Symbolic Manipulation of Boolean Functions Using a Graphical Representation

Randal E. Bryant[1]

Dept. of Computer Science
Carnegie-Mellon University

Abstract

In this paper we describe a data structure for representing Boolean functions and an associated set of manipulation algorithms. Functions are represented by directed, acyclic graphs in a manner similar to the representations of Lee and Akers, but with further restrictions on the ordering of decision variables in the graph. Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation. Our algorithms are quite efficient as long as the graphs being operated on do not grow too large. We present performance measurements obtained while applying these algorithms to problems in logic design verification.

This paper was presented at the 22nd Design Automation Conference, 1985.

[1]This research was funded at the California Institute of Technology by the Defense Advanced Research Projects Agency ARPA Order Number 3771 and at Carnegie-Mellon University by the Defense Advanced Research Projects Agency ARPA Order Number 3597

1 Introduction

Many aspects of digital logic design would benefit from efficient computer algorithms for representing and manipulating Boolean functions. For example, if we could construct representations of the Boolean functions computed by a logic circuit (either combinational or sequential) and compare them to functions describing the desired behavior of the system, then we could truly *verify* the correctness of the circuit. Instead, most designers today rely on simulators for this task, only verifying the correctness for the particular data simulated. As another example, suppose we could construct representations of the functions computed by a circuit and by the circuit in the presence of some fault. Then by computing the exclusive-or of these two functions and finding some input pattern that yields a 1 for this new function, we would obtain a test for the fault. This technique was used before the advent of search-based test generation algorithms [1], and forms the basis of the Boolean difference method. [2]

Unfortunately, many of the operations one would like to perform with Boolean functions, such as testing whether there is any assignment of input variables for which a given Boolean expression evaluates to 1 (satisfiability), or whether two Boolean expressions denote the same function (equivalence) require solutions to NP-Complete or coNP-Complete problems [3]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. Our goal is to develop algorithms that achieve substantially better performance for a reasonably large class of Boolean functions, including those encountered in logic design applications.

In this paper we present a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. Our representation resembles the binary decision diagram notation introduced by Lee [4] and further popularized by Akers [5]. However, we place further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner. We will describe these algorithms and present performance results when applied to problems in logic design verification.

Our representation has several advantages over previous approaches to Boolean function manipulation. [6, 7] First, most commonly-encountered functions are represented by graphs of reasonable size. For example, the even and odd parity functions and the functions representing the output bits of an adder are represented by graphs where the number of vertices grow linearly with the number of arguments. In contrast, a sum-of-products representation of each of these functions grows exponentially with the number of arguments. Second, the performance of a program based on our algorithms when processing a sequence of operations degrades slowly, if at all. That is, the time complexity of any single operation is bounded by the product of the graph sizes for the functions being operated on. For example, complementing a function requires time proportional to the size of the function graph, while combining two functions with a binary operation (of which intersection, subtraction, and testing for implication are special cases) requires at most time proportional to the product of the two graph sizes. In contrast, attempting to complement a function or compute the exclusive-or of two functions can cause many other Boolean function manipulation programs to "blow up", either running out of storage or requiring an inordinate amount of computer time. Finally, our representation in terms of *reduced* graphs is a canonical form, i.e. every function has a unique representation. Hence, testing for equivalence simply involves testing whether the two graphs match exactly, while testing for satisfiability simply involves comparing the graph to that of the constant function **0**.

Unfortunately, our approach does have its own set of undesirable characteristics. Most significantly, at the start of processing we must choose some ordering of the system inputs as arguments to all of the functions to be represented. For some functions, the size of the graph representing the function is highly sensitive to this ordering. Our experience, however, has been that a human with some understanding of the problem domain can generally choose an appropriate ordering without great difficulty. Furthermore, some functions can be represented by Boolean expressions of reasonable length but the representation as a function graph is too large to be practical (i.e. more than 50,000 vertices.) Our experience has been that such functions arise in only one class of digital logic designs, namely multipliers.

2 Representation

We will assume the functions to be represented all have the same n arguments, written $x_1, \ldots, x_n$. In expressing a system such as a combinational logic network as a Boolean function, we must choose some ordering of the inputs and this ordering must be the same for all functions to be represented. The function which for all arguments yields the value 1 (respectively 0) is denoted **1** (resp. **0**).

A *function graph* is defined as a rooted, directed acyclic graph containing two types of vertices. A *nonterminal* vertex v has as attributes an argument index $index(v) \in \{1, \ldots, n\}$, and two child vertices $low(v)$,$high(v)$. A *terminal* vertex v has as attribute a value $value(v) \in \{0,1\}$. Furthermore, for any nonterminal vertex v, if $low(v)$ is also nonterminal, then we must have $index(v) < index(low(v))$. Similarly, if $high(v)$ is nonterminal, then we must have $index(v) < index(high(v))$. This ordering requirement differentiates our representation from conventional binary decision diagrams. A function graph having root vertex v denotes a Boolean function f_v defined recursively as:

1. If v is a terminal vertex:
   a. If $value(v)=1$, then f_v=**1**
   b. If $value(v)=0$, then f_v=**0**
2. If v is a nonterminal vertex with $index(v)=i$, then f_v is the function

$$f_v(x_1, \ldots, x_n) = \bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) + x_i \cdot f_{high(v)}(x_1, \ldots, x_n).$$

In other words, we can view a set of argument values $x_1, \ldots, x_n$ as describing a path in the graph starting from the root, where if some vertex v along the path has $index(v) = i$, then the path continues to the low child if $x_i = 0$ and to the high child if $x_i = 1$. The value of the function for these arguments equals the value of the terminal vertex at the end of the path.

A function graph can be reduced in size without changing the denoted function by eliminating redundant vertices and duplicate subgraphs. The resulting graph will be our primary data structure for representing a Boolean function. More formally, a function graph is said to be *reduced* if it contains no vertex v with $low(v)=high(v)$, nor does it contain distinct vertices v and v' such that the subgraph consisting of v and all of its descendants is isomorphic to the subgraph consisting of v' and all of its descendants. It can be shown that for any Boolean function, its representation as a reduced function graph is unique. Hence, our algorithm for reducing a function graph not only saves storage, it also produces a canonical representation.

3 Properties

<==<bool1.press<

Figure 1: Example Function Graphs

In this section we will explore the efficiency of our representation by means of several examples illustrated in Figure 1. In the figure, a nonterminal vertex is represented by a circle containing the index with the two children indicated by branches labeled 0 (low) and 1 (high). A terminal vertex is represented by a square containing the value.

The function which yields the value of the ith argument is represented by a graph with a single nonterminal vertex having index i and having as low child a terminal vertex with value 0 and as high child a terminal vertex with value 1. We present this graph mainly to point out that an input variable can be viewed as a Boolean function and hence can be operated on by the manipulation algorithms described in this paper.

The odd parity function of n variables is represented by a graph containing $2n+1$ vertices. This compares favorably to its representation in reduced sum-of-products form (requiring 2^n terms.) This graph resembles the familiar parity ladder contact network first described by Shannon [8]. In fact, we can adapt his construction of a contact network to implement an arbitrary symmetric function to show that any symmetric function of n arguments is represented by a reduced function graph having $O(n^2)$ vertices.

As a third example, the graph representing the function $x_1 \cdot x_2 + x_4$ contains 5 vertices as shown. This

example illustrates several key properties of reduced function graphs. First, observe that there is no vertex having index 3, because the function is independent of x_3. More generally, a reduced function graph will not contain any vertices with a given index unless the function depends on this variable. Second, observe that even for this simple function, several of the subgraphs are shared by different branches. This sharing yields efficiency not only in the size of the function representation, but also in the performance of our algorithms_ once some operation has been performed on a subgraph, the result can be utilized by all places sharing this subgraph.

<==<bool2.press<

Figure 2: Example of Argument Ordering Dependency

Figure 2 shows an extreme case of how the ordering of the arguments can affect the size of the graph representing a function. The functions $x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ and $x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ differ from each other only by a permutation of their arguments, yet one is represented by a function graph with 8 vertices while the other requires 16 vertices. Generalizing this to functions of $2n$ arguments, the function $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$ is represented by a graph of $2n+2$ vertices, while the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$ requires 2^{n+1} vertices. Consequently, a poor initial choice of input ordering can have very undesirable effects.

Upon closer examination of these two graphs, we can gain a better intuition of how this problem arises.

Imagine a bit-serial processor that computes a Boolean function by examining the arguments x_1, x_2, and so on in order, producing output 0 or 1 after the last bit has been read. Such a processor requires internal storage to remember enough about the arguments it has already seen to correctly compute the value of the function from the values of the remaining arguments. Some functions require little internal storage. For example, to compute parity a bit-serial processor need only store the parity of the arguments it has already seen. Similarly, to compute the function $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$, the processor need only remember whether any of the preceding pairs of arguments were both 1 as well as the value of the previous argument. On the other hand, to compute the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$, we would need to store the first n arguments to correctly deduce the value of the function from the remaining arguments. A function graph can be thought of as such a processor, with the set of vertices having index i describing the processing of argument x_i. Rather than storing intermediate information as bits in a memory, however, this information is encoded in the set of possible branch destinations. That is, if the bit-serial processor requires b bits to encode information about the first i arguments, then in any graph for this function there must be at least 2^b vertices that are either terminal or are nonterminal with index greater than i having incoming branches from vertices with index less than or equal to i. For example, the function $x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ requires 2^3 branches between vertices with index less than or equal to 3 to vertices which are either terminal or have index greater than 3. In fact, the first 3 levels of this graph must form a complete binary tree to obtain this degree of branching. In the generalization of this function, the first n levels of the graph form a complete binary tree, and hence the number of vertices grows exponentially with the number of arguments.

Unfortunately, the Boolean functions representing the output bits of an integer multiplier provide a case for which no ordering of the inputs produces a graph that is polynomial in the word size. Again, our bit-serial processor provides insight into this_ there is no implementation of a bit-serial multiplier that requires internal storage less than the word size.

4 Operations

We view a symbolic manipulation program as executing a sequence of commands that build up representations of functions and determine various properties about them. For example, suppose we wish to construct the representation of the function computed by a combinational logic gate network. Starting from graphs representing the input variables, we proceed through the network, constructing the function computed at the output of each logic gate by applying the gate operator to the functions at the gate inputs. A similar procedure is followed to construct the representation of the function denoted by some Boolean expression. At this point we can test various properties of the function, such as whether it equals **0** (satisfiability) or **1** (tautology), or whether it equals the function denoted by some other expression (equivalence). In this section we will describe the two most important algorithms for our Boolean function manipulation program.

4.1 Reduction

The reduction algorithm transforms an arbitrary function graph into a reduced graph denoting the same function. It closely follows an algorithm presented in Example 3.2 of Aho, Hopcroft, and Ullman [9] for testing whether two trees are isomorphic. Proceeding from the terminal vertices up to the root, a unique integer identifier is assigned to each unique subgraph root. That is, for each vertex v it assigns a label $id(v)$ such that for any two vertices u and v, $id(u) = id(v)$ if and only if $f_u = f_v$. Given this labeling, the

algorithm constructs a graph with one vertex for each unique label. By working from the terminal vertices up to the root, a procedure can label the vertices by the following inductive method. First, two terminal vertices should have the same label if and only if they have the same value attributes. Now assume all terminal vertices and all nonterminal vertices with index greater than i have been labeled. As we proceed with the labeling of vertices with index i, a vertex v should have $id(v)$ equal to that of some vertex that has already been labeled if and only if one of two conditions is satisfied. First, if $id(low(v)) = id(high(v))$, then vertex v is redundant, and we should set $id(v) = id(low(v))$. Second, if there is some labeled vertex u with $index(u) = i$ having $id(low(v)) = id(low(u))$, and $id(high(v)) = id(high(u))$, then the reduced subgraphs with these vertices as roots will be isomorphic, and we should set $id(v) = id(u)$.

The algorithm proceeds as follows. First, the vertices are collected into lists according to their indices. Then we process these lists working from the one containing the terminal vertices up to the one containing the root. For each vertex on a list we create a key of the form $(value)$ for a terminal vertex or of the form $(lowid, highid)$ for a nonterminal vertex, where $lowid = id(low(v))$ and $highid = id(high(v))$. If a vertex has $lowid = highid$, then we can immediately set $id(v) = lowid$. The remaining vertices are sorted according to their keys using a linear-time lexicographic sort. We then work through this sorted list, assigning a given label to all vertices having the same key. The reduced graph is constructed by creating a vertex for each unique label having as children the vertices corresponding to the labels in the key. The complexity of the algorithm is linear in the number of vertices.

<==<bool3.press<

Figure 3: Reduction Algorithm Example

Figure 3 shows an example of how the reduction algorithm works. Next to each vertex we show the key and the label generated during the labeling process. Observe that both vertices with index 3 have the same key, and hence the right hand vertex with index 2 is redundant.

4.2 Composition

The functional composition algorithm provides the basic method for creating the representation of a function according to the operators in a Boolean expression or logic gate network. It takes graphs representing functions f_1 and f_2, a binary operator <op> (i.e. any Boolean function of 2 arguments) and produces a reduced graph representing the function $f_1 <op> f_2$ defined as

$$[f_1 <op> f_2](x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) <op> f_2(x_1, \ldots, x_n).$$

This procedure can also be used to complement a function (compute $f \oplus \mathbf{1}$) to test for implication (compare $f_1 \cdot \neg f_2$ to $\mathbf{0}$), and a variety of other operations. With our representation, we can implement all of the operators with a single algorithm.

Before describing this algorithm, we must introduce some additional notation. The function resulting when some argument x_i to a function f is replaced by a constant b is called a *restriction* of f, denoted $f|_{x_i=b}$. That is, for any arguments $x_1, \ldots, x_n$,

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

The algorithm proceeds from the roots of the two argument graphs downward, creating vertices in the result graph at the branching points of the two arguments graphs. First, let us explain the basic idea of the algorithm. Then we will describe two refinements to improve the efficiency. The control structure of the algorithm is based on the following recursion:

$$f_1 <op> f_2 = \overline{x}_i \cdot (f_1|_{x_i=0} <op> f_2|_{x_i=0}) + x_i \cdot (f_1|_{x_i=1} <op> f_2|_{x_i=1})$$

To apply the operator to functions represented by graphs with roots v_1 and v_2, we must consider several cases. First, suppose both v_1 and v_2 are terminal vertices. Then the result graph consists of a terminal vertex having value $value(v_1) <op> value(v_2)$. Otherwise, suppose at least one of the two is a nonterminal vertex. If $index(v_1) = index(v_2) = i$, we create a vertex u having index i, and apply the algorithm recursively on $low(v_1)$ and $low(v_2)$ to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and $high(v_2)$ to generate the subgraph whose root becomes $high(u)$. Suppose, on the other hand, that $index(v_1) = i$, but either v_2 is a terminal vertex or $index(v_2) > i$, Then the function represented by the graph with root v_2 is independent of x_i, i.e.

$$f_2|_{x_i=0} = f_2|_{x_i=1} = f_2.$$

Hence we create a vertex u having index i, but recursively apply the algorithm on $low(v_1)$ and v_2 to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and v_2 to generate the subgraph whose root becomes $high(u)$. A similar situation holds when the roles of the two vertices in the previous case are reversed. In general the graph produced by this process will not be reduced. Hence we apply the reduction algorithm before returning the result.

If we were to implement the technique described in the previous paragraph directly we would obtain an algorithm of exponential (in n) time complexity, because every call for which one of the arguments is a nonterminal vertex generates two recursive calls. This complexity can be reduced by two refinements.

First, the algorithm need not evaluate a given pair of subgraphs more than once. Instead, we can maintain a table containing entries of the form (v_1, v_2, u) indicating that the result of applying the algorithm to subgraphs with roots v_1 and v_2 was a subgraph with root u. Then before applying the algorithm to a pair of vertices, we first check whether the table contains a corresponding entry. If so, we can immediately return the result. Otherwise, we proceed as described in the previous paragraph and add a new entry to the table. This refinement limits the complexity to the product of the two graph sizes,

showing that by exploiting the sharing of subgraphs in the data structures we gain efficiency in the algorithms. If the two graphs contain many shared subgraphs, we will obtain a high "hit rate" for our table. In practice we have found the hit rate to range between 40% and 50%.

Second, suppose the algorithm is applied to two vertices where one, say v_1, is a terminal vertex, and for this particular operator, $value(v_1)$ is a "controlling" value, i.e. either $value(v_1) <op> a = 1$ for all a, or $value(v_1) <op> a = 0$ for all a. For example, 1 is a controlling value for either argument of the operator $+$, while 0 is a controlling value for either argument of $\cdot$. In this case, there is no need to evaluate further. We simply create a terminal vertex having the appropriate value. While this refinement does not improve the worst case complexity of the algorithm, it certainly helps in many cases. In practice, we have found this case occurs around 10% of the time.

<==<bool4.press<

Figure 4: Example of Functional Composition

Figure 4 shows an example of how this algorithm would proceed in applying the "or" operation to graphs representing the functions $\neg(x_1 \cdot x_3)$ and $x_2 \cdot x_3$. This figure shows the graph created by the algorithm before reduction. Next to each vertex in the resulting graph, we indicate the two vertices on which the procedure was invoked in creating this vertex. Each of our two refinements is applied once: when the procedure is invoked on vertices a3 and b1 (because 1 is a controlling value for this operator), and on the second invocation on vertices a3 and b3. For larger graphs, we would expect these refinements to be applied more often. After the reduction algorithm has been applied, we see that the resulting graph indeed represents the function $\neg(x_1 \cdot \overline{x}_2 \cdot x_3)$.

5 Experimental Results

As with all other known algorithms for solving NP-hard problems, our algorithms have a worst-case performance that is unacceptable for all but the smallest problems. We hope that our approach will be practical for a reasonable class of applications, but this can only be demonstrated experimentally. We have already shown that the size of the graph representing a function can depend heavily on the ordering of the input variables, and that our algorithms are quite efficient as long as the functions are represented by graphs of reasonable size. Hence, the major questions to be answered by our experimental investigation are: how can an appropriate input ordering be chosen, and given a good ordering how large are the graphs encountered in typical applications.

We have implemented the algorithms described in this paper and have applied them to problems in logic design verification, test pattern generation, and combinatorics. On the whole, our experience has been quite favorable. By analyzing the problem domain, we can generally develop strategies for choosing a good ordering of the inputs. Furthermore, it is not necessary to find *the* optimal ordering. Many orderings will produce acceptable results. Functions rarely require graphs of size exponential in the number of inputs, as long as a reasonable ordering of the inputs has been chosen. Furthermore, the algorithms are quite fast, remaining practical for graphs with 20,000 or more vertices.

As an application, we consider the problem of verifying that the implementation of a logic function (in terms of a combinational logic gate network) satisfies its specification (in terms of Boolean expressions.) As examples we will use several different Arithmetic Logic Unit (ALU) designs constructed from 74181 and 74182 TTL integrated circuits [10]. The ’181 implements a 4 bit ALU slice, while the ’182 implements a lookahead carry generator. These chips can be combined to create an ALU with any word size that is a multiple of 4 bits. An ALU with an n bit word size has $6+2n$ inputs: 5 control inputs labeled m,s_0,s_1,s_2,s_3 to select the ALU function, a carry input labeled cin, and 2 data words of n bits each, labeled $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$. It produces $n+2$ outputs: n function outputs, a carry output, and a comparison output labeled $A=B$ (the logical "and" of the function outputs.)

n	**Bin. Ops.**	**Evals.**	**Secs.**	$A=B$
4	95	6405	19	197
8	194	22590	68	377
16	399	85641	239	737

Table 1: ALU Circuit Examples

For our experiments, we created logic gate networks by combining the gate-level descriptions of the chips according to the chip-level interconnections specified in the circuit manual. We then derived the functions at outputs of the networks and compared them to functions derived from Boolean expressions

obtained by encoding the functional specification in the circuit manual. We succeeded in verifying ALU's with word sizes of 4, 8, and 16 bits. The performance of our program in deriving the functions from the circuit descriptions is summarized Table 1. These data were measured with the best ordering we were able to find, as will be discussed shortly. The number of binary operations indicates the approximate complexity of each circuit, defined as the number of equivalent binary operations represented by the gate networks. The number of evaluations equals the total number of calls to the recursive routine in the composition algorithm. The CPU time is expressed in seconds as measured on a Digital Equipment Corporation VAX 11/780 (approximately 1 MIP machine.) The final column shows the number of vertices in the reduced graph for the $A{=}B$ output. In all cases, this was the largest graph generated.

As can be seen, the execution time of the individual procedures are quite short. Including the time used for memory management, for the user interface, and for reducing the graphs, the program required around 3 milliseconds per call to the recursive composition procedure. We can also see that the time required to derive the functions for these circuits grow approximately as the square of the word size. This is as good as can be expected: both the number of binary operations and the sizes of the graphs being operated on grow linearly with the word size, and the total execution time grows as the product of these two factors.

These ALU circuits also provide an interesting test case for evaluating different input orderings, because the successive bits of the function output word are functions of increasingly more variables. Figure 5 shows how the sizes of these graphs depend on the ordering of circuit inputs for 4 different cases. Case 1 is the best result obtained, corresponding to the ordering one would choose for a bit-serial implementation of the ALU: first the bits describing the function to be computed and then the successive bits of the two data words starting with the least significant bits. Case 2 provides the next best result, the same as before but with the data ordered with the most significant bit first. This ordering represents an alternative but often successful strategy: order the bits in decreasing order of importance. The ith bit of the output word depends more strongly on the ith bits of the input words than on any lower order bits. As can be seen, this strategy also works quite well. Case 3 represents an ordering with the control inputs last. This ordering could be expected to produce a rather poor result, since the outputs depend strongly on the control inputs. However, the complexity of the graphs still grows linearly, due to the fact that the number of control inputs is a constant. To explain this linear growth in terms of the bit-serial processor analogy, we could implement the ALU with the control inputs read last by computing all 32 possible ALU functions and then selecting the appropriate result once the desired function is known. Case 4 shows the effect a poor ordering, with all bits of 1 word preceding those of the other. This ordering requires the program to represent functions similar to the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$ considered in Section 3, with the same exponential growth characteristics.

6 Conclusion

These experimental results indicate that our representation works quite well for functions representing many arithmetic and logical operations on words of data, as long as we choose an ordering in which the successive bits of the input words are interleaved. Our representation seems especially efficient when compared to other representations of Boolean functions. For example, a truth table representation for output c_{15}, *cout*, or $A{=}B$ of the 16 bit ALU would require 2.7×10^{11} bits of storage, enough to fill 1500 reels of magnetic tape![2] A reduced sum-of-products representation of the most significant bit in the arithmetic

[2] 2400 foot reels, 6250 bits per inch

<==<bool5.press<

Input Orderings:

1: $m,s_0,s_1,s_2,s_3,cin,a_0,b_0 \ldots ,a_{n-1}b_{n-1}$

2: $m,s_0,s_1,s_2,s_3,cin,a_{n-1},b_{n-1}, \ldots ,a_0b_0$

3: $cin,a_0,b_0, \ldots ,a_{n-1}b_{n-1},m,s_0,s_1,s_2,s_3$

4: $m,s_0,s_1,s_2,s_3,cin,a_0, \ldots ,a_{n-1}b_0, \ldots ,b_{n-1}$

Figure 5: ALU Output Graph Sizes for Different Input Orderings

sum of two n bit numbers requires about 2^{n+2} product terms, and hence a reduced sum-of-products representation of this circuit would be equally impractical. We believe our approach will be practical for representing a large class of logic designs. With the exception of integer multiplication, we have encountered no functions that require the exponential size graphs that a worst case analysis would predict, given an appropriate choice of input ordering.

References

1. J.P. Roth, ‘‘Diagnosis of Automata Failures’’, Tech. report SR-114, IBM Thomas Watson Research Center, 1960.

2. F.F. Sellers, Jr., M.Y. Hsiao, and L.W. Bearnson, ‘‘Analyzing Errors with the Boolean Difference’’, *IEEE Transactions on Computers*, Vol. C-17, No. 7, July 1968, pp. 676-683.

3. M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of* NP-Completeness, Freeman, New York, 1979.

4. C.Y. Lee, ‘‘Representation of Switching Circuits by Binary-Decision Programs’’, *Bell System Technical Journal*, Vol. 38, July 1959, pp. 985-999.

5. S.B. Akers, ‘‘Binary Decision Diagrams’’, *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 509-516.

6. J.P. Roth, *Computer Logic, Testing, and Verification,* Computer Science Press, Potomac, MD., 1980.

7. R. Brayton, *et al*, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, 1984.

8. C.E. Shannon, ‘‘A Symbolic Analysis of Relay and Switching Circuits’’, *Transactions of the AIEE*, Vol. 57, 1938, pp. 713-723.

9. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA., 1974.

10. Texas Instruments, *TTL Data Book,* 1976.

Table of Contents

List of Figures

List of Tables