Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Verification of Arithmetic Circuits Using Binary Moment Diagrams *

Randal E. Bryant, Yirng-An Chen

Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
Email: bryant@cs.cmu.edu
Dept. of Computer & Information Science, National Chiao Tung University, Hsinchu, Taiwan 300, R.O.C.
Email: yachen@cis.nctu.edu.tw

Received: date / Revised version: date

Abstract. Binary Moment Diagrams (BMDs) provide a canonical representations for linear functions similar to the way Binary Decision Diagrams (BDDs) represent Boolean functions. Within the class of linear functions, we can embed arbitrary functions from Boolean variables to real, rational, or integer values. BMDs can thus model the functionality of data path circuits operating over word level data. Many important functions, including integer multiplication, that cannot be represented efficiently at the bit level with BDDs have simple representations at the word level with BMDs. Furthermore, BMDs can represent Boolean functions with around the same complexity as BDDs.

We propose a hierarchical approach to verifying arithmetic circuits, where component modules are first shown to implement their word-level specifications. The overall circuit functionality is then verified by composing the component functions and comparing the result to the word-level circuit specification. Multipliers with word sizes of up to 256 bits have been verified by this technique.

1 Introduction

Binary Decision Diagrams (BDDs) have proved successful for representing and manipulating Boolean functions symbolically [4] in a variety of application domains. Building on this success, there have been several efforts to extend the BDD concept to represent functions over Boolean variables, but having non-Boolean ranges, such as integers or real numbers [1,8,13,20,22,21]. This class of functions is sometimes termed "pseudo-Boolean" [17]. Many tasks can be expressed in terms of operations on such functions, including integer linear programming, matrix manipulation, spectral transforms, and word-level digital system analysis. To date, the proposed representations for these functions have proved too fragile for routine application—too often the data structures grow exponentially in the number of variables.

In this paper we propose a new representation called Multiplicative Binary Moment Diagrams (*BMDs) that improve on previous methods. *BMDs incorporate two novel features: they are based on a decomposition of a linear function in terms of its "moments," and they have weights associated with their edges which are combined multiplicatively. These features have as heritage ideas found in previous function representations, namely the Reed-Muller decomposition used by Functional Decision Diagrams (FDDs) [11,19], and the additive edge weights found in Edge-Valued Binary Decision Diagrams (EVBDDs) [20,21]. The relations between the various representations are described more fully below.

*BMDs are particularly effective for representing digital systems at the word level, where sets of binary signals are interpreted as encoding integer (fixed point) or rational (floating point) values. Common integer and floating point encodings have efficient representations as *BMDs, as do operations such as addition and multiplication. *BMDs can also represent Boolean functions as a special case, with size comparable to BDDs.

*BMDs can serve as the basis for a hierarchical methodology for verifying circuits such as multipliers. At the low level, we have a set of building blocks such as add steppers, Booth steppers, and carry save adders described at both the bit level (as combinational circuits) and at the word level (as algebraic expressions). Using a methodology proposed by Lai and Sastry [20], we verify that the bit-level implementation of each block implements its word-level specification. At the higher level (or lev-

* This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330.

	Boolean	**Numeric**	
		Terminal	**Edge-Weighted**
Point-wise	BDD	MTBDD, ADD	EVBDD
Moment	FDD	BMD	*BMD

Table 1. Categorization of Graphical Function Representations.

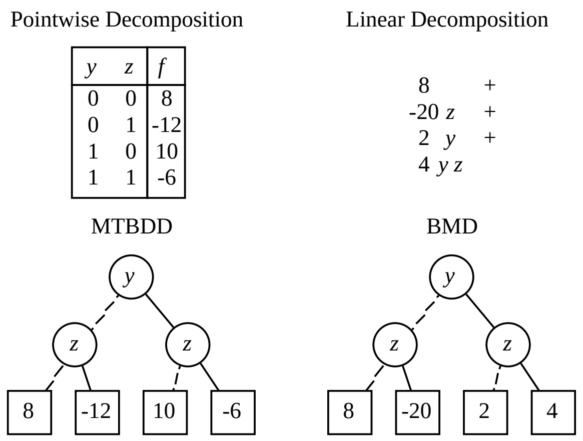

Fig. 1. Example Function Decompositions. MTBDDs are based on a point-wise decomposition (left), while BMDs are based on a linear decomposition (right).

els), a system is described as an interconnection of blocks having word-level representations, and the specification is also given at the word-level. We then verify that the composition of the block functions corresponds to the system specification. By this technique we can verify systems, such as multipliers [5], that cannot be represented efficiently at the bit level. We also can handle a more abstract level of specification than can methodologies that work entirely at the bit level.

2 Graphical Function Representations

Methods related to ordered BDDs for representing functions as graphs can be categorized as shown in Table 1. First, the range of a function can be either Boolean or numeric, e.g., integer, rational, or real. Second, we will consider two methods of decomposing a function with respect to a Boolean variable x: in terms of its value at $x = 1$ and $x = 0$ (point-wise decomposition), or its "moments," i.e., its value at $x = 0$ and how this value changes as x changes to 1. Finally, the values of a numeric function can be expressed in terms of values associated with the leaves or with the edges. Note that in all cases we assume a total ordering of the variables and that variables are tested according to this ordering along any path from the root to a leaf.

To illustrate the two ways of decomposing a function, consider the function F over a set of Boolean variables y and z, yielding the integer values shown in the table of Figure 1. A point-wise decomposition characterizes a function by its value for every possible set of argument values. By extending BDDs to allow numeric leaf values, the point-wise decomposition leads to a "Multi-Terminal" BDD (MTBDD) representation of a function [8,13] (also called "ADD" [1]), as shown on the left side of Figure 1. In our drawings of graphs based on a point-wise decomposition, the dashed line from a vertex denotes the case where the vertex variable is 0, and the solid line denotes the case where the variable is 1. Observe that the leaf values correspond directly to the entries in the function table.

Exploiting the fact that the function variables take on only the values 0 and 1, we can write a linear expression for function F directly from the function table. For variable y, the assignment $y = 1$ is encoded as y, and the assignment $y = 0$ is encoded as $1 - y$:

$$F(y,z) = \begin{array}{rccl} 8 & (1-y) & (1-z) & + \\ -12 & (1-y) & z & + \\ 10 & y & (1-z) & + \\ -6 & y & z & \end{array}$$

Expanding this expression and combining common terms yields the expression:

$$\begin{aligned} F(y,z) &= 8 - 20z + 2y + 4yz \\ &= 8y^0z^0 + -20y^0z^1 + 2y^1z^0 + 4y^1z^1 \end{aligned}$$

This representation is called the "monomial expansion" of F. It represents the function as a sum of terms $\alpha y^{b_y} z^{b_z}$ where α is a numeric coefficient and both b_y and b_z are either 0 or 1. This expansion leads to the BMD representation of a function, as shown on the right side of Figure 1. In our drawings of graphs based on a moment decomposition, the dashed line from a vertex indicates the case where the function is independent of the vertex variable x ($b_x = 0$), while the solid line indicates the case where the function varies linearly ($b_x = 1$).

2.1 Recursive Decompositions of Functions

The graph representations of functions we consider expand a function one variable at a time, rather than in terms of all the variables, as do the tabular form and the monomial expansions of Figure 1. Better insight can be gained by considering recursive decompositions of the function, where a function is decomposed in terms of a variable into two subfunctions. In our graphical representation, each vertex denotes a function. The outgoing branches from the vertex indicate the subfunctions resulting from the decomposition with respect to the vertex variable.

For function f over a set of Boolean variables, let f_x (respectively, $f_{\overline{x}}$) denote the positive (resp., negative) *cofactor* of f with respect to variable x, i.e., the function resulting when constant 1, (resp., 0) is substituted for x. BDDs are based on a point-wise decomposition, where the function is characterized with respect to some variable x in terms of its cofactors. Function f can be expressed in terms of an expansion (variously credited to Shannon and to Boole):

$$f \;=\; \overline{x} \wedge f_{\overline{x}} \;\vee\; x \wedge f_x$$

In this equation we use $\wedge$ and $\vee$ to represent Boolean sum and product, and overline to represent Boolean complement.

For expressing functions having numeric range, the Boole-Shannon expansion can be generalized as:

$$f = (1 - x) \cdot f_{\overline{x}} \;+\; x \cdot f_x \tag{1}$$

where $\cdot$, $+$, and $-$ denote multiplication, addition, and subtraction, respectively. Note that this expansion relies on the assumption that variable x is Boolean, i.e., it will evaluate to either 0 or 1. Both MTBDDs and EVBDDs [20, 22] are based on such a point-wise decomposition. As with BDDs, each vertex v describes a function f in terms of its decomposition with respect to variable $x = \mathsf{Var}(v)$. The two outgoing arcs: $\mathsf{Lo}(v)$ and $\mathsf{Hi}(v)$ denote functions $f_{\overline{x}}$ and f_x, respectively. A leaf vertex v in an MTBDD has an associated value $\mathsf{Val}(v)$.

The moment decomposition of a function is obtained by rearranging the terms of Equation 1:

$$\begin{aligned} f &= f_{\overline{x}} \;+\; x \cdot (f_x - f_{\overline{x}}) \\ &= f_{\overline{x}} \;+\; x \cdot f_{\delta x} \end{aligned} \tag{2}$$

where $f_{\delta x} = f_x - f_{\overline{x}}$ is called the *linear moment* of f with respect to x. This terminology arises by viewing f as being a linear function with respect to its variables, and thus $f_{\delta x}$ is the partial derivative of f with respect to x. Since we are interested in the value of the function for only two values of x, we can always extend it to a linear form. The negative cofactor will be termed the *constant moment*, i.e., it denotes the portion of function f that remains constant with respect to x, while $f_{\delta x}$ denotes the portion that varies linearly. Relating to the monomial expansion presented earlier, the two moments of function f partition the set of monomial terms into those that are independent of x, i.e., $b_x = 0$ ($f_{\overline{x}}$), and those that vary linearly with x, i.e., $b_x = 1$ ($f_{\delta x}$).

We will define two forms of graphs representing functions according to a moment decomposition. In both cases, vertex v denoting function f is labeled by a variable $x = \mathsf{Var}(v)$, and has two outgoing arcs: $\mathsf{Lo}(v)$ denoting function $f_{\overline{x}}$ and $\mathsf{Hi}(v)$ denoting function $f_{\delta x}$. We will term graphs of this form "Moment" Diagrams (MDs) as opposed to "Decision" Diagrams (DDs). The distinction is based on the rules used to evaluate a function for some valuation of the variables. In a decision diagram one simply traverses the unique path from the root to a leaf determined by the variable values, possibly accumulating edge weights. For example, consider the evaluation of a MTBDD for Boolean variable assignment ϕ. That is, ϕ denotes a function that for each variable x assigns

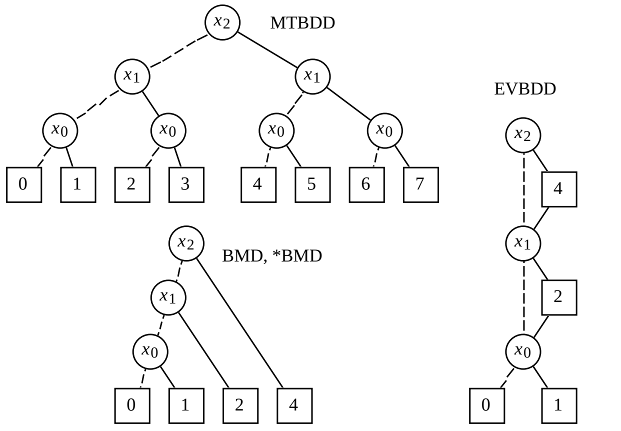

Fig. 2. Different Representations for Binary-Weighted Bits. All represent the function $X = 4x_2 + 2x_1 + x_0$.

a value $\phi(x)$ equal to either 0 or to 1. The evaluation starting at vertex v can be defined as:

$$MTBDDeval(v, \phi) = \begin{cases} \mathsf{Val}(v), & v \text{ is leaf} \\ MTBDDeval(\mathsf{Lo}(v), \phi), & \phi(\mathsf{Var}(v)) = 0 \\ MTBDDeval(\mathsf{Hi}(v), \phi), & \phi(\mathsf{Var}(v)) = 1 \end{cases} \tag{3}$$

In a moment diagram, evaluation requires consideration of multiple paths in the graph. For every vertex v labeled by a variable x that evaluates to 1, subgraphs $\mathsf{Lo}(v)$ and $\mathsf{Hi}(v)$ must both be evaluated and their results summed. The evaluation of BMD for Boolean variable assignment ϕ starting at vertex v can be defined as:

$$BMDEval(v, \phi) = \begin{cases} \mathsf{Val}(v) & v \text{ is leaf} \\ BMDEval(\mathsf{Lo}(v), \phi) & \phi(\mathsf{Var}(v)) = 0 \\ BMDEval(\mathsf{Lo}(v), \phi) + BMDEval(\mathsf{Hi}(v), \phi) & \phi(\mathsf{Var}(v)) = 1 \end{cases} \tag{4}$$

In return for the more complex evaluation rule of moment diagrams, we obtain graphs that are potentially much more compact.

By way of comparison, the moment decomposition of Equation 2 is analogous to the Reed-Muller expansion (also called the positive Davio expansion [11]) for Boolean functions:

$$f = f_{\overline{x}} \;\oplus\; x \wedge (f_x \oplus f_{\overline{x}})$$

The expression $f_x \oplus f_{\overline{x}}$ is referred to as the *Boolean difference* of f with respect to x [25], and in many ways is analogous to our linear moment. Other researchers [11, 19] have explored the use of graphs for Boolean functions based on this expansion, calling them Functional Decision Diagrams (FDDs). By our terminology, we would refer to such a graph as a "moment" diagram rather than a "decision" diagram.

2.2 Edge Versus Terminal Weights

One method to represent functions yielding numeric values, used by MTBDDs and by BMDs, is to simply introduce a distinct leaf vertex for each constant value needed. This approach has the drawback, however, that many leaves may be required, often exponential in the number of variables. Figure 2 illustrates the complexity of the function mapping a vector of Boolean variables: $x_{n-1}, \ldots, x_1, x_0$ to an integer value according to its interpretation as an unsigned binary number. As can be seen, the MTBDD representation will grow exponentially with the word size, since there are 2^n different values for the function.

A second method for defining function values is to associate weights with the edges. This idea was originated by Lai, *et al* in their definition of EVBDDs. In their case, edge weights are combined additively, i.e., the value of a function is determined by following a path from a root to a leaf, summing the edge weights encountered. As shown on the right side of Figure 2, the edge weights of EVBDDS can lead to a much more compact representation than with MTBDDs. In our drawings of EVBDDs, edge weights are shown in square boxes, where an edge without a box has weight 0. For representing a sum of weighted bits, this representation achieves a linear complexity. Various schemes can be used for "normalizing" edge weights so that the resulting graph provides a canonical form for the function. For example, the standard formulation of EVBDDs requires that edge $\mathsf{Lo}(v)$ for any vertex v have weight 0.

The bottom of Figure 2 shows the BMD representation of the same function. Observe that the graph for this function grows linearly with word size. In our drawings for BMDs, the solid line leaving vertex v indicates $\mathsf{Hi}(v)$, the linear moment. The linear moment of X with respect to any variable x_i is simply its binary weight 2^i, giving rise to the simple linear structure shown. Thus, the moment decomposition is sufficient for simplifying the representation of this function.

*BMDs also have edge weights, although the weights combine multiplicatively rather than additively. Although not the case for Figure 3, edge weighting can lead to a much more concise representation of a function. As an illustration, Figure 3 shows three representations of the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. The upper graph is a BMD, with the leaf values corresponding to the coefficients in the monomial expansion. As the figure shows, the BMD data structure misses some opportunities for sharing of common subexpressions. For example, the terms $2y + 4yz$ and $12x + 24xz$ can be factored as $2y(1 + 2z)$ and $12x(1 + 2z)$, respectively. The representation could therefore save space by sharing the subexpression $1 + 2z$. For more complex functions, one might expect more opportunities for such sharing.

The two forms of *BMDs, shown at the bottom of Figure 3 indicate how *BMDs are able to exploit the sharing of common subexpressions. In our drawings of *BMDs, we indicate the weight of an edge in a square box. Unlabeled edges have weight 1. In evaluating the function for a set of arguments, the weights are multiplied together when traversing downward. There are a

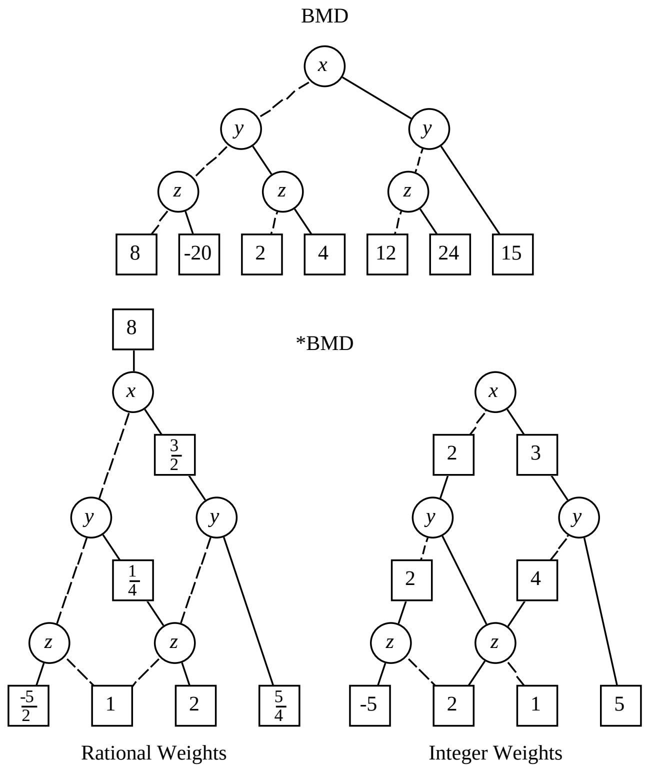

Fig. 3. Examples of BMD and *BMDs. All represent the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. *BMDs have weights on the edges that combine multiplicatively.

variety of different rules for manipulating edge weights, resulting in different representations. We will describe two different sets of rules—one that results in rational weights, even when manipulating integer functions (left), and one that yields integer weights, but is only applicable for integer functions (right). Observe that these two rules yield graphs with identical branching structure, but differing in edge weights.

For the remainder of the presentation we will consider mainly *BMDs, The effort required to implement weighted edges is justified by the savings in graph sizes. For functions with integer ranges, we will use integer edge weights. Keeping edge weights as integers is easier than maintaining rational numbers. If we approximate rational numbers with floating point representations, the vagaries of the rounding behavior could greatly complicate the use of *BMDs in formal verification.

2.3 Algebraic Structure

Although we have presented BMDs and *BMDs as methods for representing functions over Boolean variables, they can also be viewed as representing arbitrary linear functions. For example, the BMD of Figure 1 can be viewed as representing the function $F(x, y) = 8 - 20z + 2y + 4yz$ for arbitrary values of y and z. The rule for evaluating a graph given a numeric variable assignment

ϕ then becomes:

$$LinEval(v, \phi) = \begin{cases} \mathsf{Val}(v), & v \text{ is leaf} \\ LinEval(\mathsf{Lo}(v), \phi) + \phi(\mathsf{Var}(v)) \cdot LinEval(\mathsf{Hi}(v), \phi) & \text{otherwise} \end{cases} \quad (5)$$

The class of linear functions can be defined as either those that can be expressed as a sum of monomial terms, or as those functions that obey Equation 2 for all variables.

An algebraic structure for linear functions provides further insight into our representation. Let L denote the set of linear functions, and for a variable assignment ϕ let $f(\phi)$ denote the result of evaluating linear function f according to this assignment. We can define addition of linear functions in the usual way, i.e., the sum of two functions $f + g$ is a function h such that $h(\phi) = f(\phi) + g(\phi)$. It can be seen that the algebraic structure $\langle L, + \rangle$ forms a group, having as identity element the function that always evaluates to 0.

We could define multiplication over functions in a similar fashion, but then the class of linear functions would not be closed under this operation. The product of two linear functions could yield a quadratic function. In particular, the product of functions f and g, denoted $f \cdot g$ can be defined recursively as follows. If these functions evaluate to constants a and b, respectively, then their product is simply $f \cdot g = a \cdot b$. Otherwise assume the functions are given by their moment expansions (Equation 2) with respect to some variable x. The product of the functions can then be defined as:

$$f \cdot g \;= f_{\overline{x}} \cdot g_{\overline{x}} + x(f_{\overline{x}} \cdot g_{\delta x} + f_{\delta x} \cdot g_{\overline{x}}) + x^2 f_{\delta x} \cdot g_{\delta x} \quad (6)$$

One can readily show that this definition is unambiguous—the result is independent of the ordering of the variables in the successive decompositions.

Instead of conventional multiplication, we can define an operation $\hat{\cdot}$ with similar properties, except that it preserves linearity. This involves "demoting" the quadratic term in the equation for conventional multiplication to a linear term. The *linear product* of functions f and g, denoted $f \,\hat{\cdot}\, g$, is defined recursively as follows. If these functions evaluate to constants a and b, respectively, then their linear product is simply their product: $f \,\hat{\cdot}\, g = a \cdot b$. Otherwise assume the functions are given by their moment expansions (Equation 2) with respect to some variable x. Their linear product is defined as

$$f \,\hat{\cdot}\, g \;= f_{\overline{x}} \,\hat{\cdot}\, g_{\overline{x}} + x(f_{\overline{x}} \,\hat{\cdot}\, g_{\delta x} + f_{\delta x} \,\hat{\cdot}\, g_{\overline{x}} + f_{\delta x} \,\hat{\cdot}\, g_{\delta x}) \quad (7)$$

One can show that the definition is independent of the ordering in the decomposition. The algebraic structure $\langle L, +, \hat{\cdot} \rangle$ forms a ring. That is, $\hat{\cdot}$ is associative, and it distributes over $+$. Furthermore, the function that always yields 1 serves as a unit for this ring.

Although the linear product operation is not the same as conventional multiplication, there are two important cases where we can safely use $f \,\hat{\cdot}\, g$ as a replacement for $f \cdot g$. First, under the *Boolean domain restriction*, i.e., considering only variable assignments ϕ such that $\phi(x) \in \{0, 1\}$, we are guaranteed that $[f \cdot g](\phi) = [f \,\hat{\cdot}\, g](\phi)$. Second, define the *support* of a function f as those variables x such that $f_{\delta x} \neq 0$. Under the *independent support assumption*, where functions f and g have disjoint support sets, we have that $f \cdot g = f \,\hat{\cdot}\, g$ for any variable assignment. In particular, for any variable x we must have that either $f_{\delta x}$ or $g_{\delta x}$ is identically 0, and hence the quadratic term of Equation 6 drops out.

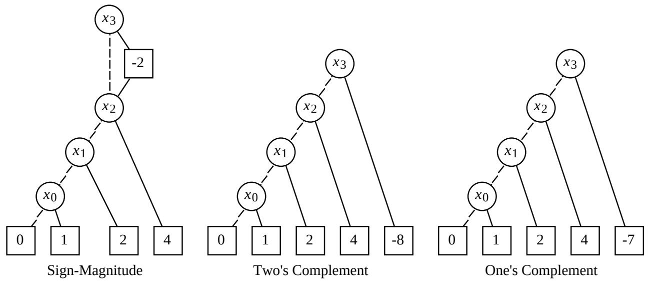

Fig. 4. Representations of Signed Integers. All commonly used encodings can be represented easily.

In general, we can "linearize" any operation *op* to create an operation $\hat{op}$ such that for any Boolean variable assignment ϕ, we have $[f \,\hat{op}\, g](\phi) = f(\phi) \; op \; g(\phi)$. This involves generating moments with respect to each variable x as:

$$\begin{aligned} [f \,\hat{op}\, g]_{\overline{x}} &= f_{\overline{x}} \,\hat{op}\, g_{\overline{x}} & (8) \\ [f \,\hat{op}\, g]_{\delta x} &= [f \,\hat{op}\, g]_{x} - [f \,\hat{op}\, g]_{\overline{x}} & \\ &= [f_x \,\hat{op}\, g_x] - [f_{\overline{x}} \,\hat{op}\, g_{\overline{x}}] & \\ &= [(f_{\overline{x}} + f_{\delta x}) \,\hat{op} (g_{\overline{x}} + g_{\delta x})] - [f_{\overline{x}} \,\hat{op}\, g_{\overline{x}}] & (9) \end{aligned}$$

As before, the definition is independent of the variable ordering. In general, this linearization would not yield valid results for non-Boolean variable assignments, whether or not the arguments have independent support. For example, the linearized form of exponentiation would convert $(x + 2)^y$ into $1 + y + xy$.

3 Representation of Numeric Functions

*BMDs provide a concise representation of functions defined over "words" of data, i.e., vectors of bits having a numeric interpretation. Let $\mathbf{x}$ represent a vector of Boolean variables: $x_{n-1}, \ldots, x_1, x_0$. These variables can be considered to encode an integer X according to some encoding, e.g., unsigned binary, two's complement, BCD, etc. As Figure 2 shows, the *BMD (as well as BMD) representations for X according to an unsigned binary encoding have linear complexity. Figure 4 illustrates the *BMD representations of several common encodings for signed integers, where x_{n-1} is the sign bit. The sign-magnitude encoding gives integer value $X = -1^{x_{n-1}} X'$,

Form	$X+Y$	$X*Y$	X^2	c^X
MTBDD	exponential	exp.	exp.	exp.
EVBDD	linear	exp.	exp.	exp.
BMD	linear	quadratic	quadratic	exp.
*BMD	linear	linear	quadratic	linear

Table 2. Word-Level Operation Complexity. Expressed in how the graph sizes grow relative to the word size.

where X' is the unsigned integer encoded by the remaining bits. Observe that this can be expressed in the linear form $(1 - 2x_{n-1})X'$, yielding a graph structure where both moments for variable x_{n-1} point to the graph for X', but having edge weights 1 and -2. As the other graphs in the figure illustrate, both two's complement and one's complement encodings can be viewed as sums of weighted bits, where the sign bit is weighted either -2^{n-1} (two's complement) or $1 - 2^{n-1}$ (one's complement) [23].

The conciseness of *BMDs arises from two important properties of typical encodings. First, many encodings are based on a sum of weighted bits. In terms of the monomial expansion, this implies that the terms are all of low degree. Second, the regularity of the encodings gives rise to many subexpressions differing only by multiplicative factors. This leads to sharing of subgraphs in the *BMD, with edge weights denoting the different factors.

3.1 Word-Level Operations

Table 2 provides a comparative summary of the four function representations for a number of word-level operations on unsigned data. *BMD examples of these functions are included in this paper. As can be seen, MTBDDs are totally unsuited for this class of functions. The range of the functions to be represented is simply too large. EVBDDs yield better results for representing word-level data and for representing "additive" operations (e.g, addition and subtraction) at the word level. This capability was exploited by Lai and Sastry in verifying adder circuits against word-level specifications [20]. On the other hand, EVBDDs cannot efficiently represent more complex functions such as multiplication, squaring, and exponentiation. Thus, for example, they cannot be used for verifying multipliers. In fact, all published examples that can be handled efficiently at the word level using EVBDDs can be handled at the bit level using BDDs. Their utility in verifying circuits is mainly for providing a more abstract form of specification.

Both BMDs and *BMDs are much more effective for representing word-level operations. BMDs remain of polynomial (quadratic) size for both multiplication and for squaring, although they grow exponentially for exponentiation. *BMDs do even better, being quadratic for squaring and linear for all other operations listed. By verifying circuits at the word level with *BMDs, we can handle classes of systems that are beyond the capability of BDDs and other bit-level techniques.

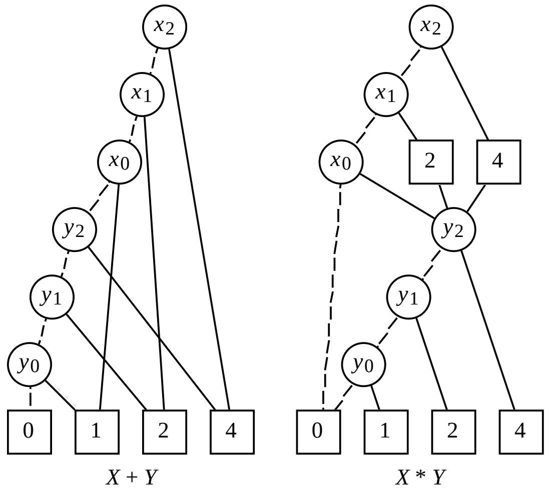

Fig. 5. Representations of Word-Level Sum and Product. The graphs grow linearly with word size.

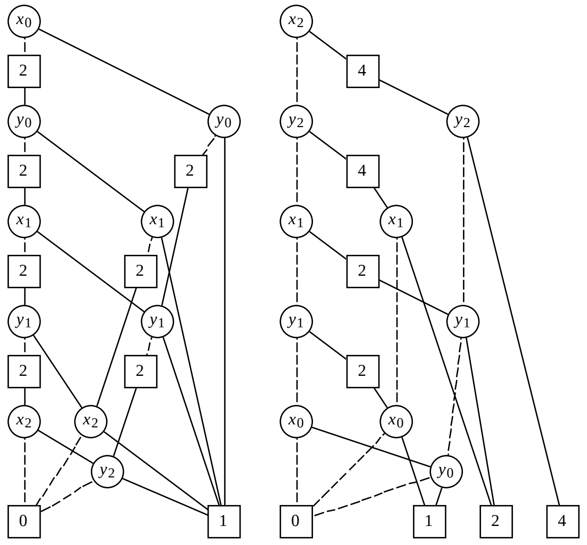

Fig. 6. Representations of Word-Level Product for Other Variable Orderings. The graphs grow linearly with the word size regardless of the variable ordering.

Figure 5 illustrates the *BMD representations of addition and multiplication expressed at a word level. Observe that the sizes of the graphs grow only linearly with the word size n. Word-level addition can be viewed as summing a set of weighted bits, where bits x_i and y_i both have weight 2^i. Word-level multiplication can be viewed as summing a set of partial products of the form $x_i 2^i Y$.

As with BDDs, the representation of a function depends on the variable ordering. For example, Figure 6 shows the *BMDs for word-level multiplication under two additional variable orderings. Observe that although

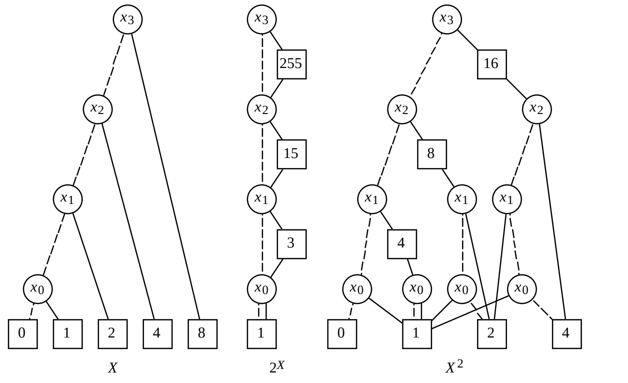

Fig. 7. Representations of Unary Operations at Word-Level. The graph for 2^X grows linearly with the word size, while that for X^2 grows quadratically.

these graphs appear more complex than the one of Figure 5, their complexity still grows only linearly with n. In our experience, *BMDs are much less sensitive to variable ordering than are BDDs.

Figure 7 illustrates the *BMD representations of two unary operations on word-level data. For representing the function c^X (in this case $c = 2$), the *BMD has linear complexity. It expresses the function as a product of factors of the form $c^{2^i x_i} = (c^{2^i})^{x_i}$. Since x_i evaluates to either 0 or to 1, the exponentiation can be linearized as: $a^{x_i} = 1 + (a-1)x_i$. In the graph, a vertex labeled by variable x_i has outgoing edges with weights 1 and $c^{2^i} - 1$ both leading to a common vertex denoting the product of the remaining factors.

For representing the function X^2, both the BMD and the *BMD have quadratic complexity. The representation can be seen to follow a recursive expansion of the function based on the decomposition: $X = X_n = 2^{n-1}x_{n-1} + X_{n-1}$, where X_k denotes the weighted sum of variables x_0 through x_{k-1}. In terms of this decomposition we have:

$$X_n^2 = \left(2^{n-1}x_{n-1} + X_{n-1}\right)^2$$
$$2^{2n-2}x_{n-1}^2 + 2^n x_{n-1}X_{n-1} + X_{n-1}^2$$

Since x_{n-1} is Boolean-valued, we can demote the quadratic term x_{n-1}^2 to a linear term x_{n-1}. Thus, the constant moment for the function is X_{n-1}^2, while the linear moment is $2^{2n-2} + 2^n X_{n-1} = 2^n(X_{n-1} + 2^{n-2})$. In our example with $n = 4$, the left subgraph represents the function X_3^2, while the right side represents the subgraph $16(X_3 + 4)$. Observe that the different constant offsets for each bit cause the growth of the graph to be quadratic rather than linear. That is, there is no sharing between the graphs for the terms $X_{i-1} + 2^{i-2}$ for different values of i. For many applications, this quadratic complexity is acceptable. For example, we could represent the square of a 32-bit number by a graph of around 530 vertices.

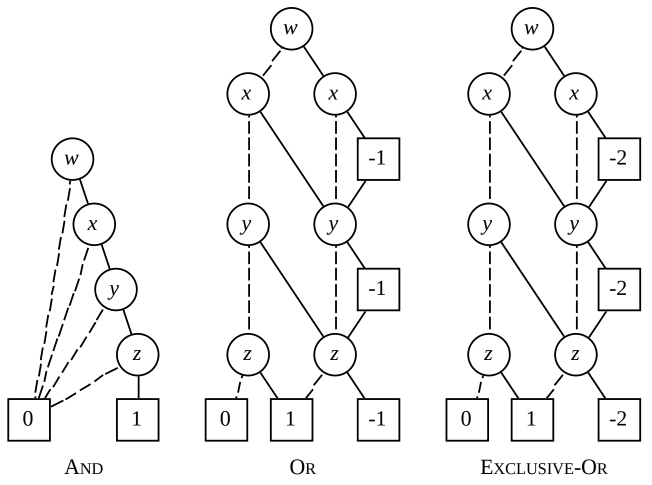

Fig. 8. Representations of Boolean Functions. Representations as *BMDs are comparable in size to BDDs.

4 Representation of Boolean Functions

Boolean functions are just a special case of numeric functions having a restricted range. Therefore such functions can be represented as BMDs or *BMDs. The algebraic structure introduced in Section 2.3 provides a convenient notation for translating Boolean operations into operations on linear functions. In particular, let f and g denote functions have Boolean ranges. Then we can define the standard Boolean operations as:

$$\begin{aligned} \overline{f} &= 1 - f \\ f \wedge g &= f \cdot g \\ f \vee g &= f + g - (f \cdot g) \\ f \oplus g &= f + g - 2(f \cdot g) \end{aligned} \tag{10}$$

Figure 8 illustrates the *BMD representations of several common Boolean functions over multiple variables, namely their Boolean product and sum, as well as their exclusive-or sum. As this figure shows, the *BMD of Boolean functions may have values other than 0 or 1 for edge weights and leaf values. Under all variable assignments, however, the function will evaluate to 0 or to 1. As can be seen in the figure, these functions all have representations that grow linearly with the number of variables, as is the case for their BDD representations. The representation for AND follows due to the parallel between Boolean and linear products. The representation for OR can be seen to follow an iterative structure. In particular, let F_n denote the OR of variables $x_1, x_2, \ldots, x_n$, and G_n denote their NOR, i.e., $G_n = 1 - F_n$. Function F_n can be rewritten as:

$$\begin{aligned} F_n &= x_n \vee F_{n-1} \\ &= x_n + F_{n-1} - (x_n \cdot F_{n-1}) \\ &= F_{n-1} + x_n(1 - F_{n-1}) \\ &= F_{n-1} + x_n G_{n-1} \end{aligned}$$

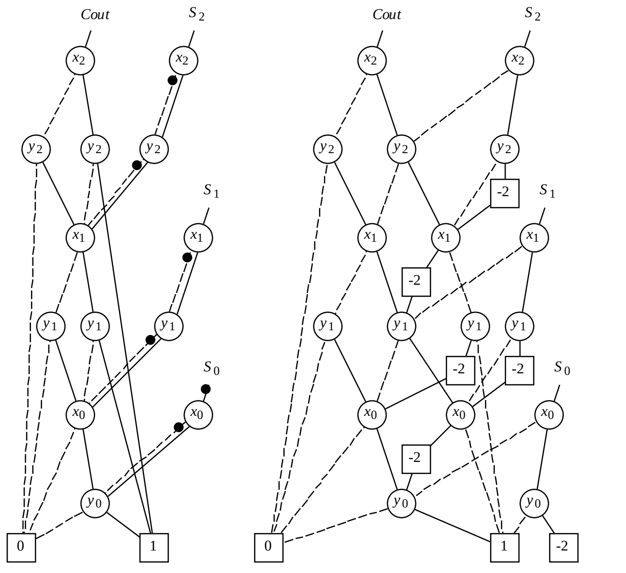

Fig. 9. Bit-Level Representations of Addition Functions. Each graph represents all four outputs of a 3-bit adder.

Thus, the moments of function F_n with respect to variable x_n are F_{n-1} and G_{n-1}. Based on this result, function G_n can be rewritten as:

$$\begin{aligned} G_n &= 1 - F_n \\ &= 1 - F_{n-1} - x_n G_{n-1} \\ &= G_{n-1} + x_n(-G_{n-1}) \end{aligned}$$

Thus, the moments of function G_n with respect to variable x_n are G_{n-1} and $-G_{n-1}$. In the center graph of Figure 8, the vertices on the left side denote the sequence of OR functions, while those on the right side denote the sequence of NOR functions.

The representation for EXCLUSIVE-OR follows a similar iterative structure. It can be generated by defining function F_n to be the EXCLUSIVE-OR of variables $x_1, x_2, ..., x_n$, while letting G_n denote the function $G_n = 1 - 2F_n$. It can be shown that F_n has moments F_{n-1} and G_{n-1}, while G_n has moments G_{n-1} and $-2G_{n-1}$.

Figure 9 illustrates the similarity between BDDs and *BMDs when representing the Boolean functions describing an adder circuit at the bit level. Observe the relation between the word-level representation (Figure 5) and the bit-level representation of addition. Both are functions over variables representing the adder inputs, but the former is a single function yielding an integer value, while the latter is a set of Boolean functions: one for each output signal for the circuit. The relation between these two representations will be discussed more fully in our development of a verification methodology.

The BDD representation shown in Figure 9 employs two techniques to reduce its size [3]. First, it represents a set of functions by a single graph with multiple roots, allowing different functions to share common subgraphs. In fact, the set of functions is maintained in *strong canonical form*, where every function to be represented is denoted by a unique root vertex. The *BMD representation can also use this form of sharing and maintained in strong canonical form. Second, the BDD contains "negative edges" (indicated by dots on the edge) to denote Boolean complementation. The use of edge weights in *BMDs has a similar effect, although edge weights cannot be used to directly represent the complement operation: $\overline{f} = 1 - f$. Observe in any case that the *BMD representation for these functions has a similar structure to the BDD representation. Both grow linearly with the word size, with the *BMD requiring 7 vertices per bit position, and the BDD requiring 5.

In all of the examples shown, the *BMD representation of a Boolean function is of comparable size to its BDD representation. In general this will not always be the case. Enders [12] has characterized a number of different function representations and shown that *BMDs can be exponentially more complex than BDDs, and vice-versa. The two representations are based on different expansions of the function, and hence their complexity for a given function can differ dramatically. In our experience, *BMDs generally behave almost as well as BDDs when representing Boolean functions.

5 Factoring and Other Decision Properties

One powerful property of BDDs is that, given a BDD representation of a function f over a set of variables $\mathbf{x}$, one can easily find solutions to the equation $f(\mathbf{x}) = 0$ by tracing paths from the root to the leaf with value 0. This strength of BDDs is also a limitation. Since any problem that can be expressed as a function f having an efficient BDD representation is amenable to easy solution, this implies that BDDs cannot efficiently represent functions corresponding to intractable problems.

Imagine for example, that it were possible to construct the $2n$ BDDs giving a bit-level representation of multiplication over n-bit integers $\mathbf{x}$ and $\mathbf{y}$. Then we could potentially factor a large number K, by solving the equation:

$$\bigvee_{i=0}^{2n-1} P_i(\mathbf{x}, \mathbf{y}) \oplus k_i \quad = \quad 0$$

where P_i is the function representing bit i of the product, and k_i is the ith bit of K. Observe in this equation that the values k_i are constants, and therefore the computation involves forming the product of either true or complemented multiplier output functions. Experts consider factoring to be a "hard" problem. In fact, the RSA encryption algorithm [27] relies on the assumption that given the public key, one cannot derive the two prime factors of the key in a reasonable amount of time. Thus, one would expect that some step in the above scheme for factoring would break down. In the case of BDDs, the

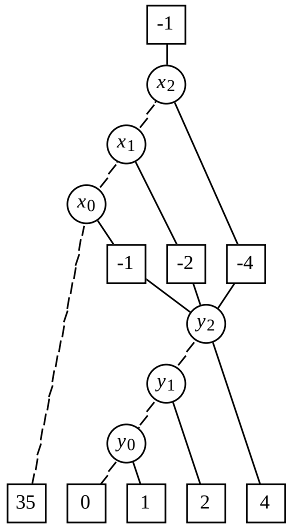

Fig. 10. Representation of Factoring Problem. Solving requires finding variable assignment that evaluates to 0—not an easy task.

problem comes in trying to generate the BDD representations of the functions P_i. It can be shown that these graphs grow exponentially with the word size [5].

Define the task of "finding a zero for function f" as finding a (Boolean) variable assignment such that $f(x) = 0$. We will call a representation for functions "easily invertible" if it is always possible to find a zero for the function in time polynomial in the size of the representation. Both BDDs and MTBDDs have this property—one simply finds a path to the leaf with value 0. One can also show that FDDs are easily invertible [2], even though evaluation does not involve simply following a single path in the graph.

On the other hand, EVBDDs are not easily invertible, assuming $P \neq NP$. The following argument shows that the problem of finding a zero of a function represented by an EVBDD is NP-complete. First, the problem is clearly NP, since given an assignment to the variables, one can evaluate an EVBDD and determine whether the function yields 0 for this assignment. Furthermore, any instance of the NP-complete Partition problem [14] can readily be translated into an EVBDD solution problem. This problem is defined as: given a set of n elements A, where each element i has a nonnegative integer "size" s_i, determine whether there exists a subset A' such that

$$\sum_{i \in A'} s_i = \sum_{i \in A - A'} s_i$$

To translate this into an equation solution problem, let $S = \sum_{i \in A} s_i$, and define the function f as:

$$f(x_1, \ldots, x_n) = -S/2 + \sum_{i=1}^{n} x_i s_i \qquad (11)$$

This function has an EVBDD with n nonterminal vertices. It is similar in structure to that of Figure 2, except that the outgoing solid arc from a vertex with variable x_i has weight s_i, and the root has weight $-S/2$. The challenge of solving this problem for EVBDDs can be seen to lie with the edge weights. One must find a path through the graph such that the edge weights encountered sum to 0.

By a similar argument, one can show that BMDs and *BMDs also do not form easily invertible representations. Both are clearly in NP, since evaluation can be performed in time linear in the graph sizes. Furthermore, both provide linear-sized representations of the function defined in Equation 11. For example, the BMD representation of this function has structure similar to that of Figure 2. The solid arc from a vertex with variable x_i points to a leaf with value s_i, while the dashed arc from the vertex with variable x_0 points to a leaf with value $-S/2$. The *BMD has similar structure, but possibly with weights moved up into the edges.

The challenge of finding a zero of a BMD or *BMD can be seen to lie with the evaluation rule, given by Equation 4—evaluation requires considering multiple paths in the graph. We can readily represent the factoring problem, as shown in Figure 10 by constructing a *BMD representation of the function $X \cdot Y - K$ (in this example $K = 35$). The BMD representation of this function is somewhat more complex, but still of size quadratic in n. The lack of an efficient inversion algorithm prevents one from factoring by this method.

The example of factoring illustrates the fact that the strengths and weaknesses of BDDs versus *BMDs are somewhat orthogonal. Tasks that can easily be performed on BDDs are much more difficult to perform on *BMDs. On the other hand, *BMDs can represent circuit functions that cause exponential blow up for BDDs or to their extensions as MTBDDs and EVBDDs.

6 Algorithms

In this section we describe key algorithms for constructing and manipulating *BMDs. The algorithms have a similar style to their counterparts for BDDs. Unlike operations on BDDs where the complexities are at worst polynomial in the argument sizes, most operations on *BMDs potentially have exponential complexity. We will show in the experimental results, however, that these exponential cases do not arise in our applications.

*6.1 Representation of *BMDs*

We will represent a function as a "weighted pair" of the form $\langle w, v \rangle$ where w is a numeric weight and v designates a graph vertex. Weights can either be maintained as integers or real numbers. Maintaining rational-valued weights follows the same rules as the real case. Vertex $v = \Lambda$ denotes a terminal leaf, in which case the weight

```
pair MakeBranch(variable $x$, pair $\langle w_l, v_l \rangle$, pair $\langle w_h, v_h \rangle$)
{ Create a branch, normalize weights. }
{ Assumes that $x < \mathsf{Var}(v_h)$ and $x < \mathsf{Var}(v_l)$ }
    if $w_h = 0$ then return $\langle w_l, v_l \rangle$
    $w \leftarrow$ NormWeight($w_l$, $w_h$)
    $w_l \leftarrow w_l / w$
    $w_h \leftarrow w_h / w$
    $v \leftarrow$ UniqueVertex($x$, $\langle w_l, v_l \rangle$, $\langle w_h, v_h \rangle$)
    return $\langle w, v \rangle$

vertex UniqueVertex(variable $x$, pair $\langle w_l, v_l \rangle$, pair $\langle w_h, v_h \rangle$)
{ Maintain set of graph vertices without duplication }
    key $\leftarrow$ [$x$, $w_l$, $\mathsf{Uid}(v_l)$, $w_h$, $\mathsf{Uid}(v_h)$]
    found, $v \leftarrow$ LookUp(UTable, key)
    if found then return $v$
    $v \leftarrow$ New(vertex)
    $\mathsf{Var}(v) \leftarrow x$; $\mathsf{Uid}(v) \leftarrow$ Unid();
    $\mathsf{Lo}(v) \leftarrow \langle w_l, v_l \rangle$; $\mathsf{Hi}(v) \leftarrow \langle w_h, v_h \rangle$
    Insert(UTable, key, $v$)
    return $v$

integer NormWeight(integer $w_l$, integer $w_h$)
{ Normalization function, integer weights. }
    $w \leftarrow \gcd(w_l, w_h)$
    if $w_l < 0$ or ($w_l = 0$ and $w_h < 0$)
        then return $-w$
        else return $w$

real NormWeight(real $w_l$, real $w_h$)
{ Normalization function, real weights }
    if $w_l = 0$
        then return $w_h$
        else return $w_l$
```

Fig. 11. Algorithms for Maintaining *BMD. These algorithms preserve a strong canonical form.

```
pair ApplyWeight(wtype $w'$, pair $\langle w, v \rangle$)
{ Multiply function by constant }
    if $w' = 0$ then return $\langle 0, \Lambda \rangle$
    return $\langle w' \cdot w, v \rangle$
```

Fig. 12. Algorithm for Multiplying Function by Weight. This algorithm ensures that edge to a nonterminal vertex has weight 0.

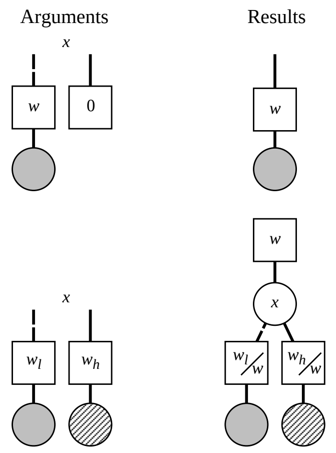

Fig. 13. Normalizing Transformations Made by ***MakeBranch*****.** These transformations enforce the rules on branch weights.

denotes the leaf value. The weight w must be nonzero, except for the terminal case. Each vertex v has the following attributes:

$\mathsf{Var}(v)$ The vertex variable.
$\mathsf{Hi}(v)$ The pair designating the linear moment.
$\mathsf{Lo}(v)$ The pair designating the constant moment.
$\mathsf{Uid}(v)$ Unique identifier for vertex.

Observe that each edge in the graph is also represented as a weighted pair.

6.2 Maintaining Canonical Form

The functions to be represented are maintained as a single graph in *strong canonical form.* That is, pairs $\langle w_1, v_1 \rangle$ and $\langle w_2, v_2 \rangle$ denote the same function if and only if $w_1 = w_2$ and $v_1 = v_2$. We assume that the set of variables is totally ordered, and that all of the vertices constructed obey this ordering. That is, for any vertex v, its variable $\mathsf{Var}(v)$ must be less than any variable appearing in the subgraphs $\mathsf{Lo}(v)$ and $\mathsf{Hi}(v)$.

Maintaining a canonical form requires obeying a set of conventions for vertex creation and for weight manipulation. These conventions are expressed by the pseudocode shown in Figures 11 and 12. The *MakeBranch* algorithm provides the primary means of creating and reusing vertices in the graph. It is given as arguments a variable and two moments, each represented as weighted pairs. It returns a pair representing the function given by Equation 2. It assumes that the argument variable is less than any variable in the argument subgraphs. The steps performed by *MakeBranch* are illustrated in Figure 13. In this figure two moments are drawn as weighted pointers.

When the linear moment is the constant 0, we can simply return the constant moment as the result, since this function is independent of variable x. Observe that this rule differs from the reduction rule for a graph based on a point-wise decomposition such as BDDs. In such cases a vertex can be eliminated when both of its children are identical. This reflects the difference between the two different function decompositions. Our rule for *BMDs is similar to that for FDDs [11, 19].

For other values of the linear moment, the routine first factors out some weight w, adjusting the weights of the two arguments accordingly. We show two versions of a function *NormWeight* according to whether integer or real-valued weights are to be used. For the integer case, we want to extract any common factor while en-

Termination conditions

op	$\langle w_1, v_1\rangle$	$\langle w_2, v_2\rangle$	$\langle w, v\rangle$
$+$	$\langle 0, \Lambda\rangle$		$\langle w_2, v_2\rangle$
$+$		$\langle 0, \Lambda\rangle$	$\langle w_1, v_1\rangle$
$+$	$\langle w_1, v\rangle$	$\langle w_2, v\rangle$	$ApplyWeight(w_1 + w_2,\ \langle 1, v\rangle)$
$*$	$\langle w_1, \Lambda\rangle$		$ApplyWeight(w_1,\ \langle w_2, v_2\rangle)$
$*$		$\langle w_2, \Lambda\rangle$	$ApplyWeight(w_2,\ \langle w_1, v_1\rangle)$
$\div$		$\langle w_2, \Lambda\rangle$	$ApplyWeight(1/w_2,\ \langle w_1, v_1\rangle)$

Table 3. Termination Cases for Apply Algorithms. Each line indicates an operation, a set of terminations, and the returned result.

suring that all weights are integers. Hence we take the greatest common divisor (gcd) of the argument weights. In addition, we adopt the convention that the sign of the extracted weight matches that of the constant moment. This assumes that gcd always returns a nonnegative value. For real-valued weights we adopt the convention that the weighted pair designating the constant moment for a vertex always has weight 0 (only when this moment is the constant 0) or 1. In the former case the weight of the pair designating the first moment will have weight 1. Thus, normalizing real-valued weights involves moving one of the argument weights up and adjusting the other.

Once the weights have been normalized *MakeBranch* calls the function *UniqueVertex* to find an existing vertex or create a new one. This function maintains a table (typically a hash table) where each entry is indexed by a key formed from the variable and the two moments. Every vertex in the graph is stored according to such a key and hence duplicate vertices are never constructed.

Figure 12 shows the code for a function *ApplyWeight* to multiply a function, given as a weighted pair, by a constant value, given as a weight w'. This procedure simply adjusts the pair weight, detecting the special case where the multiplicative constant is 0.

As long as all vertices are created through calls to the *MakeBranch* function and all multiplications by constants are performed by calls to *ApplyWeight*, the graph will remain in strongly canonical form.

6.3 The Apply Operations

As with BDDs, *BMDs are constructed by starting with base functions corresponding to constants and single variables, and then building more complex functions by combining simpler functions according to some operation. In the case of BDDs this combination is expressed by a single algorithm that can apply an arbitrary Boolean operation to a pair of functions. In the case of *BMDs we require algorithms tailored to the characteristics of the individual operations. To simplify the presentation, we show only a few of these algorithms and attempt to do so in as uniform a style as possible. These algorithms are referred to collectively as "Apply" algorithms.

```
pair PlusApply(pair ⟨w1, v1⟩, pair ⟨w2, v2⟩)
{ Compute sum of two functions }
    done, ⟨w, v⟩ ← TermCheck(+, ⟨w1, v1⟩, ⟨w2, v2⟩)
    if done then return ⟨w, v⟩

    w', ⟨w1, v1⟩, ⟨w2, v2⟩ ← Rearrange(+, ⟨w1, v1⟩, ⟨w2, v2⟩)
    key ← [+, w1, Uid(v1), w2, Uid(v2)]
    found, ⟨w, v⟩ ← LookUp(OpTable, key)
    if found then return ApplyWeight(w', ⟨w, v⟩)

    x ← Min(Var(v1), Var(v2))

    { Begin recursive section }
    ⟨w1l, v1l⟩ ← SimpleMoment(⟨w1, v1⟩, x, 0)
    ⟨w2l, v2l⟩ ← SimpleMoment(⟨w2, v2⟩, x, 0)
    ⟨w1h, v1h⟩ ← SimpleMoment(⟨w1, v1⟩, x, 1)
    ⟨w2h, v2h⟩ ← SimpleMoment(⟨w2, v2⟩, x, 1)

    ⟨wl, vl⟩ ← PlusApply(⟨w1l, v1l⟩, ⟨w2l, v2l⟩)
    ⟨wh, vh⟩ ← PlusApply(⟨w1h, v1h⟩, ⟨w2h, v2h⟩))
    { End recursive section }

    ⟨w, v⟩ ← MakeBranch(x, ⟨wl, vl⟩, ⟨wh, vh⟩)
    Insert(OpTable, key, ⟨w, v⟩)
    return ApplyWeight(w', ⟨w, v⟩)

pair SimpleMoment(pair ⟨w, v⟩, variable x, integer b)
{ Find moment of function under special condition. }
{ Variable either at root vertex v, or not present in graph. }
{ b = 0 for constant moment, b = 1 for linear }

    if Var(v) ≠ x
        if b = 0
            then return ⟨w, v⟩
            else return ⟨0, Λ⟩
    if b = 0
        then return ApplyWeight(w, Lo(v))
        else return ApplyWeight(w, Hi(v))
```

Fig. 14. Apply Algorithm for Adding Two Functions. The algorithm is similar to the counterpart for BDDs.

Figure 14 shows the fundamental algorithm for adding two functions. The function *PlusApply* takes two weighted pairs indicating the argument functions and returns a weighted pair indicating the result function. This algorithm can also be used for subtraction by first multiplying the second argument by weight -1. This code closely follows the Apply algorithm for BDDs [3]. It utilizes a combination of recursive descent and "memoizing," where all computed results are stored in a table and reused whenever possible. The recursion is based on the property that taking moments of functions commutes with addition. That is, for functions f and g and for variable x:

$$[f+g]_{\overline{x}} = f_{\overline{x}} + g_{\overline{x}}$$
$$[f+g]_{\delta x} = f_{\delta x} + g_{\delta x}$$

Rearrangements

Arguments		**Results**		
op	Condition	w'	$\langle w_1, v_1\rangle$	$\langle w_2, v_2\rangle$
$*$	$\mathsf{Uid}(v_1) > \mathsf{Uid}(v_2)$	$w_1 \cdot w_2$	$\langle 1, v_1\rangle$	$\langle 1, v_2\rangle$
$*$	$\mathsf{Uid}(v_1) \leq \mathsf{Uid}(v_2)$	$w_1 \cdot w_2$	$\langle 1, v_2\rangle$	$\langle 1, v_1\rangle$
$+$	$\lvert w_1\rvert > \lvert w_2\rvert$	W	$\left\langle w_1/w', v_1\right\rangle$	$\left\langle w_2/w', v_2\right\rangle$
$+$	$\lvert w_1\rvert \leq \lvert w_2\rvert$	W	$\left\langle w_2/w', v_2\right\rangle$	$\left\langle w_1/w', v_1\right\rangle$
$\div$		w_1/w_2	$\langle 1, v_1\rangle$	$\langle 1, v_2\rangle$

Table 4. Rearrangements for Apply Algorithms. $W = \mathit{NormWeight}(w_2,\ w_1)$. These rearrangements increase the likelihood of reusing a previously-computed result.

```
{ Begin recursive section }
$\langle w_{1l}, v_{1l}\rangle \leftarrow \mathit{SimpleMoment}(\langle w_1, v_1\rangle,\ x,\ 0)$
$\langle w_{2l}, v_{2l}\rangle \leftarrow \mathit{SimpleMoment}(\langle w_2, v_2\rangle,\ x,\ 0)$
$\langle w_{1h}, v_{1h}\rangle \leftarrow \mathit{PlusApply}(\mathit{SimpleMoment}(\langle w_1, v_1\rangle,\ x,\ 1),$
        $\langle w_{1l}, v_{1l}\rangle)$
$\langle w_{2h}, v_{2h}\rangle \leftarrow \mathit{PlusApply}(\mathit{SimpleMoment}(\langle w_2, v_2\rangle,\ x,\ 1),$
        $\langle w_{2l}, v_{2l}\rangle)$

$\langle w_l, v_l\rangle \leftarrow \mathit{BinApply}(op, \langle w_{1l}, v_{1l}\rangle, \langle w_{2l}, v_{2l}\rangle)$
$\langle w_h, v_h\rangle \leftarrow \mathit{PlusApply}(\mathit{BinApply}(op, \langle w_{1h}, v_{1h}\rangle,$
        $\langle w_{2h}, v_{2h}\rangle), \langle -w_l, v_l\rangle)$
{ End recursive section }
```

Fig. 15. Recursive Section of Apply Algorithm for Arbitrary Binary Operation. This generic algorithm does not exploit particular properties of the operation.

This routine, like the other Apply algorithms, first checks a set of termination conditions to determine whether it can return a result immediately. This test is indicated as a call to function *TermCheck* having as arguments the operation and the arguments of the operation. This function returns two values: a Boolean value *done* indicating whether immediate termination is possible, and a weighted pair indicating the result to return in the event of termination. Some sample termination conditions are shown in Table 3. For the case of addition, the algorithm can terminate if either argument represents the constant 0, or if the two arguments are multiples of each other, indicated by weighted pairs having the same vertex element.

Failing the termination test, the routine attempts to reuse a previously computed result. To maximize possible reuse it first rearranges the arguments and extracts a common weight w'. This process is indicated as a call to the function *Rearrange* having the same arguments as *TermCheck*. This function returns three values: the extracted weight and the modified arguments to the operation. Some sample rearrangements are shown in Table 4. For the case of addition rearranging involves normalizing the weights according to the same conditions used in *MakeBranch* and ordering the arguments so that the first has greater weight. For example, suppose at some point we compute $6y-9z$. We will extract weight -3 (assuming integer weights) and rearrange the arguments as $3z$ and $-2y$. If we later attempt to compute $15z-10y$, we will be able to reuse this previous result with extracted weight 5.

If the routine fails to find a previously computed result, it makes recursive calls to compute the sums of the two moments according to the minimum variable in its two arguments. In generating the arguments for the recursion, it calls a function *SimpleMoment* to compute the moments. This routine can only compute a moment with respect to a variable that either does not appear in the graph or is at its root, a condition that is guaranteed by the selection of x as the minimum variable in the two graphs. When the variable does not appear in the graph, the constant moment is simply the original function, while the linear moment is the constant 0. When the variable appears at the root, the result is the corresponding subgraph multiplied by the weight of the original argument. The final result of *PlusApply* is computed by calling *MakeBranch* to generate the appropriate function and multiplying this function by the constant extracted when rearranging the arguments.

Observe that the keys for table *OpTable* index prior computations by both the weights and the vertices of the (rearranged) arguments. In the worst case, the rearranging may not be effective at creating matches with previous computations. In this event, the weights on the arcs would be carried downward in the recursion, via the calls to *SimpleMoment.* In effect, we are dynamically generating BMD representations from the *BMD arguments. Thus, if functions f and g have BMD representations of size m_f and m_g, respectively, there would be no more than $m_f m_g$ calls to *PlusApply*, and hence the overall algorithm has worst case complexity $O(m_f m_g)$. As we have seen, many useful functions have polynomial BMD sizes, guaranteeing polynomial performance for *PlusApply.* On the other hand, some functions blow up exponentially in converting from a *BMD to a BMD representation, in which case the algorithm may have exponential complexity. We will see with the experimental results, however, that this exponential blow-up does not occur for the cases we have tried. The termination checks and rearrangements are very effective at stopping the recursion. Enders [12] has shown that the complexity of the addition operation on *BMD grows exponentially in worst case.

The Apply algorithms for other operations have a similar overall structure to that for addition, but differing in the recursive evaluation. Comments in the code of Figure 14 delimit the "recursive section" of the routine. In this section recursive calls are made to create a pair of weighted pointers $\langle w_l, v_l\rangle$ and $\langle w_h, v_h\rangle$ from which the returned result is constructed. For the remaining Apply algorithms we show only their recursive sections.

Figure 15 shows the recursive section for applying an arbitrary binary operation op to a pair of functions. This algorithm can be seen to implement the linearized form $\hat{op}$ defined by Equations 8 and 9. At each recursive step

```
{ Begin recursive section }
$\langle w_{1l}, v_{1l}\rangle \leftarrow$ SimpleMoment($\langle w_1, v_1\rangle$, $x$, 0)
$\langle w_{2l}, v_{2l}\rangle \leftarrow$ SimpleMoment($\langle w_2, v_2\rangle$, $x$, 0)
$\langle w_{1h}, v_{1h}\rangle \leftarrow$ SimpleMoment($\langle w_1, v_1\rangle$, $x$, 1)
$\langle w_{2h}, v_{2h}\rangle \leftarrow$ SimpleMoment($\langle w_2, v_2\rangle$, $x$, 1)

$\langle w_l, v_l\rangle \leftarrow$ MultApply($\langle w_{1l}, v_{1l}\rangle$, $\langle w_{2l}, v_{2l}\rangle$)
$\langle w_{hh}, v_{hh}\rangle \leftarrow$ MultApply($\langle w_{1h}, v_{1h}\rangle$, $\langle w_{2h}, v_{2h}\rangle$)
$\langle w_{hl}, v_{hl}\rangle \leftarrow$ MultApply($\langle w_{1h}, v_{1h}\rangle$, $\langle w_{2l}, v_{2l}\rangle$)
$\langle w_{lh}, v_{lh}\rangle \leftarrow$ MultApply($\langle w_{1l}, v_{1l}\rangle$, $\langle w_{2h}, v_{2h}\rangle$)
$\langle w_h, v_h\rangle \leftarrow$ PlusApply($\langle w_{hh}, v_{hh}\rangle$, PlusApply($\langle w_{hl}, v_{hl}\rangle$,
        $\langle w_{lh}, v_{lh}\rangle$))
{ End recursive section }
```

Fig. 16. Recursive Section for Apply Operation for Multiplying Functions. This operation exploits the ring properties of linear product.

```
AffineSubst(pair $\langle w, v\rangle$, assignment $\mu$, assignment $\beta$)
{ Replace each variable $x$ in function by $\mu(x)\cdot x + \beta(x)$ }
    if $v = \Lambda$ then return $\langle w, v\rangle$
    Key $\leftarrow [v, \mu, \beta]$
    found, $\langle w_t, v_t\rangle \leftarrow$ LookUp(SubstTable, key)
    if found then return ApplyWeight($w$, $\langle w_t, v_t\rangle$)
    $x \leftarrow$ Var($x$)
    $\langle w_l, v_l\rangle \leftarrow$ AffineSubst(Lo($v$), $\mu$, $\beta$)
    $\langle w_h, v_h\rangle \leftarrow$ AffineSubst(Hi($v$), $\mu$, $\beta$)
    $\langle w_l, v_l\rangle \leftarrow$ PlusApply($\langle w_l, v_l\rangle$, ApplyWeight($\beta(x)$, $\langle w_h, v_h\rangle$))
    $\langle w_h, v_h\rangle \leftarrow$ ApplyWeight($\mu(x)$, $\langle w_h, v_h\rangle$)
    $\langle w_t, v_t\rangle \leftarrow$ MakeBranch($x$, $\langle w_l, v_l\rangle$, $\langle w_h, v_h\rangle$)
    Insert(SubstTable, key, $\langle w_t, v_t\rangle$)
    return ApplyWeight($w$, $\langle w_t, v_t\rangle$)
```

Fig. 17. Affine Substitution Algorithm. Each variable in the function is replaced by an affine transformation of the variable.

of the computation in Figure 15, we must sum the moments of the arguments to generate their positive cofactors, recursively apply the operation to these cofactors, and then subtract the constant moment to obtain a linear moment. By computing the positive cofactor at each vertex, we in effect dynamically construct an MTBDD representation of the arguments. Thus, one would expect that this computation would perform poorly unless either the arguments have efficient MTBDD representations, or the termination checks and rearrangements can stop the recursion from expanding into a large number of cases.

Rather than resorting to the generic Apply algorithm of Figure 15, it is preferable to exploit properties of the operation so that the positive cofactors of the arguments do not need to be generated. Figure 16 shows how this can be done for multiplication, using the formulation of linear product given by Equation 7. Each call to *MultApply* requires four recursive calls, plus two calls to *PlusApply*. With the rearrangements shown in Table 4, we can always extract the weights from the arguments. Hence if the arguments have *BMD representations of m_f and m_g vertices, respectively, no more than $m_f m_g$ calls will be made to *MultApply*. Unfortunately, this bound on the calls does not suffice to show a polynomial bound on the complexity of the algorithm. The calls to *PlusApply* may blow up exponentially. Enders [12] has shown that the complexity of the multiplication operation on BMD and *BMD grows exponentially in worst case.

6.4 Affine Substitution

Figure 17 shows an algorithm for performing a very general form of function evaluation we will call *affine substitution*. The idea is to substitute for each variable x a function of the form $mx + b$. The result will be a function over the same set of variables, or possibly a subset of these variables. By selecting different values of m and b we can obtain many useful substitutions. For example, with $b = a$ and $m = 0$, we obtain the result of assigning value a to the variable. Thus, this operation generalizes the linear evaluation shown in Equation 5, including accounting for the edge weights. With $m = 1$ and $b = 0$, an identity substitution will be performed, and hence the algorithm can be used for partial evaluation, where some variables are assigned constants, while others are unchanged. With $m = -1$ and $b = 1$, we replace the variable by its Boolean complement.

The algorithm is shown as having functional arguments μ and β. When applied to a variable x, these "assignments" yield the constant factors to be used in the affine substitution. The algorithm follows from the linear expansion of function f with respect to each variable x. Given that $f = f_{\overline{x}} + x f_{\delta x}$, substituting $mx + b$ for x yields:

$$f|_{x \leftarrow mx+b} \;=\; f_{\overline{x}} + b f_{\delta x} + x m f_{\delta x}$$

and hence this substitution yields a function with moments $f_{\overline{x}} + b f_{\delta x}$ and $m f_{\delta x}$.

The routine maintains a table of previously computed substitutions. Observe that for given assignments μ and β, recursive calls are generated from a vertex only once. The total number of calls to *AffineSubst* is therefore linear in the graph size. Of course, the resulting calls to *PlusApply* could cause the algorithm to blow up exponentially. For the special case of full evaluation, however, where $\mu(x) = 0$ for all variables x, each recursive call must return a constant function, and hence the overall complexity is linear.

7 Verification Methodology

Figure 18 illustrates schematically an approach to circuit verification originally formulated by Lai and Sastry [20] using EVBDDs. The overall goal is to prove a correspondence between a combinational circuit, represented

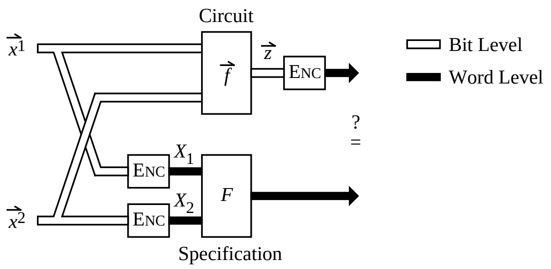

Fig. 18. Formulation of Verification Problem. The goal of verification is to prove a correspondence between a bit-level circuit and a word-level specification

by a vector of Boolean functions $\mathbf{f}$, and the specification, represented by the word level function F. More precisely, assume that the circuit inputs are partitioned into vectors of binary signals $\mathbf{x}^1, \ldots, \mathbf{x}^k$ (in the figure $k = 2$). For each set of signals $\mathbf{x}^i$, we are given an encoding function $\textsc{Enc}_i$ describing a word level interpretation of the signals. This function will typically be a standard encoding, such as a 16-bit two's complement integer. The circuit implements a set of Boolean functions over the inputs, denoted by the vector of functions $\mathbf{f}(\mathbf{x}^1, \ldots, \mathbf{x}^k)$. Typically this circuit is given in the form of a network of logic gates. Furthermore, we are given an encoding function $\textsc{Enc}_o$ defining a word level interpretation of the output. Finally, we are given as specification a word-level function $F(X_1, \ldots, X_k)$. The task of verification is then to prove the equivalence:

$$\textsc{Enc}_o(\mathbf{f}(\mathbf{x}^1, \ldots, \mathbf{x}^k)) \;=\; F(\textsc{Enc}_1(\mathbf{x}^1), \ldots, \textsc{Enc}_k(\mathbf{x}^k)) \qquad (12)$$

That is, the circuit output, interpreted as a word should match the specification when applied to word interpretations of the circuit inputs.

*BMDs provide a suitable data structure for this form of verification, because they can represent both bit-level and word-level functions efficiently. EVBDDs can also be used for this purpose, but only for the limited class of circuit functions having efficient word-level representations as EVBDDs. By contrast, BDDs can only represent bit-level functions, and hence the specification must be expanded into bit-level form. While this can be done readily for standard functions such as binary addition, a more complex function such as binary to BCD conversion would be difficult to specify at the bit level.

7.1 Component Verification

The component partitioning allows us to efficiently represent both their bit-level and word-level functions. This allows the test of Equation 12 to be implemented directly. As an example, consider the adder circuit having bit-level functions given by the *BMD of Figure 9, where this *BMD is derived from a gate-level representation of the circuit using Apply operations, much as would be done with BDDs. The word-level specification is given

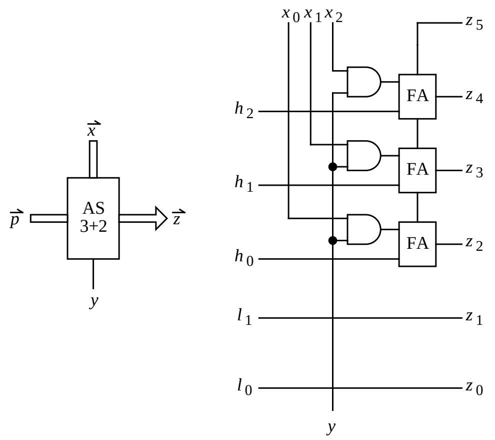

Fig. 19. Bit-Level Representation of Add-Stepper. This circuit is a basic component of the Multplier.

by the left-hand *BMD of Figure 5. In generating the *BMD from the specification we are also incorporating the requirement that input words X and Y have an unsigned binary encoding. Given that the output is also to have an unsigned binary encoding, we would use our Apply algorithms to convert the bit-level circuit outputs to the word level as:

$$P \;=\; 2^0 \cdot S_0 + 2^1 \cdot S_1 + 2^2 \cdot S_2 + 2^3 \cdot Cout$$

We would then compare the *BMD for P to the one shown on the left-hand side of Figure 5.

For circuits that can be represented efficiently as *BMDs at both the bit and the word level, the test of Equation 12 can be implemented directly. As an example, consider an $n+m$-Add-Stepper, illustrated in Figure 19 for $n = 3$ and $m = 2$. This circuit forms a basic building block for the class of multipliers we will verify. It has as inputs an $n+m$-bit partial product input $\mathbf{p}$, split into high order elements $h_{n-1}, \ldots, h_0$, and low order elements $l_{m-1}, \ldots, l_0$. This naming convention is adopted to expedite the multiplier verification, as will be discussed shortly. The other inputs are an n-bit multiplicand $x_{n-1}, \ldots, x_0$, and a single bit multiplier y. It produces an $n+m+1$ bit partial product output $z_{n+m}, \ldots, z_0$.

The bit-level structure for the circuit is shown in the figure, consisting of $\textsc{And}$ gates and full adders blocks. Each full adder has three inputs a, b, and c. It produces a sum output at the right hand side with function $a \oplus b \oplus c$. It has a carry output at the top, with function expressed in terms of linear operators as $a \cdot b + a \cdot c + b \cdot c - 2a \cdot b \cdot c$. From this representation we can use the algorithms *PlusApply* and *MultApply* to generate a *BMD representation of $f_i(\mathbf{p}, \mathbf{x}, y)$, the function at each output z_i for $0 \leq i \leq n+m$.

The word-level specification for an $n+m$-Add-Stepper is simply $P + 2^m \cdot y \cdot X$, where P and X are the word-level interpretations of the partial product and multiplicand

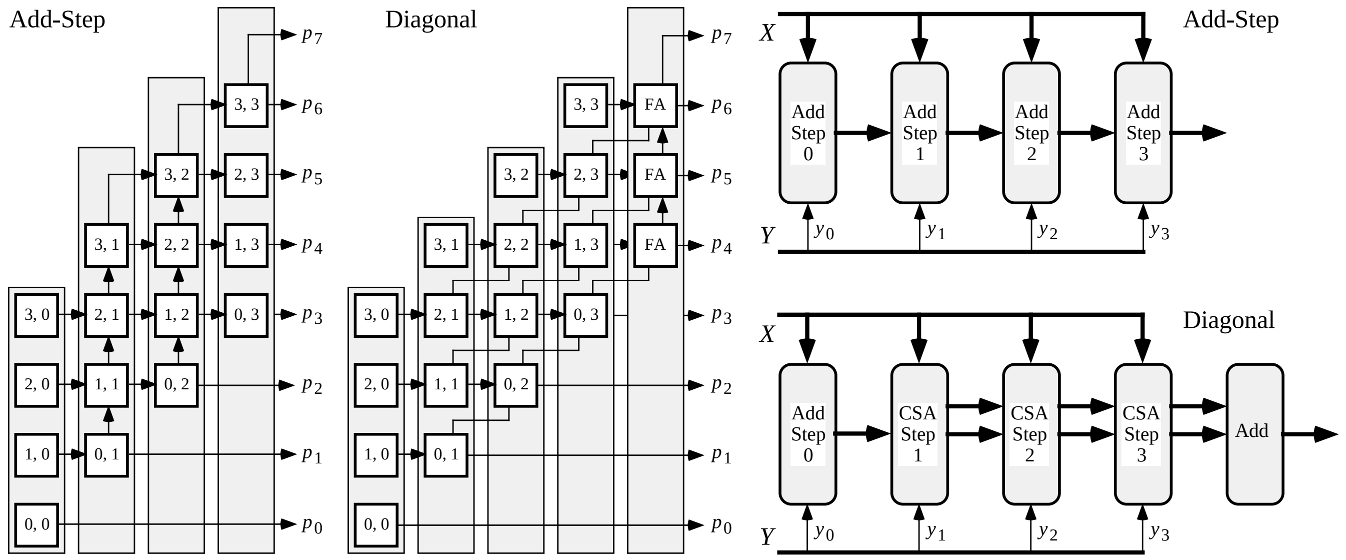

Fig. 20. Multiplier Circuits Different Levels of Abstraction Each square contains an AND gate and a full adder. The vertical rectangles indicate the word-level partitioning yielding the representations shown on the right.

inputs. Both of these inputs are encoded as unsigned integers, as is the output. Verification therefore involves proving that the weighted sum of the bit-level output functions: $\sum_{i=0,n+m} 2^i f_i$ is equivalent to the word-level specification. As with BDDs, this process can be completely automated and works well even for more complex realizations such as carry-lookahead adders.

7.2 Hierarchical Verification

For larger scale circuits, representing the bit-level functionality becomes too cumbersome and hence the method described above cannot be applied directly. For example, attempting to construct the bit-level functions for a multiplier would cause exponential blow-up with *BMDs, just as it does with BDDs. Instead, we can follow a hierarchical approach in which the overall circuit is divided into components, each having a word-level specification. Verification then involves proving 1) that each component implements its word-level specification, and 2) that the composition of the word-level component functions matches the specification. This approach works well for circuits in which the components have simple word-level specifications. Such is the case for most arithmetic circuits.

Figure 20 illustrates the design of two different 4-bit multipliers. Each box labeled i, j in the figure represents a "cell" consisting of an AND gate to form the partial product $x_i \wedge y_j$, and a full adder to add this bit into the product. The vertical rectangles in the figure indicate a word-level partitioning of the circuits, yielding the component interconnection structure shown on the upper right. All word-level data in the circuit uses an unsigned binary encoding. Considering the design labeled "Add-Step", each "Add Step i" component has as input the multiplicand word X, one bit of the multiplier y_i, and a (possibly 0) partial sum input word PI_i. It generates a partial sum word PO_i, where the functionality is specified as $PO_i = PI_i + 2^i \cdot y_i \cdot X$.

Verifying the multiplier therefore involves two steps. First, we can verify that each component implements its specification as mentioned in Section 7.1. Then, we must prove that the composition of the word-level functions matches that of integer multiplication, i.e.,

$$\begin{aligned} &0 + 2^0 \cdot y_0 \cdot X + 2^1 \cdot y_1 \cdot X + 2^2 \cdot y_2 \cdot X + 2^3 \cdot y_3 \cdot X \\ &= \left(\textstyle\sum_{i=0,3} 2^i \cdot y_i\right) \cdot X \\ &= X \cdot Y \end{aligned}$$

Observe that upon completing this process, we have truly verified that the circuit implements unsigned integer multiplication. By contrast, BDD-based approaches just show that a circuit is equivalent to some (hopefully) "known good" realization of the function. For such a simple example, one can readily perform the word-level algebraic manipulation manually. For more complex cases, however, we would like our verifier to compose and compare the functions automatically.

7.3 Abstracting Carry Save Adders

In verifying actual multiplier circuits, we often encounter "carry save adders" (CSAs), requiring an extension to the methodology. For example, the multiplier design labeled "Diagonal" in Figure 20 is similar to the Add-Step design, but where the carry output from each cell is directed to the cell diagonally up and right, rather than directly up. This modification requires an additional stage of full adders (FAs) to generate the final result, but it also shortens the critical path length. Circuit C6288 of the ISCAS benchmarks has this form, with 16-bit input word sizes and with each full adder realized by 9 NOR gates.

In forming a word-level partitioning of the circuit, shown in the lower right of Figure 20, we see that all but the first and last components have two partial sum inputs and two partial sum outputs. Each "CSA Step i" component can be represented at the word level as having input words SI_i and CI_i and output words SO_i and CO_i. The full adders take the form of a carry save adder, reducing three input words to two. The word-level functions realized by a carry save adder do not have a simple description in terms of operations such as addition and multiplication. Thus we cannot directly abstract their behavior up to the word level.

To verify circuits containing CSAs we exploit the fact that the correctness of the overall circuit behavior does not depend on the individual CSA output functions, but rather on their combined values. A CSA has the property that the sum of its two outputs is equal to the sum of its three inputs, perhaps weighting some inputs or outputs by powers of two. We can give a word-level specification for CSA Step i as:

$$SO_i + 2^{i+1} \cdot CO_i \quad = \quad SI_i + 2^i \cdot CI_i + 2^i \cdot y_i \cdot X$$

Rearranging terms, we can view output SO_i as dependent on CO_i:

$$SO_i \quad = \quad SI_i + 2^i \cdot CI_i + 2^i \cdot y_i \cdot X - 2^{i+1} \cdot CO_i \quad (13)$$

In verifying component CSA Step i we verify this equivalence using the *BMD representation of component output CO_i.

In composing the word-level functions, we replace function CO_i by the unsigned integer C_i represented by a vector of Boolean variables $\mathbf{c}^i$. That is, function C_i becomes input CI_{i+1} to the following stage, while function

$$SI_i + 2^i \cdot CI_i + 2^i \cdot y_i \cdot X - 2^{i+1} \cdot C_i$$

becomes input SI_{i+1}. In doing this, we effectively abstract the detailed value, treating word CO_i as an arbitrary unsigned binary-encoded integer. Verifying that the final output functions match the word-level specification $X \cdot Y$ indicates that the overall circuit behavior is correct.

One way to view the methodology described above is that at the component level we treat the carry outputs as existentially quantified—for the particular carry functions implemented by the CSA, Equation 13 must hold. On the other hand, we treat these values as universally quantified when composing the word-level component functions—for any values of the carry output word, the circuit realizes a multiplier as long as the sum output satisfies Equation 13. Such a methodology is conservative—if the verifier succeeds we are guaranteed the circuit is correct, but if it fails it may simply indicate that the overall behavior depends on the detailed sum and carry output functions rather than on their relative values. All of the multiplier circuits containing CSAs we have encountered to date can be successfully verified despite this conservatism.

Circuit	CPU Time (Min.)			Memory (MB)		
	16	64	256	16	64	256
Add-step	0.04	0.9	18.8	0.7	1.1	6.5
CSA XOR cells	0.06	1.2	21.8	0.8	1.3	9.0
CSA NOR cells	0.06	1.3	22.7	0.8	1.3	9.5
Booth	0.1	2.5	33.3	0.8	1.6	14.4
Bit-Pair	0.1	1.6	29.6	0.8	1.9	13.9

Table 5. Verification Results for Combinational Multipliers. Results are shown for three different word sizes.

7.4 *Experimental Results*

Table 5 indicates the results for verifying a number of multiplier circuits. Performance is expressed as the number of CPU minutes and the number of megabytes of memory required on a SUN Sparcstation 10. Observe that the computational requirements are quite reasonable even up to circuits with 256-bit word sizes, requiring up to 653,056 logic gates. The time for the multiplier verification grows quadratically in the word size. Given that the hardware complexity scales quadratically in the word size, this performance is reasonable. We know of no other automated verification of a circuit of such size, regardless of logic function. The design labeled "CSA NOR cells" is based on the logic design of ISCAS '85 benchmark C6288, a 16-bit version. Our verification of this circuit requires less than 4 seconds.

These results are especially appealing in light of prior results on multiplier verification. A brute force approach based on BDDs cannot get beyond even modest word sizes. Yang *et al* [28] have successfully built the OBDDs for a 16-bit multiplier C6288. This required over 40 million vertices and about 3.8GB memory on a 64-bit machine (i.e. 1.9GB on a 32-bit machine). Increasing the word size by one bit causes the number of vertices to increase by a factor of approximately 2.87, and hence even more powerful computers will not be able to get much beyond this point.

Burch [6] has implemented a BDD-based technique for verifying certain classes of multipliers. His method effectively creates multiple copies of the multiplier and multiplicand variables, leading to BDDs that grow cubically with the word size. This approach works for multipliers, such as ours, that form all possible product bits of the form $x_i \wedge y_j$ and then sum these bits. Burch reports verifying C6288 in 40 minutes on a Sun-3 using 12 MBytes of memory. The limiting factor in dealing with larger word sizes would be the cubic growth in memory requirement. Furthermore, this approach cannot handle multipliers that use multiplier recoding techniques, although Burch describes extensions to handle some forms of recoding.

Jain *et al* [18] have used Indexed Binary Decision Diagrams (IBDDs) to verify several multiplier circuits. This form of BDD allows multiple repetitions of a variable along a path from root to leaf. They were able to verify C6288 in 22 minutes of CPU time on a SUN-4/280, generating a total of 149,498 graph vertices. They were also able to verify a multiplier using Booth encoding, but this required almost 4 hours of CPU time and generated over 1 million vertices in the graphs.

8 Related Work

Our approach must partition a circuit into a hierarchical form, with the "leaf" elements being small enough to have an efficient bit-level representation. Unfortunately, some designs to not have a clean, hierarchical structure. For example, logic synthesis tools will often flatten a design and perform transformations across module boundaries. It would be difficult for a user to define a hierarchical structure for the optimized design. To overcome this constraint, Hamaguchi *et al* [16] proposed a method that constructs a *BMD representation of a combinational circuit by working backward from the encoded primary outputs to the primary inputs. This approach avoids creating a bit-level representation of any part of the circuit and hence can be applied to a complete, flattened netlist. For a 64$\times$64 multiplier, Hamaguchi *et al.*[16] reported 22,340 seconds of CPU time on Sun Sparc 10/51 machine. Their results show that the complexity grows cubically with the word size for integer multipliers. Although their method requires more computational effort than our's, its higher degree of automation is attractive. Unfortunately, their experiments showed that even minor circuit design errors cause an exponential blow-up of the *BMDs. Generally, users would like a tool to help them diagnose defective circuits in addition to confirming the correctness of good circuits.

Adapting the idea of OKFDDs [11], Clarke, *et al* [9] have developed a hybrid between MTBDDs and BMDs, which they call Hybrid Decision Diagrams (HDDs). In their representations, each variable can use one of six different decompositions. These include the Shannon and positive Davio, corresponding to point-wise and moment decompositions, respectively. In their experience, the variables for the control signals should use Shannon decomposition to achieve smaller graph sizes. Clarke *et al* [10] presented word-level SMV, a verification tool adding a word-level specification to a BDD-based symbol model checker. This tool uses BDDs for Boolean functions, HDDs for integer functions and a layered backward substitution method (a variant of Hamaguchi's method) [7]. For integer multipliers, their complexity grows cubically, but the constant factor is much smaller than Hamaguchi's. Chen *et al* [7] have applied word-level SMV to verify arithmetic circuits in one of Intel's processors. In this work, floating-point circuits are partitioned into several sub-circuits whose specifications can be expressed in terms of integer operations, because HDDs can not represent floating-point functions efficiently. Each sub-circuits are verified in a flattened manner.

9 Conclusions

*BMDs provide an efficient representation for functions mapping Boolean variables to numeric values. They can represent a number of word-level functions in a compact form. They also represent Boolean functions with complexity comparable to BDDs. They are therefore suitable for implementing a verification methodology in which bit-level circuits are compared to word-level specifications. By exploiting circuit hierarchy, we are able to verify circuits having functions that are intractable to represent at the bit level.

Verification of multipliers and other arithmetic circuits using *BMDs seems quite promising, but these ideas must be tested and extended further. In developing a comprehensive verification system based on our hierarchical methodology, it would be good to have a "proof manager" that keeps track of what components have been verified, checks for compatibility between encodings, etc.

The hierarchical verification methodology described here extends to sequential circuits as well. For modeling such circuits, one could implement a form of symbolic simulator, where blocks of the circuit can be modeled at either the bit or the word level. For example, one could verify a sequential multiplier by first simulating a single cycle at the bit level to show it implements an add step, and then a series of cycles at the word level to show this implements multiplication.

Our method shows some promise for verifying floating point hardware, although difficult obstacles must be overcome. Using a version that supports rational numbers, we can efficiently represent the word level functions denoted by standard floating point formats. This fact follows from our ability to represent integer formats plus exponentials. Floating point hardware, however, only computes approximations of arithmetic functions. Thus, verification requires proving equivalence within some tolerance, rather than the strict equivalence of the current methodology. It is unclear whether such a test can be performed efficiently.

Many techniques developed for improving the efficiency and compactness of BDDs could be extended to *BMDs. Among these are dynamic variable reordering [24], and loosening the ordering requirement from a uniform total ordering to one in which variables may appear in different orders along different paths in the graphs [15,26]. Our experience thus far has been that variable ordering is not as critical when representing functions at the word level as it is with bit-level representations. Nonetheless, these ideas bear further investigation.

References

1. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F .Somenzi, "Algebraic decision diagrams and their applications," *International Conference on Computer-Aided Design*, November, 1993, pp. 188–191.
2. B. Becker, R. Drechsler, and R. Werchner, "On the relation between BDDs and FDDs," *Information and Computing*, Vol 123, No. 2, pp. 185-197, Dec. 1995.
3. K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," *27th Design Automation Conference*, June, 1990, pp. 40–45.
4. R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.
5. R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.
6. J. R. Burch, "Using BDDs to verify multipliers," *28th Design Automation Conference*, June, 1991, pp. 408–412.
7. Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking" *Formal Methods in Computer-Aided Design*, 1996, pp. 19-33.
8. E. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang, "Spectral transforms for large Boolean functions with application to technology mapping," *30th Design Automation Conference*, June, 1993, pp. 54–60.
9. E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid decision diagrams: Overcoming the limitations of MTBDDs and BMDs" *International Conference on Computer-Aided Design*, 1995, pp. 159-163.
10. E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking—Avoiding the Pentium FDIV error," *33rd Design Automation Conference*, 1996, pp 645-648.
11. R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski, "Efficient representation and manipulation of switching functions based on ordered Kronecker function decision diagrams," *31st Design Automation Conference*, June, 1994, pp. 415–419.
12. R. Enders, "Note on the complexity of binary moment diagram representations," *IFIP WG 10.5 workshop on applications of Reed-Muller expansion in circuit design*, pp. 191-197,1995.
13. M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal Methods in System Design*, Vol. 10, No. 2/3, (April/May 1997), pp. 149–170.
14. M. R. Garey, and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, 1979.
15. J. Gergov, and C. Meinel, "Efficient analysis and manipulation of OBDD's can be extended to FBDD's," *IEEE Transactions on Computers*, Vol. 43, No. 10 (Oct., 1994), pp. 1197–1209.
16. K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," *International Conference on Computer-Aided Design*, 1995, pp. 78-82.
17. P. L. Hammer (Ivănescu), and S. Rudeanu, *Boolean Methods in Operations Research*, Springer-Verlag, 1968.
18. J. Jain, J. Bitner, M. S. Abadir, J. A. Abraham and D. S. Fussell ,"Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions," *IEEE Transactions on Computers*, Vol. C-46, No. 11 (November, 1997), pp. 1230–1245.
19. U. Kebschull, E. Schubert, and W. Rosentiel, "Multilevel logic based on functional decision diagrams," *European Design Automation Conference*, 1992, pp. 43–47.
20. Y.-T. Lai, and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," *29th Design Automation Conference*, June, 1992, pp. 608–613.
21. Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 8 (August, 1994), pp. 959–975.
22. Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "FGILP: An integer linear program solver based on function graphs," *International Conference on Computer-Aided Design*, November, 1993, pp. 685–689.
23. R. M. M. Oberman, *Digital Circuits for Binary Arithmetic*, John Wiley and Sons, 1979.
24. R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *International Conference on Computer-Aided Design*, November, 1993, pp. 42–47.
25. F. F. Sellers, M. Y. Hsiao, and C. L. Bearnson, "Analyzing errors with the Boolean difference," *IEEE Transactions on Computers*, Vol. C-17 (1968), pp. 676–683.
26. D. Sieling, and I. Wegener, "Graph driven BDDs—a new data structure for Boolean functions," *Theoretical Computer Science*, Vol. 141, No. 1–2, (1994), pp. 283–310.
27. R. D. Silverman, "Massively distributed computing and factoring large integers," *Communications of the ACM*, Vol. 34, No. 11 (November, 1991), pp. 95–103.
28. B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron, "Space- and time-efficient BDD construction via working set control," *Asian-South Pacific Design Automation Conference*, February, 1998, pp. 423–432.