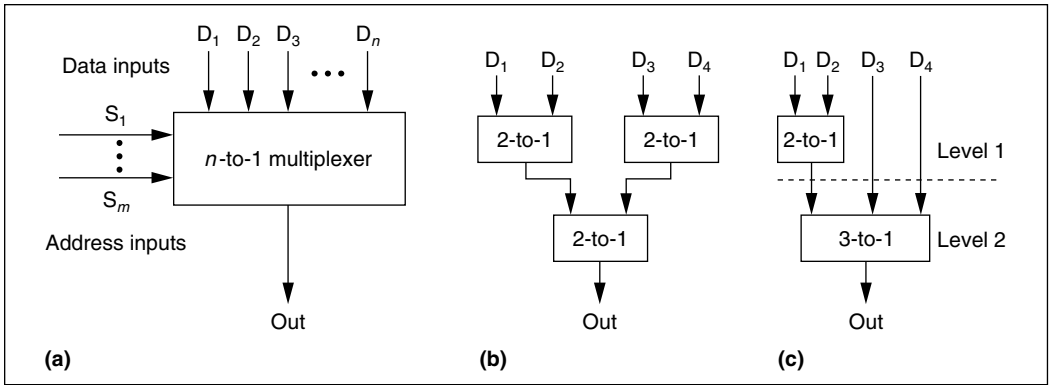

Figure 1. Multiplexers: (a) block diagram of a mux; (b) and (c) two possible implementations of a 4-to-1 mux from smaller muxes.

code on the address inputs. An *incomplete mux* is one for which $2^{p-1} < n < 2^p$. An n-to-1 mux can be implemented using a tree of smaller muxes. Figures 1b and 1c show two possible implementations of a 4-to-1 mux using component muxes. A mux that has all its input pins connected to primary inputs is said to be at *level 1*. The *level* of any other mux is defined as one plus the maximum level of any mux whose output is connected to the current mux's input.

Multiplexer area-minimization algorithm

Given a library of muxes, the problem considered here is to synthesize a larger mux function using a tree of mux components from the library such that the resulting mux's total area is kept to a minimum. Later we describe techniques for minimizing the delay of the minimum-area implementation. The information corresponding to each n-to-1 mux in the library that our algorithm requires can be represented by a pair <n, *area*>, where n is the number of data inputs. Table 1 shows two example mux library entries.

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

To reduce the exhaustive search complexity, we use a search technique based on heuristic cost functions. We start with a node representing n, the data input count of the mux to be implemented. Let's call this an OR node with label n (shown with two circles in Figure 2). 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 OR node's descendants AND nodes. There are $\lfloor n/2 \rfloor + 1$ AND descendants of an OR node with label n. An AND node's children, corresponding to the partition <$n1$, $n2$>, are OR nodes with labels $n1$ and $n2$. This nomenclature resembles that of AND-OR graphs.[4] The descendants of OR nodes represent various architectural choices. At a particular OR node, we are free to choose one of the AND nodes. However, once an AND node is chosen, we must build structures corresponding to both of its children. In our algorithm, we choose the AND node on the basis of cost estimates that represent the area of the mux's final implementation.

Figure 2 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 would not be an optimal solution.

Table 1. Examples of component multiplexers in technology libraries.

n	2	3	4	6	8
Library 1 area	8	14	19	33	42
Library 2 area	8	14	21	33	48

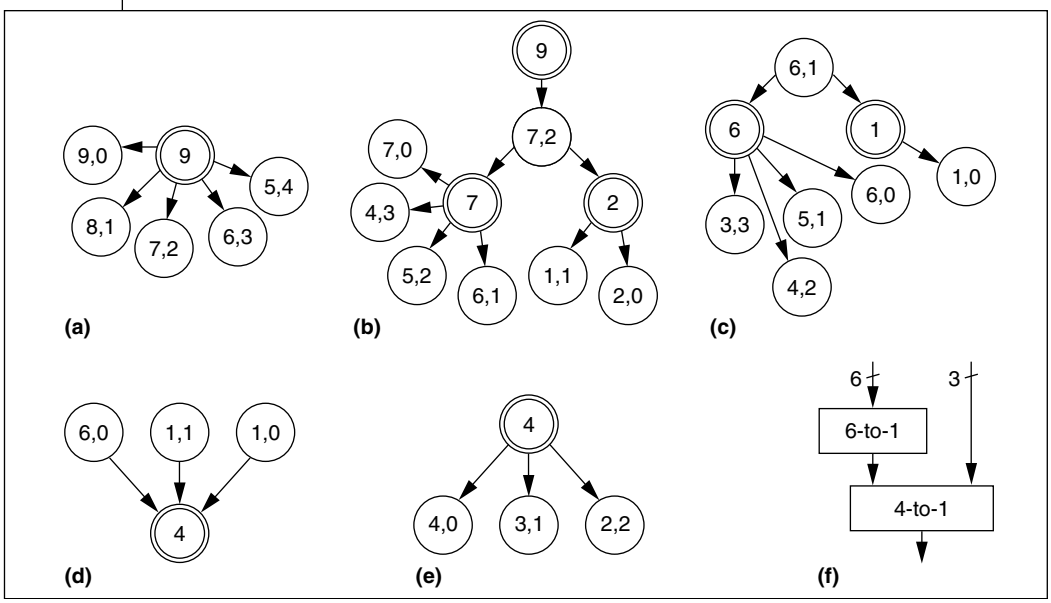

Figure 2. Synthesis of a 9-to-1 mux using library 2 (Table 1): (a) AND descendants of OR node 9; (b) descendants of OR nodes 7 and 2; (c) tree after choosing partition (8, 1); (d) output of 8-to-1 mux and three signal lines form OR node 4; (e) partitions of OR node 4; (f) final implementation.

Our algorithm works in the following way. We start with the OR node with label 9, as shown in Figure 2a. Of the five partitions (AND nodes), the partition (7, 2) is chosen, since it yields the lowest cost (from the cost functions described later in this section). The OR node with label 7 (Figure 2b) has four possible partitions, of which (6, 1) yields the lowest cost. For the OR node 2, we have two partitions, of which (1, 1) yields the lowest cost. The OR node with label 6 has four child AND nodes, of which (6, 0) corresponds to the lowest computed cost. Corresponding to (6, 0), we implement the 6-to-1 mux 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 exists 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 (Figure 2e). Cost computations tell us to implement a 4-to-1 mux corresponding to this node. Figure 2f shows the implementation. Figure 3 shows the pseudocode for the algorithm.

Next, we describe the method for calculating the estimated costs of the partitions. If we examine the area values of the different muxes 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 (14 units) is less than the area required if we implement the 3-to-1 mux using two 2-to-1 muxes in the library (16 units). This is the basic philosophy we use to generate the different cost functions.

The cost function that guides 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 muxes 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 muxes in an implementation of an $n1$-to-1 ($n2$-to-1) mux using a *best-fit strategy*. The best-fit strategy selects muxes 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 ($m = 4$), that is, 19. If $n1 = 12$, for library 1 (Table 1), the local cost is equal to the sum of areas of an 8-to-1 and a 4-to-1 mux, that is, 19 + 42 = 61 units.

The local cost gives us a local picture of a particular partition's cost. Next, we consider the global impact cost. This cost estimate serves to evaluate the effect of a particular choice of partition on the subsequent levels of the mux implementation. Suppose we choose ($n1$, $n2$) as the partition of n. Figure 4 shows the different components of the global impact cost if we

choose the AND node ($n1$, $n2$) during our search process. If we implement the first level of an $n1$-to-1 mux and an $n2$-to-1 mux using the best-fit strategy, then we can calculate the number of inputs (derived from the outputs of the muxes and stand-alone signal lines) ($n1$, $n2$) will contribute to the next level. *Let's call it* n*3*. 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 muxes 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), that is, 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 present in the next level if node m is implemented using the best-fit strategy. *Let this contribution be* m*3*. 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 the parent of this OR node is an AND node representing the partition (7, 4). Thus, node 7 has a sibling that 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 mux exists in the library). Therefore, the fact that $m3 = 1$ represents the output of the 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. Similarly, we can estimate the impact of choosing ($n1$, $n2$) on subsequent levels. Our algorithm has the flexibility to consider additional levels of the tree. The global impact cost plus the local cost equals the total estimated cost. We illustrate the cost calculation method using some examples. The algorithm for cost function computation has not been included here due to space constraints but is available from our technical report.[5]

For Figure 2b (library 2), we compute the AND node (2, 0) cost as follows:

```
Algorithm Synthesize Minimum Area Mux :

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

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

begin
    Determine_structure(n)
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 which 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_outputs = Determine_next_level (n1, n2) +
          Determine_next_level (n2, n1);
    }
    return (num_outputs)
end Determine_next_level;
```

Figure 3. Pseudocode for the area-minimization algorithm.

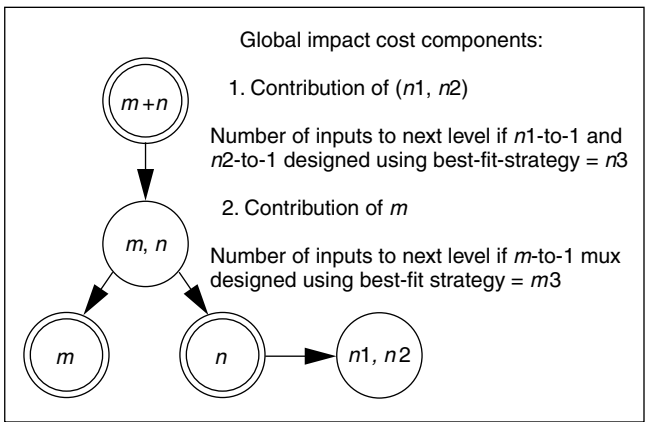

Figure 4. Global impact cost computation.

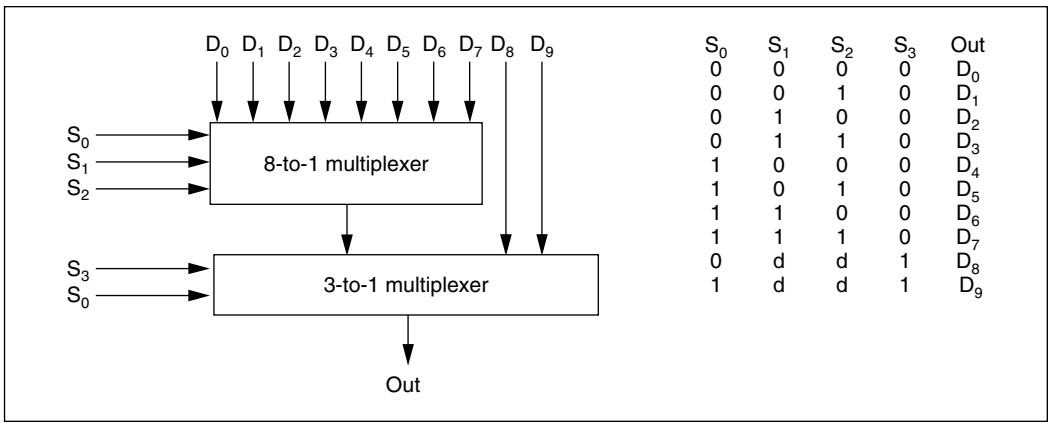

Figure 5. A 10-to-1 mux implementation.

- Local cost = area of the 2-to-1 mux = 8 units.
- 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 Figure 2b (library 2), the AND node (1, 1) cost is

- Local cost = 0 units.
- 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 mux = 21 units.
- Total estimated cost = 0 + 21 = 21 units.

Synthesis of multiplexer address signals

Next we consider the problem of generating address signals for the component muxes that implement the synthesized mux. The decomposition of a given mux with n data inputs into a tree of muxes of different sizes may require greater than $\lceil \log_2 n \rceil$ address signals, as we show later in this section. If we assume that the generated muxes have the minimum number of address inputs, that is, $\lceil \log_2 n \rceil$ address inputs, then extra logic may be required to translate the input address signals into the component address signals.

Figures 5 and 6 illustrate the synthesis of the component mux 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, doesn't require extra logic or extra signals for the address inputs because signal S_0 is assigned to the address input of both component muxes. However, implementation of an 11-to-1 mux with an 8-to-1 mux and a 4-to-1 mux requires extra logic (shown in Figure 6) if we require the number of address signals of our implemented mux to be minimum (4); otherwise, we require five address signals. We have developed an algorithm that, given a tree of mux components, tells us whether extra logic is necessary for the address signals. The algorithm, called Determine_Control, has been presented in our technical report.[5]

Different strategies may be adopted when Determine_Control reports the need for extra address signals for a given mux implementation. The simplest solution is to allocate extra address signals. Another strategy is to consider a set of minimal-cost decompositions instead of choosing only one at each step of the area-minimization algorithm. The third option is to add translation logic to generate the extra address signals required. The first approach, which transfers the responsibility for generating extra address signals to the control logic, is suitable for a high-level synthesis tool where address logic for multiple muxes may be shared. However, our experience shows that the second strategy—choosing an alternative implementation—produces good results.

Results

Now we compare the area results of the muxes generated by our technique with those of

muxes generated by commercial tools from Ambit (Cadence),[6] Synopsys,[7] and an ASIC vendor, referred to as tools A, B, and C in Table 2 (not in the same order). We ran these tools aiming at area optimization and used the highest effort option for technology mapping. For tool A, a commercial RTL synthesis tool, we wrote Verilog code with case statements to specify the mux operation. We specified an incomplete truth table (corresponding to the incomplete mux) using case statements with *full_case* annotation. This appears in the tool A1 column of Table 2. We also ran tool A with some special mux components instantiated. These results appear in the tool A2 column of Table 2. The LSI Logic G10-p library[8] served as the mapping library.

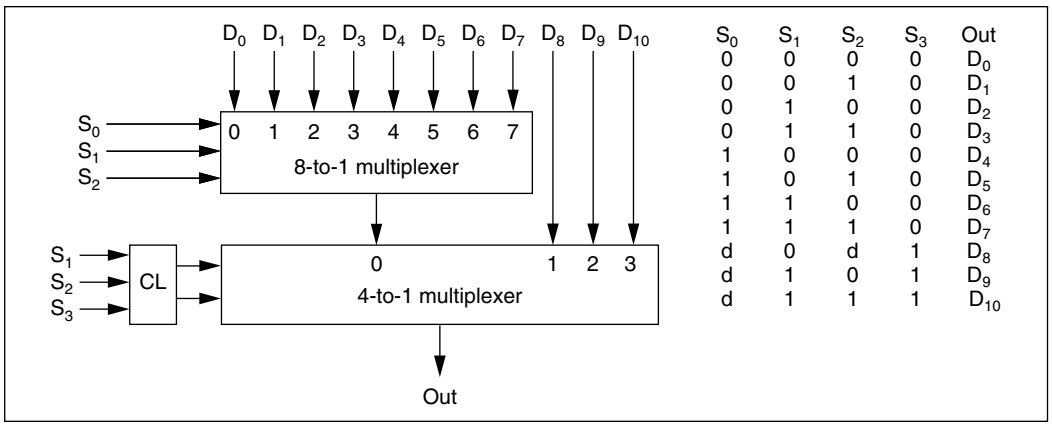

Figure 6. An 11-to-1 mux requiring extra address signal logic.

Tool B generates regular structures, such as muxes and counters, and it always generates muxes out of component muxes in the library. However, it cannot generate muxes with more than 16 data inputs. The version of tool B available to us cannot be used for the LSI Logic G10-p library but can be used for the LSI Logic LCB500K library.[9] (Tool B is an ASIC vendor's internal tool, and it was hard to get a version of this tool that will run for the LSI Logic G10p library.) The area result comparison for our algorithm and tool B appears in columns 5 and 6 of Table 2.

Tool C is another commercial RTL synthesis tool. However, our version of tool C supports the LCA300K library.[10] For tool C also, we used case statements written in Verilog as input. For incomplete muxes, we used *full_case* annotation. Columns 7 and 8 in Table 2 show the area comparisons.

As Table 2 shows, tool A1 always generates muxes with larger areas than those of muxes generated by our algorithm. A comparison with an exhaustive algorithm shows that our heuristic algorithm generates mux structures requiring minimum area. Tool A2 can generate the

Table 2. Area comparison: new algorithm versus tools A, B, and C. Boldface type indicates the minimum-area figures.

Number of	G10-p Library			LCB500K Library		LCA300K Library	
data inputs	New algorithm	Tool A1	Tool A2	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

Table 3. Comparison of mux delays (G10-p library). Boldface type indicates the minimum-delay figures (all times in nanoseconds).

No. of data Inputs	New algorithm	Tool A1	Tool A2
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

minimum decomposition only for the 16-to-1 case. The table also shows that tool B performs well, although our algorithm always performs as well as, or better than, tool B. However, tool B cannot generate muxes with more than 16 data inputs. Table 2 also shows that we consistently generate muxes requiring less area than those generated by tool C. We limit the comparison to 20 data inputs because muxes with more than 20 are typically implemented using tristate gates. The CPU time required to execute our algorithm is extremely small: less than a second to synthesize a 20-to-1 mux on a Sun Ultra-Sparc-2 workstation.

Delay-minimization algorithm

Next we minimize the delay of the muxes generated by our area-minimization algorithm. Unlike the work of Thakur et al.,[11] which considers decomposition of a given mux into a tree of 2-to-1 muxes, we consider the components determined by our area-minimization 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 mux component. The delay minimization algorithm is not reported in this article because of space constraints. We have presented the algorithm in our technical report.[5] Table 3 shows the delay values obtained by applying our delay-minimization algorithm to the minimal-area mux implementation, generated by our area-minimization algorithm for the G10-p library. The delay values have been computed by tool A from the structural Verilog descriptions of the muxes generated by our algorithm. Note that delay values reported in columns 3 and 4 of Table 3 correspond to the delays of the implementations whose area values appear in columns 3 and 4 of Table 2. The results in Tables 2 and 3 indicate that the mux implementations generated using our technique are efficient in terms of both area and delay.

ONE BIG ADVANTAGE of our technique is that our algorithm does not require a full truth table as input for synthesizing incomplete muxes. The procedure is well suited for use in high-level synthesis tools and has been implemented in TOPS, the Stanford Center for Reliable Computing's totally-optimized-synthesis-for-test tool.[12]

In this work, area minimization is our primary objective. However, an interesting extension 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 that implementation's delay. We are further investigating techniques for generating efficient muxes using mux components and some other basic gates (ANDs, ORs, NANDs, and so on) in the library. ■

Acknowledgments

We thank Jonathan T.Y. Chang (currently with Intel), Samy Makar (currently with Transmeta), and Robert B. Norwood (currently with John Brown University) of the Stanford Center for Reliable Computing for their helpful comments. This work was supported by the Advanced Research Projects Agency under contracts DABT-94-C-0045 and DABT63-97-C-0024.

■ References

1. E. Detjens et al., "Technology Mapping in MIS," *Proc. IEEE Int'l Conf. Computer Aided Design*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1987, pp. 116-119.
2. K. Keutzer, "Dagon: Technology Binding and Local Optimization by DAG Matching," *Proc. ACM/IEEE Design Automation Conf.*, ACM, New

York, 1987, pp. 617-623.
3. E.J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice Hall, Englewood Cliffs, N.J., 1986.
4. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, Calif., 1980.
5. S. Mitra, L.J. Avra, and E.J. McCluskey, *Efficient Multiplexer Synthesis*, Tech. Report CRC-TR-00-3, Center for Reliable Computing, Stanford Univ., 2000; http://crc.stanford.edu.
6. *BuildGates User Guide Release 2.0*, Ambit Design Systems, Santa Clara, Calif., Dec. 1997.
7. *DesignWare Components Quick Reference Guide, Version 1998.02*, Synopsys, Mountain View, Calif., Feb. 1998.
8. *G10-p Cell-Based ASIC Products Databook*, LSI Logic Corp., Milpitas, Calif., May 1996.
9. *LCB500K Preliminary Design Manual*, LSI Logic Corp., Milpitas, Calif., June 1995.
10. *LCA300K Gate Array 5 Volt Series Products Databook*, LSI Logic Corp., Milpitas, Calif., Oct. 1993.
11. S. Thakur, D.F. Wong, and S. Krishnamurthy, "Delay Minimal Decomposition of Multiplexers in Technology Mapping," *Proc. ACM/IEEE Design Automation Conf.*, ACM, New York, 1996, pp. 254-257.
12. Stanford Center for Reliable Computing; http://crc.stanford.edu/projects/topsSummary.html.

Subhasish Mitra is a research associate at Stanford University's Center for Reliable Computing. His research interests include digital testing, logic synthesis, and fault-tolerant computing. He received a BE in computer science and engineering from Jadavpur University, Calcutta, India, in 1994, an MTech in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1996, and a PhD in electrical engineering from Stanford University in 2000.

LaNae Avra is a development engineer with Ambit Design Systems (now part of Cadence Design Systems), working on high-level optimizations for the BuildGates synthesis tool. Her research interests include behavioral and RTL synthesis, design-for-test, and synthesis-for-test techniques. She received BS degrees from both Hope College in Holland, Michigan, and Rensselaer Polytechnic Institute in Troy, New York. She received an MS and a PhD in electrical engineering from Stanford University in 1990 and 1994, respectively.

Edward J. McCluskey is a professor of electrical engineering and computer science, as well as director of the Center for Reliable Computing. He developed the first algorithm for designing combinational circuits—the Quine-McCluskey logic minimization procedure—as a doctoral student at MIT. At Bell Labs and Princeton, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. McCluskey and his students at the Center for Reliable Computing worked out many key ideas for fault equivalence, probabilistic modeling of logic networks, pseudo-exhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry.

McCluskey served as the first president of the IEEE Computer Society. He is the recipient of many awards, including the IEEE Emanuel R. Piore Award. He is a Fellow of the IEEE, AAAS, and ACM, and a member of the National Academy of Engineering. He has published several books, including two widely used texts.

■ Readers may address comments and questions regarding this article to Subhasish Mitra, Center for Reliable Computing, Gates Bldg. 2A, Room 236, Stanford University, Stanford, CA 94305; smitra@crc.stanford.edu.