Verification of Arithmetic Circuits with Binary Moment Diagrams*

Randal E. Bryant, Yirng-An Chen
Carnegie Mellon University
Pittsburgh, PA 15213

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 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 [2] 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, 5, 9]. 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) [8], and the additive edge weights found in Edge-Valued Binary Decision Diagrams (EVBDDs) [9].

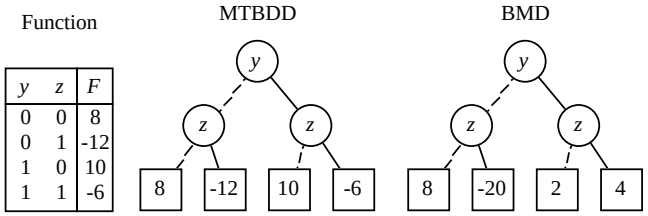

Figure 1: **Example Function Decompositions.** MTBDDs are based on a pointwise decomposition (left), while BMDs are based on a linear decomposition (right).

*BMDs are particularly effective for representing digital systems at the word level, where sets of binary signals are interpreted as encoding integer values. Common integer 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 component modules such as add steppers, Booth steppers, and carry save adders described at both the bit level (in terms of logic gates) and at the word level (as algebraic expressions). Using a methodology proposed by Lai and Vrudhula [9], we verify that the bit-level implementation of each block implements its word-level specification. At the higher level (or levels), a system is described as an interconnection of components 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, 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. The *BMD Data Structure

*BMDs represent functions having Boolean variables as arguments and numeric values as results. Their structure is similar to that of Ordered BDDs, except that they are based on a "moment" decomposition, and they have numeric values for terminal values and edge weights. As with OBDDs we assume there is some total ordering of the variables such that variables are tested according to this ordering along any path from the root to a leaf

* 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.

32nd ACM/IEEE Design Automation Conference ®

2.1. Function Decompositions

To illustrate 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. BDDs are based on a pointwise decomposition, characterizing a function by its value for every possible set of argument values. By extending BDDs to allow numeric leaf values, the pointwise decomposition leads to a "Multi-Terminal" BDD (MTBDD) representation of a function [5] (also called "ADD" [1]), as shown on the left side of Figure 1. In this drawing, 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 $1 - y$. Expanding and simplifying the resulting expression yields:

$$\begin{aligned} F(x,y) &= \begin{bmatrix} 8 & (1-y) & (1-z) & + \\ -12 & (1-y) & z & + \\ 10 & y & (1-z) & + \\ -6 & y & z & \end{bmatrix} \\ &= 8 - 20z + 2y + 4yz \end{aligned}$$

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, while the solid line indicates the case where the function varies linearly. Observe that the leaf values correspond to the coefficients in the linear expansion.

Generalizing from this example, one can view each vertex in the graphical representation of a function as denoting the decomposition of a function with respect to the vertex variable. The different representations can be categorized according to which decomposition they use.

Boolean function f can be decomposed in terms of variable x 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. Term f_x (respectively, $f_{\overline{x}}$) denotes 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. This decomposition is the basis for the BDD representation.

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

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

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 [9] are based on such a pointwise decomposition. As with BDDs, each vertex describes a function in terms of its decomposition with respect to the variable labeling the vertex. The two outgoing arcs denote the negative and positive cofactors with respect to this variable.

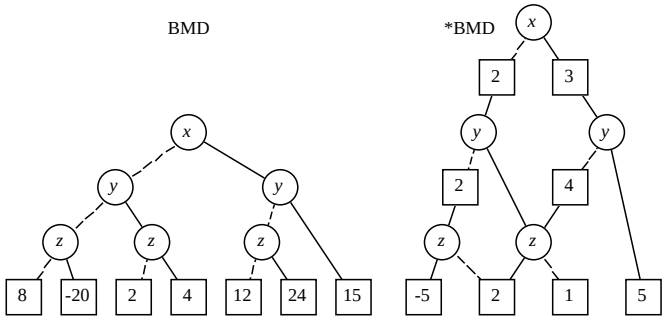

Figure 2: **Example of BMD vs. *BMD.** Both represent the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. *BMDs have weights on the edges that combine multiplicatively.

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

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

where $f_x = f_x - f_{\overline{x}}$ 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_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. Each vertex of a BMD describes a function in terms of its moment decomposition with respect to the variable labeling the vertex. The two outgoing arcs denote the constant and linear moments of the function with respect to the variable.

The moment decomposition of Equation 2 is analogous to the Reed-Muller expansion 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 commonly known as the *Boolean difference* of f with respect to x, and in many ways is analogous to our linear moment. Other researchers [8] 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 Weights

The BMD data structure encodes numeric values only in the terminal vertices. As a second refinement, we adopt the concept of edge weights, similar to those used in EVBDDs. In our case, however, edge weights combine multiplicatively, and hence we call these data structures *BMDs. As an illustration, Figure 2 shows representations of the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. In the BMD representation, leaf values correspond to the coefficients in the linear 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 *BMD shown in Figure 2 indicates how *BMDs are able to exploit the sharing of common subexpressions. In our drawings of

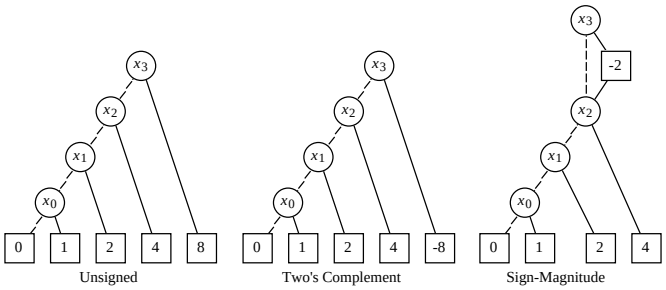

Figure 3: **Representations of Integers.** All commonly used encodings can be represented with linear complexity.

*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. Several rules for manipulating edge weights can be formulated that guarantee the resulting graph form is canonical. For representing functions with integer ranges, we require that the edge weights for the two branches leaving a vertex be relatively prime. We also require that weight 0 only appear as a terminal value, and that when a node has one such branch, the other branch has weight 1. This property is maintained by the way in which the *BMDs are generated, in a manner analogous to BDD generation methods. For the remainder of the presentation we will consider only *BMDs, The effort required to implement weighted edges is justified by the savings in graph sizes.

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 $\vec{x}$ represent a vector of Boolean variables: $x_{n-1}, \ldots, x_1, x_0$. These variables can be considered to represent an integer X according to some encoding, e.g., unsigned binary, two's complement, BCD, etc. Figure 3 illustrates the *BMD representations of several common encodings for integers. An unsigned number is encoded as a sum of weighted bits. The *BMD representation has a simple linear structure with the different weights forming the leaf values. For representing signed numbers, we assume x_{n-1} is the sign bit. The two's complement encoding has a *BMD representation similar to that for unsigned integers, but with bit x_{n-1} having weight -2^{n-1}. For a one's complement encoding (not shown), the sign bit has weight $-2^{n-1}+1$. Sign-magnitude integers also have *BMD representations of linear complexity, but with the constant moment with respect to x_{n-1} scaling the remaining unsigned number by 1, and the linear moment scaling the number by -2. In evaluating the function for $x_{n-1} = 1$, we would add these two moments effectively scaling the number by -1.

3.1. Word-Level Operations

Figure 4 illustrates the *BMD representations of several common arithmetic operations on word-level data. 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 $x_i 2^i Y$. In representing the function c^X (in this case $c = 2$), the *BMD expresses the function as a product of factors of the form $c^{2^i x_i} = (c^{2^i})^{x_i}$. In the graph, a vertex labeled by variable x_i has outgoing edges with weights 1

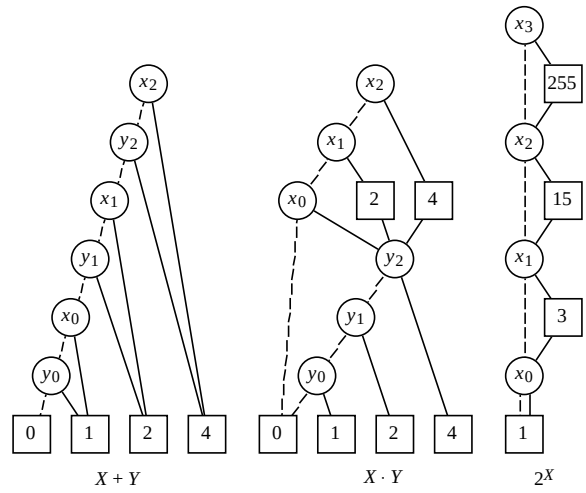

Figure 4: **Representations of Word-Level Sum, Product, and Exponentiation.** The graphs grow linearly with word size.

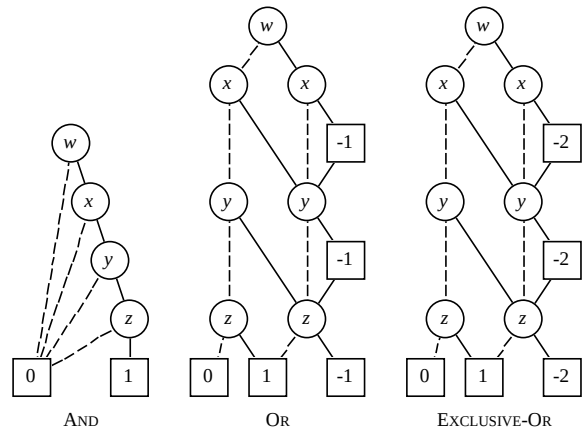

Figure 5: **Representations of Boolean Functions.** Representations as *BMDs are comparable in size to BDDs.

and $c^{2^i} - 1$ both leading to a common vertex denoting the product of the remaining factors. Interestingly, the sizes of these representations are hardly sensitive to the variable ordering—they remain of linear complexity in all cases. We have found that variable ordering is much less of a concern when representing word-level functions with *BMDs than it is when representing Boolean functions with BDDs.

These examples illustrate the advantage of *BMDs over other methods of representing word-level functions. MTBDDs are totally unsuited—the function ranges are so large that they always require an exponential number of terminal vertices. EVBDDs have linear complexity representing word-level data and for representing "additive" operations (e.g, addition and subtraction) at the word level. On the other hand, they have exponential size when representing more complex functions such as multiplication, and exponentiation.

4. Representation of Boolean Functions

In verifying arithmetic circuits, we abstract from the bit-level representation of a circuit, where each signal is binary-valued, to a word level, where bundles of signals encode words of data. In performing this transformation we must represent both Boolean and word-level

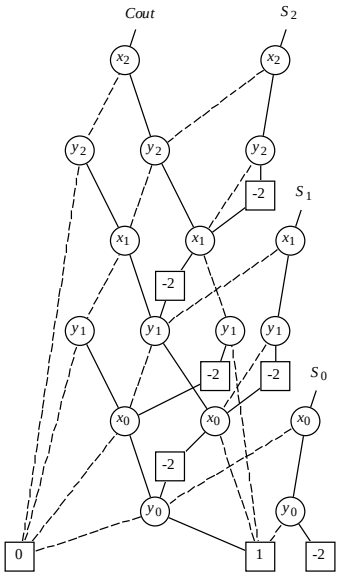

Figure 6: **Bit-Level Representation of Addition Functions.** The graph represents all four outputs of a 3-bit adder.

functions. Hence we require our data structure to be suitable for representing Boolean functions as well.

Boolean functions are just a special case of numeric functions having a restricted range. Therefore such functions can be represented as *BMDs. Figure 5 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.

Figure 6 shows the the bit-level representation of a 3-bit adder. It represents the 4 adder outputs as a single *BMD having multiple roots, much as is done with a shared BDD representation. The complexity of this representation grows linearly with the word size. Observe the relation between the word-level representation (Figure 4, left) 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 circuit output. The relation between these two representations will be discussed more fully in our development of a verification methodology.

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 [6] 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. Algorithms

Our algorithms for constructing and manipulating *BMDs follow the same paradigm as BDD-based Boolean manipulation algorithms. Each function is denoted by a "weighted pointer" of the form $\langle w, v \rangle$, where w denotes a branch weight and v denotes a vertex. A terminal vertex having value w is denoted by the weighted pointer $\langle w, \Lambda \rangle$. In this paper, we give a brief overview of the algorithms. A more detailed description of the algorithms for constructing and manipulating *BMDs is given in [3].

A single graph is maintained in "strong canonical form," i.e., as a multi-rooted DAG with a unique weighted pointer to each function represented. Testing two functions for equivalence then becomes a simple task of comparing the pointer weights and destinations for equality. A vertex is added to the graph only after transformations are applied to ensure a canonical representation. In particular, suppose the program needs to create a vertex for a function f expressed relative to variable x by moments represented by weighted pointers $\langle w_0, v_0 \rangle$ and $\langle w_1, v_1 \rangle$. We would first test if $w_1 = 0$, in which case f is independent of x, and hence we can also represent it as $\langle w_0, v_0 \rangle$. This reduction rule is similar to that used in FDDs and in zero-suppressed BDDs [10]. Otherwise, we would compute $w = \gcd(w_0, w_1)$ and create a vertex v labeled by variable x and having moments represented by pointers $\langle w_0 \div w, v_0 \rangle$ and $\langle w_1 \div w, v_1 \rangle$. Function f is then represented by weighted pointer $\langle w, v \rangle$.

*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. These algorithms are referred to collectively as "Apply" algorithms. Apply algorithms have been implemented for addition, multiplication, and exponentiation of a constant. Boolean operations can be expressed in terms of addition and multiplication and hence do not require additional algorithms.

The apply algorithms proceed by traversing the argument graphs to recursively apply the operation to the subgraphs. To reduce the number of recursive calls, a hash table is maintained keyed by the arguments to previous calls. This enables the program to reuse previous computations. Unlike with BDDs, however, there is no polynomial bound on the number of recursive calls when applying an operation to two *BMDs. Although there are a limited number of possible vertex combinations in the arguments, many different argument weights may be encountered. *BMDs are therefore more susceptible to problems with "exponential blowup" than are BDDs.

6. Verification Methodology

Figure 7 illustrates schematically an approach to circuit verification originally formulated by Lai and Vrudhula [9]. The overall goal is to prove a correspondence between a logic circuit, represented by a vector of Boolean functions $\vec{f}$, and the specification, represented by the word level function F. The Boolean functions can correspond to the outputs of a combinational circuit in terms of the primary inputs, or to the outputs of a sequential circuit operated for a fixed number of steps, in terms of the initial state and the input values at each step. Assume that the circuit inputs (and possibly initial state) are partitioned into vectors of binary signals $\vec{x}^1, \ldots, \vec{x}^k$ (in the figure $k = 2$). For each

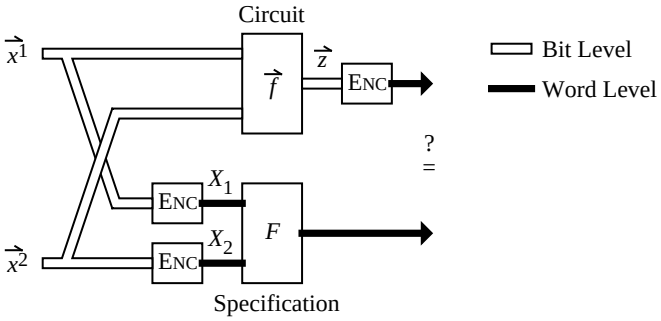

Figure 7: **Formulation of Verification Problem.** The goal of verification is to prove a correspondence between a bit-level circuit and a word-level specification

set of signals $\vec{x}^i$, we are given an encoding function ENC_i describing a word-level interpretation of the signals. An example of such an encoding function would be 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 $\vec{f}(\vec{x}^1, \ldots, \vec{x}^k)$. Typically this circuit is given in the form of a network of logic gates. Furthermore, we are given an encoding function 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:

$$\text{ENC}_o(\vec{f}(\vec{x}^1, \ldots, \vec{x}^k)) \quad = \quad F(\text{ENC}_1(\vec{x}^1), \ldots, \text{ENC}_k(\vec{x}^k)) \qquad (3)$$

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.

6.1. Hierarchical Verification

For circuits that cannot be verified efficiently at the bit level, such as multipliers, we propose a hierarchical verification methodology. The circuit is partitioned into component modules based on its word-level structure. Each component is verified against a word-level specification. Then the word-level functions of the components are composed and compared to the overall circuit specification.

Figure 8 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 must prove that each component implements its specification. Second, we must prove that the composition of the word-level functions matches that of integer multiplication, i.e.,

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

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.

6.2. Component Verification

The component partitioning allows us to efficiently represent both their bit-level and word-level functions. This allows the test of Equation 3 to be implemented directly. As an example, consider the adder circuit having bit-level functions given by the *BMD of Figure 6, 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 by the left-hand *BMD of Figure 4. 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:

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

We would then compare the *BMD for P to the one shown on the left-hand side of Figure 4.

6.3. 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 8 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 8, 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

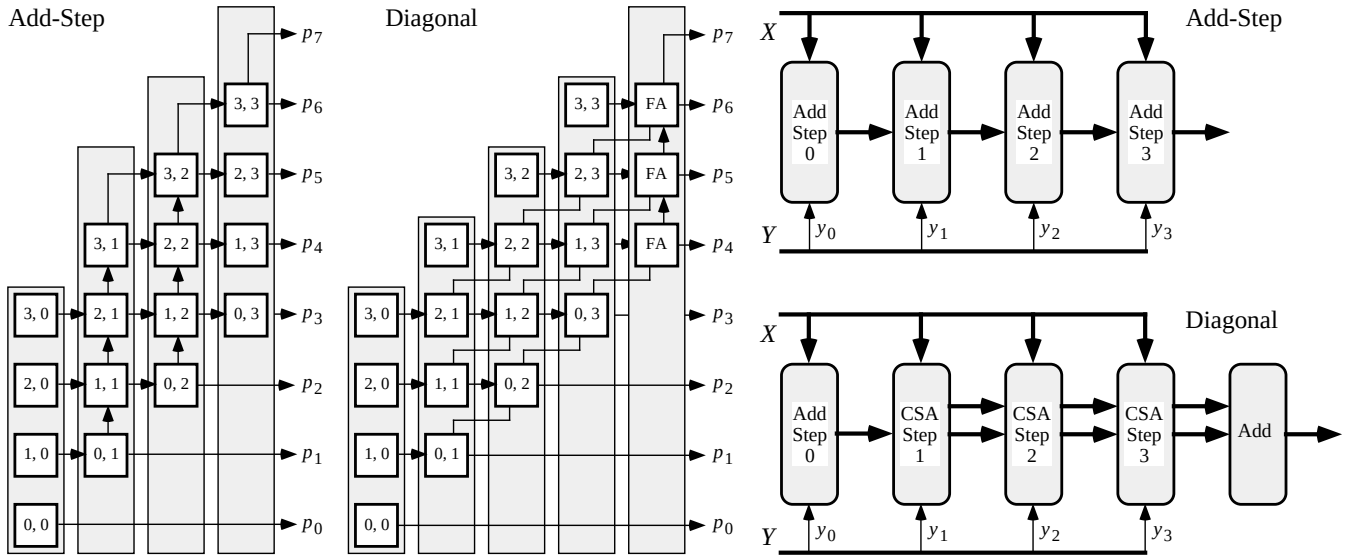

Figure 8: **Multiplier Circuits Different Levels of Abstraction** Each square contains an AND gate and a full adder. The vertical rectangles indicate the word-level partitioning yielding the representations shown on the right.

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:

$$SO_i \quad = \quad SI_i + 2^i \cdot CI_i + 2^i \cdot y_i \cdot X - 2^{i+1} \cdot CO_i \qquad (4)$$

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 $\vec{c}^i$. 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 4 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 4. 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 1: **Verification Results for Combinational Multipliers.** Results are shown for three different word sizes.

6.4. Experimental Results

Table 1 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. 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. Ochi *et al* [12] have successfully built the OBDDs for a 15-bit multiplier, requiring over 12 million vertices. Increasing the word size by one bit causes the number of vertices to increase by a factor of approximately 2.7, and hence even more powerful computers will not be able to get much beyond this point.

Burch [4] 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* [7] 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. To our knowledge, this is the best result on verifying this circuit at a bit level. 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.

7. 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. We are currently developing a a comprehensive verification system based on our hierarchical methodology. We have devised a simple notation "ACV" (for Arithmetic Circuit Verifier) that allows the user to describe circuits hierarchically. For each module in the hierarchy, one gives a structural definition as an interconnection of logic gates and other modules. One can also declare encodings of the inputs and outputs, as well as give a specification in terms of arithmetic and Boolean operations. We are implementing a program that will accept descriptions in this language and either verify their correctness or supply a "counterexample," i.e., an input pattern for some module causing a mismatch between the specified and actual behavior.

Our method shows some promise for verifying floating point hardware, although difficult obstacles must be overcome. Using a version that represents rational-valued functions, 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.

Some of the applications proposed for EVBDDs and MTBDDs may work well with *BMDs. Among these are matrix operations and spectral transforms. Applications requiring efficient equation solving, such as integer linear programming, on the other hand, are probably not good candidates.

All of the applications described so far assume the underlying system is described in terms of binary-valued variables. In fact, *BMDs provide a canonical representation for multivariate linear functions. This could prove useful for applications in symbolic algebra. Independent of this work, Minato has devised a canonical representation for multi-variate *polynomial* functions [11]. Although he describes his approach as an extension of zero-suppressed BDDs [10], the representation is closely related to *BMDs. For a function variable x, he introduces vertex labels representing $x, x^2, x^4, x^8, \ldots, x^{2^k}$. A term x^m with $m < 2^{k+1}$ is then represented according to the binary representation of m. With this representation, the connection to symbolic algebra becomes even more intriguing.

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] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[3] R. E. Bryant, and Y.-A. Chen, "Verification of arithmetic functions with binary moment diagrams," Technical Report CMU-CS-94-160, Carnegie Mellon University, May, 1994.

[4] J. R. Burch, "Using BDDs to verify multipliers," *28th Design Automation Conference*, June, 1991, pp. 408–412.

[5] 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 ACM/IEEE Design Automation Conference*, Dallas, TX, June, 1993, pp. 54–60.

[6] R. Enders, "Note on the complexity of binary moment diagram representations," unpublished paper, Siemens AG, Munich Germany, 1994.

[7] J. Jain, J. Bitner, M. Abadir, J. A. Abraham, and D. S. Fussell, "Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions," submitted for publication, 1994.

[8] U. Kebschull, E. Schubert, and W. Rosentiel, "Multilevel logic based on functional decision diagrams," *European Design Automation Conference*, 1992, pp. 43–47.

[9] 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.

[10] S.-i. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *30th Design Automation Conference*, June, 1993, pp. 272–277.

[11] S.-i. Minato, "Implicit manipulation of polynomials using zero-suppressed BDDs," unpublished manuscript, 1994.

[12] H. Ochi, K. Yasuoka, and S. Yajima, "Breadth-first manipulation of very large binary-decision diagrams," *International Conference on Computer-Aided Design*, November, 1993, pp. 48–55.