

An Efficient Graph Representation for Arithmetic Circuit Verification

Yirng-An Chen, Randal E. Bryant, *Fellow, IEEE*

Abstract— In this paper, we propose a new data structure, called **Multiplicative Power Hybrid Decision Diagrams (*PHDDs)**, to provide a compact representation for functions that map Boolean vectors into integer or floating-point values. The size of the graph to represent the IEEE floating-point encoding is linear with the word size. The complexity of floating-point multiplication grows linearly with the word size. The complexity of floating-point addition grows exponentially with the size of the exponent part, but linearly with the size of the mantissa part. We applied *PHDDs to verify integer multipliers and floating-point multipliers before the rounding stage, based on a hierarchical verification approach. For integer multipliers, our results are at least 6 times faster than *BMDs. Previous attempts at verifying floating-point multipliers required manual intervention, but we verified floating-point multipliers before the rounding stage automatically.

Keywords— BDD, Binary Moment Diagram, BMD, Hybrid Decision Diagram, HDD, Multiplicative Power Hybrid Decision Diagram, *PHDD, Formal Verification

I. INTRODUCTION

The floating-point (FP) division bug [14] in Intel's Pentium processor and the overflow flag erratum of the FIST instruction (FP to integer conversion) [17] in Intel's Pentium Pro and Pentium II processors have demonstrated the importance and the difficulty of verifying FP arithmetic circuits and the high cost of an arithmetic bug. Formal verification or exhaustive simulation can be used to ensure the correctness of arithmetic circuits. However, it is impossible to perform exhaustive simulations for arithmetic circuits with large word size.

Formal verification techniques such as theorem proving and word-level decision diagrams have been used to verify arithmetic circuits. Most of the IEEE FP standard has been formalized by Carreño and Miner [5] for the HOL and PVS theorem provers. In [21], Moore, *et al.* applied a mechanical theorem prover, ACL2 [19], to verify the division microcode program used on the AMD5_K86 microprocessor. Similar work has been done by Clarke, *et al.* [11] to verify the SRT division algorithm used in many modern microprocessors. To verify arithmetic circuits, theorem provers require users to guide the proof which is structured as series of lemmas describing the effect of circuit modules and their interactions [2]. Thus, the verification process is not only very tedious but also implementation-dependent.

Word-level decision diagram is another approach to verify arithmetic circuits. Binary Moment Diagrams

(BMDs) [4] have proved successful for representing and manipulating functions mapping Boolean vectors to integer values symbolically. They have been used in the verification of arithmetic circuits [7]. Clarke, *et al.* [10] extended BMDs to a form they call Hybrid Decision Diagrams (HDDs), where a function may be decomposed with respect to each variable in one of six ways, but without edge weights. Drechsler, *et al.* [15] extended Multiplicative BMDs (*BMDs) to a form called K*BMDs, where a function may be decomposed with respect to each variable in one of three ways, and with both additive and multiplicative edge weights. None of these diagrams can represent functions which map Boolean vectors to floating-point values, unless rational numbers are introduced into the representation [3]. But using rational numbers in the representation requires more memory space to store the numerator and denominator separately, and more computation to extract the rational numbers.

After the famous Pentium division bug [14], Intel researchers applied word-level SMV [12] with Hybrid Decision Diagrams (HDDs) [10] to verify the functionality of the FP unit in one of Intel's processors [9]. Verification of floating-point arithmetic circuits using any of these three diagrams requires the circuits to be divided into several sub-circuits for which specifications can be expressed in terms of integer functions and their operations [7], [9], [12]. The correctness of the overall circuit must be proved by users from the specifications of the verified sub-circuits. For instance, the floating-point multiplier was divided into the circuits for the mantissa multiplication, the exponent addition, and the rounding in [9]. The verification of these three sub-circuits was performed automatically by word-level SMV [12], but the correctness of the entire multiplier must be proved by users from the verified specifications of these three sub-circuits. To avoid partitioning floating-point arithmetic circuits for verification, it is necessary to have decision diagrams that represent and manipulate floating-point functions efficiently.

In this paper, we propose a new representation, called **Multiplicative Power Hybrid Decision Diagrams (*PHDDs)**, which improves on previous diagrams in representing floating-point functions. *PHDDs can represent functions having Boolean variables as arguments and floating-point values as results. This structure is similar to that of HDDs [10], except that they are based on powers-of-2 edge weights and complement edges for negation. We show that the size of floating-point multiplication grows linearly with the word size. For floating-point addition, we show that the complexity grows linearly with the mantissa size, but exponentially with the exponent size. It is still

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-96-C-0071.

Y.-A. is with Novas Software Inc., San Jose, CA 95110, USA.

R. E. Bryant is with Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

practical for formats up to IEEE double precision.

Based on a hierarchical verification methodology [4], [7], we have applied *PHDDs to verify different sizes and types of integer multipliers. Compared with *BMDs, *PHDDs are consistently six times faster and use less memory. We have also applied *PHDDs to verify different sizes and types of floating-point multipliers before the rounding stage, which have never before been verified symbolically and automatically. Our results show that the verification of floating-point multipliers requires minimal effort beyond integer multipliers. Our next step is to look into the rounding stage and entire floating-point adders. Earlier results using HDDs [9] show that the rounding stage itself can be handled.

The remainder of the paper is organized as follows. Section 2 briefly discusses previous work on word-level decision diagrams: *BMDs, HDDs and K*BMDs. The *PHDD data structure is presented in section 3. Section 4 shows *PHDDs for several numeric functions including integers and floating-point numbers. Section 5 analyzes the *PHDD complexities of floating-point multiplication and floating-point addition. Section 6 compares the differences among *PHDD, *BMD, HDD and K*BMD. The performance results are shown in Section 7. Finally, Section 8 contains our conclusions and possible future work.

II. PREVIOUS WORK: BMDs, *BMDs, HDDs AND K*BMDs

For expressing functions Boolean variables into integer values, BMDs[4] use the moment decomposition of a function:

$$\begin{aligned} f &= (1-x) \cdot f_{\bar{x}} + x \cdot f_x \\ &= f_{\bar{x}} + x \cdot (f_x - f_{\bar{x}}) \\ &= f_{\bar{x}} + x \cdot f_{\delta x} \end{aligned} \quad (1)$$

where \cdot , $+$ and $-$ denote multiplication, addition and subtraction, respectively. Term f_x ($f_{\bar{x}}$) denotes the positive (negative) cofactor of f with respect to variable x , i.e., the function resulting when constant 1 (0) is substituted for x . By rearranging the terms, we obtained the third line of Equation 1. Here, $f_{\delta x} = f_x - f_{\bar{x}}$ 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 . The negative cofactor $f_{\bar{x}}$ will be termed the constant moment, i.e., it denotes the portion of function f that remains constant with respect to x . This decomposition is also called positive Davio in K*BMDs [15]. 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.

An extension of BMDs is to incorporate "weight values" on the edges, yielding a representation called Multiplicative BMDs (*BMDs) [4]. The edge function $f_e(m, f)$ is obtained from the function f of the node through multiplication of integer value m (i.e., $f_e(m, f) = w \times f$). The

edge weights are extracted by the greatest common divisor (GCD) of edge weights. With *BMDs, word-level functions such as $X + Y$, $X - Y$, $X \times Y$ and 2^X all have linear-sized representations.

Clarke, *et al.* [10] extended BMDs to a form they call Hybrid Decision Diagrams (HDDs), where a function may be decomposed with respect to each variable in one of six decomposition types. In our experience with HDDs, we found that three of their six decomposition types are useful in the verification of arithmetic circuits. These three decomposition types are Shannon, Positive Davio, and Negative Davio. Therefore, Equation 1 is generalized to the following three equations according the variable's decomposition type:

$$f = \begin{cases} (1-x) \cdot f_{\bar{x}} + x \cdot f_x & (\text{Shannon}) \\ f_{\bar{x}} + x \cdot f_{\delta x} & (\text{Positive Davio}) \\ f_x + (1-x) \cdot f_{\delta \bar{x}} & (\text{Negative Davio}) \end{cases} \quad (2)$$

Here, $f_{\delta \bar{x}} = f_{\bar{x}} - f_x$ is the partial derivative of f with respect to \bar{x} . The BMD representation is a subset of HDDs. In other words, the HDD graph is the same as the BMD graph, if all of the variables use positive Davio decomposition.

Adding both additive and multiplicative weights into HDDs yields another representation called Kronecker *BMDs (K*BMDs) [15]. In this representation, the edge function $f_e(a, m, f)$ is obtained from the function f of the node through multiplication and addition of integer values m and a (i.e., $f_e(a, m, f) = a + w \times f$). In this representation, variables can only use one of Shannon, positive Davio and negative Davio decompositions. Both HDDs and K*BMDs are the superset of BMDs and *BMDs, respectively.

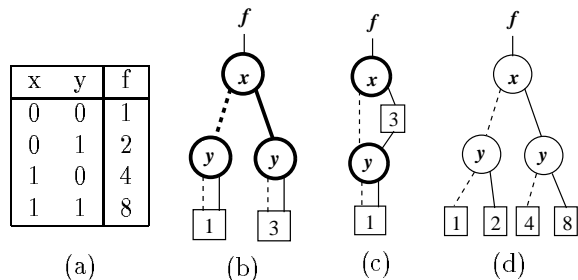


Figure 1. An integer function with Boolean variables, $f = 1 + y + 3x + 3xy$, is represented by (a) Truth table, (b) BMDs, (c) *BMDs, (d) HDDs with Shannon decompositions. The dashed-edges are 0-branches and the solid-edges are the 1-branches. The variables with Shannon and positive Davio decomposition types are drawn in vertices with thin and thick lines, respectively.

As an example, Figure 1 shows an integer function f with Boolean variables x and y represented by a truth table, BMDs, *BMDs, and HDDs with Shannon decompositions (also called MTBDD [13]). In our drawing, the variables with Shannon and positive Davio decomposition types are drawn in vertices with thin and thick lines, respectively. The dashed (solid) line from a vertex with variable x points to the vertex represented function $f_{\bar{x}}$, f_x , and

f_x (f_x , $f_{\delta x}$ and $f_{\delta \bar{x}}$) for Shannon, positive Davio and negative Davio decompositions, respectively. Figure 1.b shows the BMD representation. To construct this graph, we apply Equation 1 to function f recursively. First, with respect to variable x , we can get $f_{\bar{x}} = 1 + y$, represented as the graph of the dashed-edge of vertex x , and $f_{\delta x} = 3 + 3y$, represented by the solid branch of vertex x . Observe that $f_{\delta x}$ can be expressed by $3 \times f_{\bar{x}}$. By extracting the factor 3 from $f_{\delta x}$, the graph became Figure 1.c. This graph is called a Multiplicative BMD (*BMD) which extracts the greatest common divisor (GCD) from both branches. The edge weights combine multiplicatively. The HDD with Shannon decompositions can be constructed from the truth table. The dashed branch of vertex x is constructed from the first two entries of the table, and the solid branch of vertex x is constructed from the last two entries of the table.

Observe that if variables x and y are viewed as bits forming 2-bit binary number, $X=y+2x$, then the function f can be rewritten as $f = 2^{(y+2x)} = 2^X$. Observe that HDDs with Shannon decompositions and BMDs grow exponentially for this type of functions. *BMDs can represent them efficiently, due to the edge weights. However, *BMDs, HDDs and K*BMDs cannot represent the functions as $f = 2^{X-B}$, where B is a constant, because they can only represent integer functions.

III. THE *PHDD DATA STRUCTURE

In this section, we introduce a new data structure, Multiplicative Power Hybrid Decision Diagrams (*PHDDs) [8], to represent functions that map Boolean vectors to integer or floating-point values. This structure is similar to that of HDDs, except that they use power-of-2 edge weights and negation edges. The power-of-2 edge weights allow us to represent and manipulate functions mapping Boolean vectors to floating-point values. Negation edges can further reduce graph size by as much as a factor of 2. We assume that there is a total ordering of the variables such that the variables are tested according to this ordering along any path from the root to a leaf. Each variable is associated with its own decomposition type and all nodes of that variable use the corresponding decomposition.

A. Edge Weights

*PHDDs use three of HDD's six decompositions as expressed in Equation 2. Similar to *BMDs, we adapt the concept of edge weights to *PHDDs. Unlike *BMD edge weights, we restrict our edge weights to be powers of a constant c . Thus, Equation 2 is rewritten as:

$$\langle w, f \rangle = \begin{cases} c^w \cdot (((1-x) \cdot f_{\bar{x}} + x \cdot f_x) & (\text{Shannon}) \\ c^w \cdot (f_{\bar{x}} + x \cdot f_{\delta x}) & (\text{Positive Davio}) \\ c^w \cdot (f_x + (1-x) \cdot f_{\delta \bar{x}}) & (\text{Negative Davio}) \end{cases} \quad (3)$$

where $\langle w, f \rangle$ denotes $c^w \times f$. In general, the constant c can be any positive integer. Since the base value of the exponent part of the IEEE floating-point format is 2, we will consider only $c = 2$ for the remainder of the paper. Observe that w can be negative, i.e., we can represent rational

numbers. The power edge weights enable us to represent functions mapping Boolean variables to floating-point values without using rational numbers in our representation.

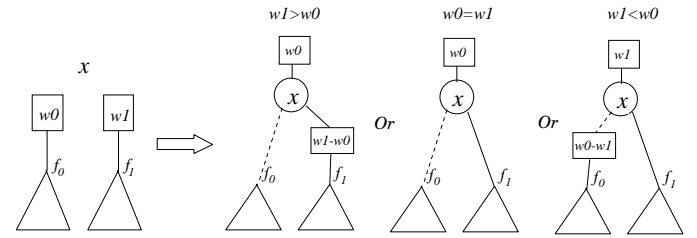


Figure 2. Normalizing the edge weights.

In addition to the HDD reduction rules [10], we apply several edge weight manipulating rules to maintain the canonical form of the resulting graph. Let $w0$ and $w1$ denote the weights at branch 0 and 1 respectively, and f_0 and f_1 denotes the functions represented by branch 0 and 1. To normalize the edge weights, we chose to extract the minimum of the edge weight $w0$ and $w1$. This is a much simpler computation than the GCD of integer *BMDs or the reciprocal of rational *BMDs [3]. Figure 2 illustrates the manipulation of edge weights to maintain a canonical form. The first step is to extract the minimum of $w0$ and $w1$. Then, the new edge weights are adjusted by subtracting the minimum from $w0$ and $w1$ respectively. A node is created with the index of the variable, the new edge weights, and pointers to f_0 and f_1 . Based on the relation of $w0$ and $w1$, the resulting graph is one of three graphs in Figure 2. Note that at least one branch has zero weight. In addition, the manipulation rule of the edge weight is the same for all of the three decomposition types. In other words, the representation is normalized if and only if the following holds:

- The leaf nodes can only have odd integers or 0.
- At most one branch has non-zero weight.
- The edge weights are greater than or equal to 0, except the top one.

B. Negation Edge

Negation edges are commonly used in BDDs [1] and KFDDs [16], but not in *BMDs, HDDs and K*BMDs. Since our edge weights extract powers-of-2 which are always positive, negation edges are added to *PHDDs to increase sharing among the diagrams. In *PHDDs, the negation edge of function f represents the negation of f . Note that $-f$ is different from \bar{f} for Boolean functions.

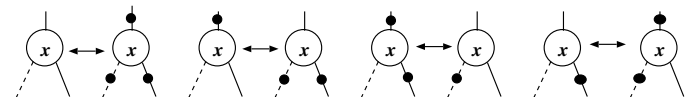


Figure 3. Rules for negation edges

Negation edges allow greater sharing and make negation a constant computation. In *PHDD data structure, we use the low order bit of the pointers to denote negation, as

is done with the complement edge of BDDs. To maintain a canonical form, we must constrain the use of negation edges. Unlike KFDDs [16], where Shannon decompositions use a different method from positive and negative Davio decompositions, *PHDDs use the same method for manipulating the negation edge for all three decomposition types. *PHDDs must follow these rules: the zero edge of every node must be a regular edge, the negation of leaf 0 is still leaf 0, and leaves must be nonnegative. Figure 3 illustrates the rules for negation edges. These four pairs of functions are functionally equivalent. In our implementation, we always create nodes for the left side of the pair. In other words, the 0-branch is always positive. These guarantee the canonical form for *PHDDs.

IV. REPRESENTATION OF NUMERIC FUNCTIONS

*PHDDs can effectively represent numeric functions that map Boolean vectors into integer or floating-point values. We first show that *PHDDs can represent integer functions with comparable sizes to *BMDs. Then, we show the *PHDD representation for floating-point numbers.

A. Representation of Integers

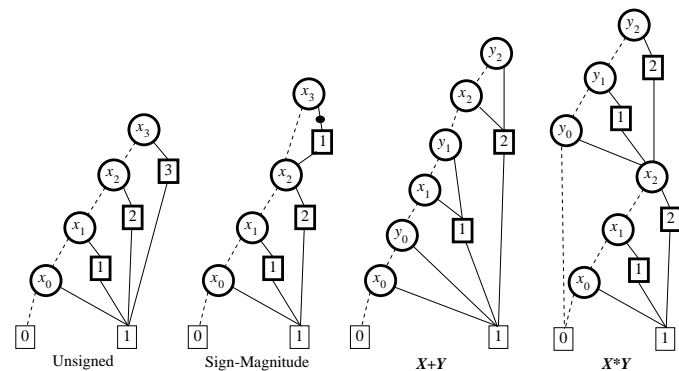


Figure 4. *PHDD Representations of Integers and Integer operations. Each variable uses positive Davio decomposition. The graphs grow linearly with word size.

*PHDDs, similar to *BMDs, can provide a concise representation of functions which map Boolean vectors to integer values. Let \vec{x} represent a vector of Boolean variables: x_{n-1}, \dots, x_1, x_0 . These variables can be considered to represent an integer X according to some encoding, e.g., unsigned binary or two's complement. Figure 4 illustrates the *PHDD representations of several common encodings for integers. In our drawing of *PHDDs, we indicate the edge weight and leaf node in square boxes with thick and thin lines, respectively. Edge weight i represents 2^i and unlabeled edges have weight 0 (2^0). An unsigned number is encoded as a sum of weighted bits. The *PHDD representation has a simple linear structure where the leaf values are formed by the corresponding edge weight and leaf 1 or 0. For representing signed numbers, we assume x_{n-1} is the sign bit. The two's complement encoding has a *PHDD representation similar to that of unsigned integers, but with bit x_{n-1} having weight -2^{n-1} represented by the edge weight $n-1$

and the negation edge. Sign-magnitude integers also have *PHDD 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 represented by edge weight 1 and the negation edge. In evaluating the function for $x_{n-1} = 1$, we would add these two moments, effectively scaling the number by -1 . Note that it is more logical to use Shannon decomposition for the sign bit.

Figure 4 also illustrates the *PHDD representations of several common arithmetic operations on integer data. Observe that the sizes of the graphs grow only linearly with the word size n . Integer addition can be viewed as summing a set of weighted bits, where bits x_i and y_i both have weight 2^i represented by edge weight i . Integer multiplication can be viewed as summing a set of partial products of the form $y_i 2^i X$. In summary, while representing the integer functions, *PHDDs with positive Davio decompositions usually will get the most compact representation among these three decompositions.

B. Representation of Floating Point Numbers

Let us consider the representation of floating-point numbers by IEEE standard 754. For example, the double-precision numbers are stored in 64 bits: 1 bit for the sign (S_x), 11 bits for the exponent (EX), and 52 bits for the mantissa (X). The exponent is a signed number represented with a bias (B) 1023. The mantissa represents a number less than 1. Based on the value of the exponent, the IEEE floating-point format can be divided into four cases:

$$FX = \begin{cases} (-1)^{S_x} \times 1.X \times 2^{EX-B} & \text{If } 0 < EX < \text{All } 1 \text{ (normal)} \\ (-1)^{S_x} \times 0.X \times 2^{1-B} & \text{If } EX = 0 \text{ (denormal)} \\ NaN & \text{If } EX = \text{All } 1 \text{ \& } X \neq 0 \\ (-1)^{S_x} \times \infty & \text{If } EX = \text{All } 1 \text{ \& } X = 0 \end{cases}$$

Currently, *PHDDs cannot handle infinity and NaN (not a number) cases in the floating-point representation. Instead, assume they are normal numbers.

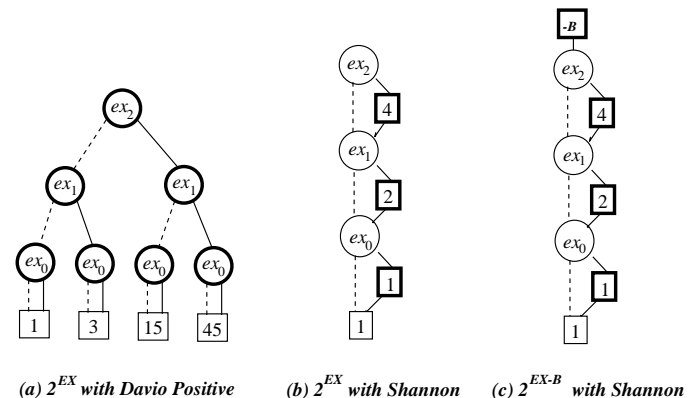


Figure 5. *PHDD Representations of 2^{EX} and 2^{EX-B} . The graph grows linearly in the word size with Shannon, but grows exponentially with positive Davio.

Figure 5 shows *PHDD representations for 2^{EX} and 2^{EX-B} using different decompositions. To represent function c^{EX} (in this case $c = 2$), *PHDDs express the function

as a product of factors of the form $c^{2^i e x_i} = (c^{2^i})^{e x_i}$. In the graph with Shannon decompositions, a vertex labeled by variable $e x_i$ has outgoing edges with weights 0 and c^{2^i} both leading to a common vertex denoting the product of the remaining factors. But in the graph with positive Davio decompositions, there is no sharing except for the vertices on the layer just above the leaf nodes. Observe that the size of *PHDDs with positive Davio decomposition grows exponentially in the word size while the size of *PHDDs with Shannon grows linearly. Interestingly, *BMDs have a linear growth for this type of function, while *PHDDs with positive Davio decompositions grow exponentially. To represent floating-point functions symbolically, it is necessary to represent 2^{EX-B} efficiently, where B is a constant. *PHDD can represent this type of functions, but *BMDs, HDDs and K*BMDs cannot represent them without using rational numbers. HDDs cannot represent either 2^{EX} or 2^{EX-B} efficiently, since they do not have edge weights. *BMDs can represent 2^{EX} efficiently [4], but not 2^{EX-B} which can be interpreted as 2^{EX} divided by 2^B . K*BMDs have the same problem as *BMDs to represent 2^{EX-B} . However, the edge weights in *PHDDs can represent function 2^{EX-B} efficiently by simply adding a edge weight $-B$ on top of the *PHDDs of 2^{EX} as shown in Figure 5.c.

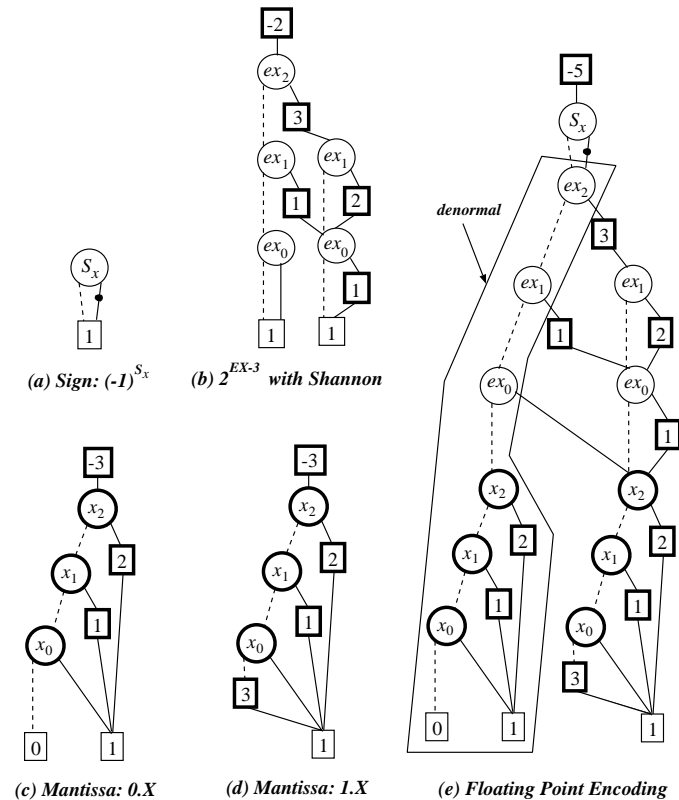


Figure 6. Representations of floating-point encodings.

Figure 6 shows the *PHDD representations for the floating-point encoding, where EX has 3 bits, X has 4 bits and the bias B is 3. The sign S_x and $e\bar{x}$ variables use Shannon decomposition, while variables \bar{x} use positive Davio. Figure 6.a shows the *PHDD representation for the

sign bit $(-1)^{S_x}$. When S_x is 0, the value is 1; otherwise, the value is -1 represented by the negation edge and leaf node 1. Figure 6.b shows the *PHDD representation for the exponent part 2^{EX-3} . The graph is more complicated than Figure 5.c, because, in the floating-point encoding, when $EX = 0$, the value of the exponent is $1 - B$, instead of $-B$. Observe that each exponent variable, except the top variable $e x_2$, has two nodes: one to represent the denormal number case and another to represent normal number case. Figure 6.c shows the representation for the mantissa part $0.X$ obtained by dividing X by 2^{-3} . Again, the division by powers of 2 is just adding the edge weight on top of the original graph. Figure 6.d shows the representation for the mantissa part $1.X$ which is the sum of $0.X$ and 1. The weight (2^{-3}) of the least significant bit is extracted to the top and the leading bit 1 is represented by the path with all variables set to 0. Finally, Figure 6.e shows the *PHDD representation for the complete floating-point encoding. Observe that negation edges reduce the graph size by half. The outlined region in the figure denotes the representation for denormal numbers. The rest of the graph represents normal numbers. Assume the exponent is n bits and the mantissa is m bits. Note that the edge weights are encoded into the node structure in our implementation, but the top edge weight requires an extra node. It can be shown that the total number of *PHDD nodes for the floating point encoding is $2(n + m) + 3$. Therefore, the size of the graph grows linearly with word size. In our experience, it is best to use Shannon decompositions for the sign and exponent bits, and positive Davio decompositions for the mantissa bits.

V. COMPLEXITY OF FLOATING POINT MULTIPLICATION AND ADDITION

In this section, we first present the complexity of floating-point multiplication based on *PHDDs. Here, we show the representations of the operation result before rounding. In other words, the resulting *PHDDs represent the precise results of floating-point multiplication. The size of the resulting graph grows linearly with the word size. Then, we discuss the complexity of floating-point addition based on *PHDDs. Again, we show the representation of the operation result before rounding. The size of the resulting graph grows exponentially with the size of the exponent part and grows linearly with the the size of the mantissa part.

A. Floating Point Multiplication

Let $F_X = (-1)^{S_x} \times v_x \cdot X \times 2^{EX-B}$ and $F_Y = (-1)^{S_y} \times v_y \cdot Y \times 2^{EY-B}$, where v_x (v_y) is 0 if EX (EY) = 0, otherwise, v_x (v_y) is 1. EX and EY are n bits, and X and Y are m bits. Let the variable ordering be the sign variables, followed by the exponent variables and then the mantissa variables. Based on the values of EX and EY , $F_X \times F_Y$ can be written as:

$$\begin{aligned}
 F_X \times F_Y &= (-1)^{S_x \oplus S_y} \times (v_x \cdot X \times 2^{EX-B}) \times (v_y \cdot Y \times 2^{EY-B}) \\
 &= (-1)^{S_x \oplus S_y} \times 2^{-2B} \times \begin{cases} 2^1 \times (0.X \times v_y \cdot Y) \times 2^{EY} & \text{Case 0} \\ 2^{EX} \times (1.X \times v_y \cdot Y) \times 2^{EY} & \text{Case 1} \end{cases}
 \end{aligned}$$

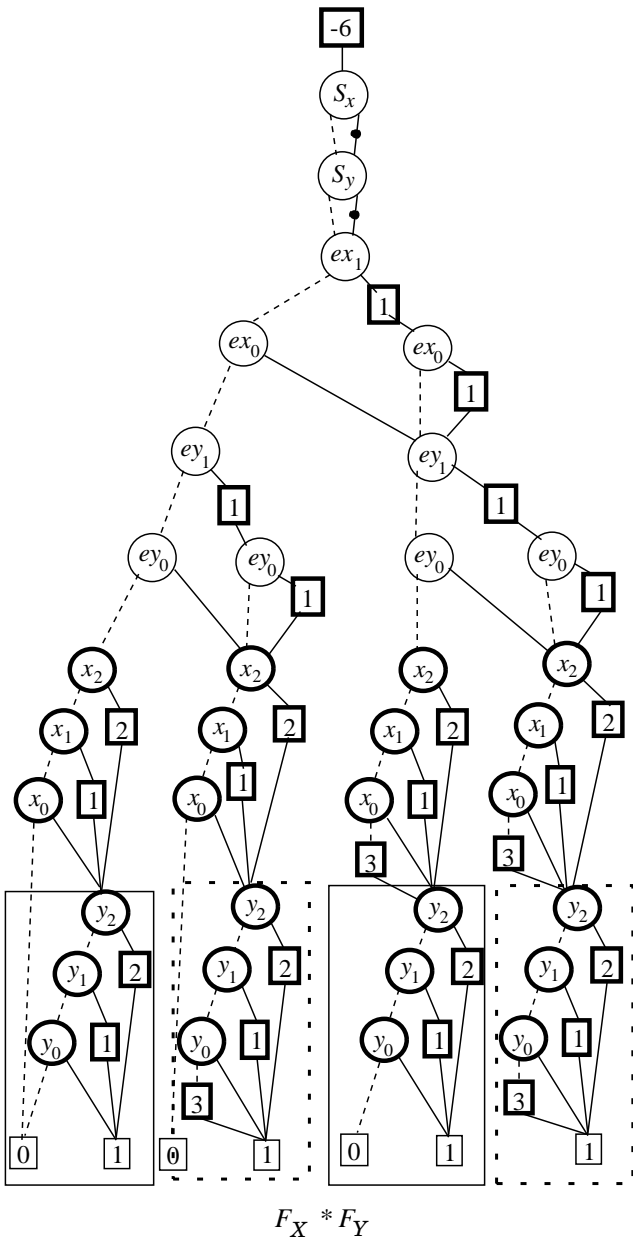


Figure 7. Representation of floating-point multiplication.

, where Case 0 represents $EX=0$, and Case 1 represents $EX \neq 0$.

Figure 7 illustrates the *PHDD representation for floating-point multiplication. Observe that two negation edges reduce the graph size to one half of the original size. When $EX = 0$, the subgraph represents the function $0.X \times v_y.Y \times 2^{EY}$. When $EX \neq 0$, the subgraph represents the function $1.X \times v_y.Y \times 2^{EY}$. The size of exponent nodes grows linearly with the word size of the exponent part. The lower part of the resulting graph shows four mantissa products (from left to right): $X \times Y$, $X \times (2^3 + Y)$, $(2^3 + X) \times Y$, $(2^3 + X) \times (2^3 + Y)$. The first and third mantissa products share the common sub-function Y shown by the solid rectangles in Figure 7. The second and fourth products share the common sub-function $2^3 + Y$ shown by the dashed rectangles in Figure 7.

What is the complexity of floating-point multiplication for *PHDDs? In the following theorem, we show that the size of the resulting graph of floating-point multiplication is $6(n+m)+3$ with the variable ordering given in Figure 7, where n and m are the number of bits in the exponent and mantissa parts. Thus, the size of the resulting graph grows linearly with the word size for floating-point multiplication.

Theorem 1: Given the ordering $S_x, S_y, ex_0, ey_0, \dots, ex_{n-1}, ey_{n-1}, x_{m-1}, \dots, x_0, y_{m-1}, \dots, y_0$, as shown in Figure 7, the size of the resulting graph of floating-point multiplication is $6(n+m)+3$, where n and m are the number of bits in the exponent and mantissa parts.

Proof: From Equation 4 and Figure 7, we know that there is no sharing in the sub-graphs for $EX = 0$ and $EX \neq 0$. For $EX = 0$, the size of the sub-graph except leaf nodes is the sum of the nodes for the exponent of F_Y ($2n-1$ nodes), the nodes for the mantissa of F_X ($2m$ nodes), and the nodes for the mantissa of F_Y (m nodes). Similarly, for $EX \neq 0$, the size of the sub-graph except leaf nodes is also $2n+3m-1$. The size for the exponent part of F_X is $2n-1$. The number of nodes for the sign bits and top level edge weight is 3, and the number of leaf nodes is 2. Therefore, the size of the resulting graph for floating-point multiplication is $6(n+m)+3$. \square

B. Floating Point Addition

According to the sign bits of two operands, floating-point addition, $F_X + F_Y$, can be divided into two cases. When $S_x \oplus S_y$ is equal to 0, the floating-point addition must be performed as “true addition”, shown as Equation 4.

$$F_X + F_Y = (-1)^{S_x} \times (2^{EX-B} \times v_x.X + v_y.Y \times 2^{EY-B}) \quad (4)$$

When $S_x \oplus S_y$ is equal to 1 (i.e., they have different sign), floating-point addition must be performed as “true subtraction”, shown as the following equation.

$$F_X + F_Y = (-1)^{S_x} \times (2^{EX-B} \times v_x.X - v_y.Y \times 2^{EY-B}) \quad (5)$$

There are distinct mantissa sums among true addition and true subtraction, because one performs addition and another performs subtraction. In the following theorem, we show that the number of distinct mantissa sums is $2^{n+3} - 10$, where n is the number of bits in the exponent part.

Theorem 2: For floating-point addition $F_X + F_Y$, the number of distinct mantissa sums is $2^{n+3} - 10$, where n is the number of bits in the exponent part.

Proof: Since there is no sharing of distinct mantissa sums among true addition and true subtraction, the number of distinct mantissa sums is the total of the number of distinct mantissa sums of true addition and true subtraction.

Let us consider the true addition operation first. Based on the relation of EX and EY , Equation 4 can be rewritten as the following:

$$F_X + F_Y = (-1)^{S_x} \times \begin{cases} 2^{1-B} \times \{0.X + 0.Y\} & \text{Case 0} \\ 2^{1-B} \times \{0.X + 1.Y \times 2^{EY-1}\} & \text{Case 1} \\ 2^{1-B} \times \{2^{EX-1} \times 1.X + 0.Y\} & \text{Case 2} \\ 2^{EX-B} \times \{1.X + 1.Y\} & \text{Case 3} \\ 2^{EX-B} \times \{1.X + 1.Y \times 2^{EY-EX}\} & \text{Case 4} \\ 2^{EY-B} \times \{2^{EX-EY} \times 1.X + 1.Y\} & \text{Case 5} \end{cases} \quad (6)$$

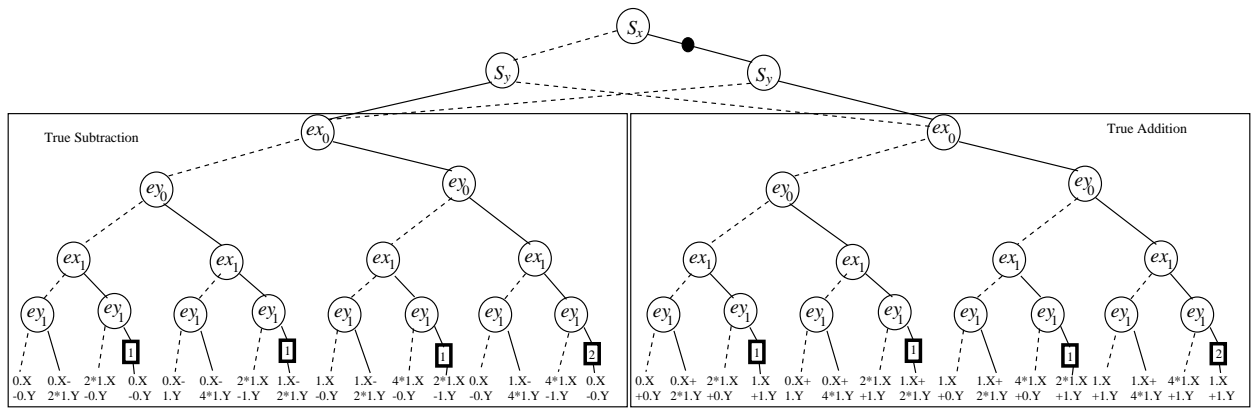


Figure 8. **Representation of floating-point addition.** For simplicity, the graph only shows sign bits, exponent bits and the possible combinations of mantissa sums.

For Case 0, $EX=0$ and $EY=0$, the number of distinct mantissa sums is only 1. For Case 1, $EX=0$ and $EY > 0$, the number of distinct mantissa sums is the same as the number of possible values of EY except 0, which is $2^n - 1$. Similarly, for Case 2, $EX > 0$ and $EY=0$, the number of distinct mantissa sums is also $2^n - 2$, but $0.X+1.Y$ has the same representation as $1.X+0.Y$ in case 1. For Case 3, $EX > 0$ and $EX=EY$, the number of distinct mantissa sums is only 1. For Case 4, $EX > 0$ and $EX < EY$, the number of distinct mantissa sums is the same as the number of possible values of $EY - EX$. Since both EX and EY can not be 0, the number of possible values of $EY - EX$ is $2^n - 2$. Therefore, the number of distinct mantissa sums is $2^n - 2$. Similarly, for Case 5, $EY > 0$ and $EY < EX$, the number of distinct mantissa sums is also $2^n - 2$. Therefore the total number of distinct mantissa sums for the true addition is $2^{n+2} - 5$.

Similarly, Equation 5 can be rewritten as the following equation:

$$F_X + F_Y = (-1)^{S_x} \times \begin{cases} 2^{1-B} \times \{0.X - 0.Y\} & \text{Case 0} \\ 2^{1-B} \times \{0.X - 1.Y \times 2^{EY-1}\} & \text{Case 1} \\ 2^{1-B} \times \{2^{EX-1} \times 1.X - 0.Y\} & \text{Case 2} \\ 2^{EX-B} \times \{1.X - 1.Y\} & \text{Case 3} \\ 2^{EX-B} \times \{1.X - 1.Y \times 2^{EY-EX}\} & \text{Case 4} \\ 2^{EY-B} \times \{2^{EX-EY} \times 1.X - 1.Y\} & \text{Case 5} \end{cases} \quad (7)$$

For Case 5 ($EY > 0$ and $EY < EX$) and Case 4 ($EX > 0$ and $EX < EY$) the numbers of distinct mantissa sums are the same as those in the corresponding cases of true addition. For Case 3 ($EX > 0$ and $EX=EY$), the mantissa sum $1.X - 1.Y$ is the same as $0.X - 0.Y$ in Case 0 ($EX=EY=0$). For both Case 1 ($EX=0$ and $EY > 0$) and Case 2 ($EX > 0$ and $EY=0$), the number of distinct mantissa sum is $2^n - 1$. Therefore, the number of distinct mantissa sums for the true subtraction is also $2^{n+2} - 5$. Thus, the total number of distinct mantissa sums is $2^{n+3} - 10$. \square

Figure 8 illustrates the *PHDD representation of floating-point addition with two exponent bits for each floating-point operand. Observe that the negation edge reduces the graph size by half. There is no sharing among the sub-graphs for true addition and true subtraction. In true subtraction, $1.X - 1.Y$ has the same representation as

$0.X - 0.Y$. Therefore, all $1.X - 1.Y$ entries are replaced by $0.X - 0.Y$. Since the number of distinct mantissa sums grows exponentially with the number of exponent bits, it can be shown that the total number of nodes grows exponentially with the size of exponent bits and grows linearly with the size of the mantissa part. Floating point subtraction can be performed by the negation and addition operations. Therefore, it has the same complexity as addition.

Now, we prove that the exact graph size of floating-point addition under a fixed variable ordering grows exponentially with the size of the exponent and linearly with the size of the mantissa. Assume that the sizes of the exponent and the mantissa are n and m bits, respectively. We assume that the variable ordering is $S_x, S_y, ex_0, ey_0, \dots, ex_{n-1}, ey_{n-1}, x_{m-1}, \dots, x_0, y_{m-1}, \dots, y_0$.

Lemma 1: The size of the mantissa part of the resulting graph is $2^{n+1}(7m - 1) - 20m - 4$, where n and m are the numbers of bits of the exponent and mantissa parts respectively.

Proof: Theorem 2 showed that the number of distinct mantissa sums is $2^{n+3} - 10$. Except the leaf nodes, each mantissa sum can be represented by $2m$ nodes, but there is some sharing among the mantissa graphs. First, let us look at the sharing among the mantissa sums of true addition. For case 4 in Equation 6, the graphs to represent function $0.X + 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$, which is also in the graph representing function $1.X + 0.Y$. Thus, there are $2^n - 1$ distinct mantissa sums to share the same graph($1.Y$). Again, the graphs to represent $1.X + 1.Y$ in case 2 and $2 \times 1.X + 0.Y$ in case 3, share the sub-graph $2+0.Y$, since $1.X + 1.Y = 0.X + (2+0.Y)$ and $2 \times 1.X + 0.Y = 2 \times 0.X + (2+0.Y)$. Therefore, we have to subtract $(2^n - 1)m$ nodes from the total nodes.

Then, let us look at the true subtraction. First, the graph to represent $0.X - 0.Y$ shares the sub-graph $0.Y$ with $0.X + 0.Y$ in true addition, because of the negation edge. For Case 4 in Equation 7, the graphs to represent function $0.X - 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$ in true addition. The graphs to represent $1.X - 0.Y$ in case 3 and $2 \times 1.X - 1.Y$ in case 0, share the sub-graph

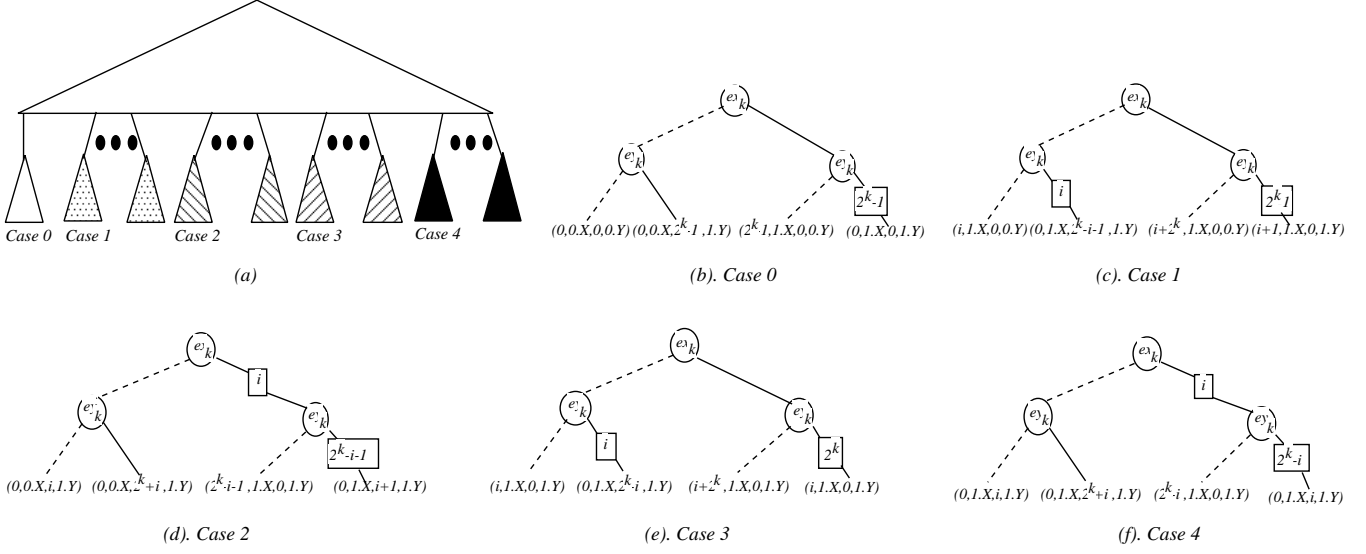


Figure 9. **Distinct sub-graphs after variable ey_{k-1} .** (a) Distinct sub-graphs after variable ey_{k-1} are divided into 5 types shown in graphs (b) to (f) which serve as template with a parameter i . (b) Case 0 only has one distinct graph. (c) $0 \leq i = EX_k - 1 \leq 2^k - 2$. (d) $0 \leq i = EY_k - 1 \leq 2^k - 2$. (e) $1 \leq i = EX_k - EY_k \leq 2^k - 2$. (f) $1 \leq i = EY_k - EX_k \leq 2^k - 2$.

$1 - 0.Y$, since $1.X + 1.Y = 0.X + (1 - 0.Y)$ and $2 \times 1.X - 1.Y = 2 \times 0.X + (1 - 0.Y)$. Therefore, we have to subtract $(2^n + 1)m$ nodes from the total nodes. Thus, the number of non-leaf nodes to represent these distinct mantissa sums is $(7 \times 2^n - 10) \times (2m)$.

The leaf nodes 1 and 0 are referenced by these non-leaf nodes. For true addition, the number of leaf nodes, except leaf node 1, is $2^n - 2$, since the leaf nodes of the mantissa sum for $EX < EY$ can be shared with the mantissa sum for $EX > EY$. To be specific, the leaf nodes are generated by the sum of the leading 1s in the form of $1 + 1 \times 2^{EY-EX}$ or $1 \times 2^{EX-EY} + 1$, and there are only $2^n - 2$ sums. Similarly, for true subtraction, there are $2^n - 4$ leaf nodes, but the leaf nodes 3 ($2^2 - 1$) and 0 ($2^0 - 1$) already exist. Thus, the total number of leaf nodes is $2 + (2^n - 2) + (2^n - 4) = 2^{n+1} - 4$. Therefore, the size of the mantissa part of resulting graph is $(7 \times 2^n - 10) \times (2m) + 2^{n+1} - 4 = 2^{n+1}(7m - 1) - 20m - 4$. \square

Lemma 2: For all $n \geq 2$, the number of *PHDD nodes of the exponent part of the resulting graph is $5 \times 2^{n+2} - 16 \times n - 18$.

Proof: As mentioned before, the resulting graph can be divided into two parts: true addition and true subtraction. First, we prove that the number of nodes of the exponent part for true addition is $5 \times 2^{n+1} - 8 \times n - 9$. We prove this claim by the induction on the number of exponent bits n .

Base Case: If $n = 2$, the number of exponent nodes for true addition is $5 \times 2^{2+1} - 8 \times 2 - 9 = 15$ as shown in Figure 8.

Induction Step: Assume the claim holds for $n = k$. To prove that the claim holds for $n = k + 1$, let EX_k and EY_k represent the low k bits of EX and EY . Thus, EX is represented as $2^k \times ex_k + EX_k$. Equation 4 can be rewritten as the following:

$$F_X + F_Y = (-1)^{S_x} \times 2^{-B} \times$$

$$\begin{cases} 2^1 \times \{2^{(l \times ex_k)} \times G + 2^{(l \times ey_k)} \times H\} & \text{Case 0} \\ 2^1 \times \{2^{(EX_k - 1 + l \times ex_k)} \times 1.X + H \times 2^{(l \times ey_k)}\} & \text{Case 1} \\ 2^1 \times \{G \times 2^{(l \times ex_k)} + 1.Y \times 2^{(EY_k - 1 + l \times ey_k)}\} & \text{Case 2} \\ 2^{EY_k} \times \{(2^{p+2^k \times ex_k}) \times 1.X + 1.Y \times 2^{(2^k \times ey_k)}\} & \text{Case 3} \\ 2^{EX_k} \times \{2^{(2^k \times ex_k)} \times 1.X + 1.Y \times 2^{(p+2^k \times ey_k)}\} & \text{Case 4} \end{cases} \quad (8)$$

where l is $2^k - 1$, p is $|EX_k - EY_k|$, and G (H) is $0.X$ ($0.Y$) if ex_k (ey_k) is 0; otherwise, G (H) is $1.X$ ($1.Y$). Figure 9.a illustrates the distinct sub-graphs after expanding variable ey_{k-1} . These sub-graphs are divided into five types, according to the cases in Equation 8. For Case 0 ($EX_k = EY_k = 0$), there is only one distinct sub-graph. For Case 1 ($EX_k > 0$ and $EY_k = 0$), there are $2^k - 1$ distinct sub-graphs, since the number of possible value of EX_k is $2^k - 1$ and each value of EX_k will generate a unique function. Similarly, there are $2^k - 1$, $2^k - 2$, and $2^k - 1$ distinct sub-graphs for Case 2 ($EX_k = 0$ and $EY_k > 0$), Case 3 ($EY_k > 0$ and $EY_k < EX_k$), and Case 4 ($EX_k > 0$ and $EX_k < EY_k$), respectively. Thus, the total number of distinct sub-graphs is $2^{k+2} - 4$.

Figures 9.b shows the sub-graph for Case 0. In the graphs, each tuple (i, P, j, Q) represents $2^i \times P + 2^j \times Q$. For example, tuple $(0, 0.X, 0, 0.Y)$ represents $2^0 \times 0.X + 2^0 \times 0.Y$. Figures 9.c to 9.f show the graphs with a parameter i for Cases 1, 2, 3 and 4, which serve as the template of the graphs in the cases. For instance, the graph in Case 1 with $i = 1$ represents the function $2^{(EX_k - 1 + (2^k - 1) \times ex_k)} \times 1.X + H \times 2^{((2^k - 1) \times ey_k)}$ with $EX_k = 1$ in Case 2 of Equation 8.

Since each sub-graph is distinct, the nodes with variable ex_k are unique (i.e. no sharing). Observing from these five types of sub-graphs, the possible sharing among the nodes with variable ey_k is these cases: the ey_k nodes in case 2 share with that in cases 3 and 4, and the nodes in case 3 share with that in case 4. For the first case, the possible sharing is the right ey_k nodes in Figure 9.e and Figure 9.g. Observe that these two ey_k node will be that same in the graph with $i = j$ in case 2 and the graph with $i = j + 1$

in case 4. Since the possible values of i are ranged from 0 to $2^k - 3$, there are $2^k - 2$ ey_k nodes shared. When $i = 2^k - 2$, the right ey_k node in the graph of case 2 will be shared with the left ey_k node in the graph with $i=1$ in Figure 9.e. Therefore, all of the right ey_k nodes in Case 2 are shared nodes and are $2^k - 1$ nodes. For the second case, the possible sharing is the left ey_k node in Figure 9.e and the right ey_k node in Figure 9.f. Observe that when $i_1 + i_2 = 2^k$, the left ey_k node in the graph with $i = i_1$ in case 3 is the same as the right ey_k node in the graph with $i = i_2$ in case 4. Since $2 \leq i_1 \leq 2^k - 2$ and $0 \leq i_2 \leq 2^k - 2$, there are $2^k - 3$ nodes shared. Therefore, the total number of exponent nodes are $5 \times 2^{k+1} - 8 \times k - 9 + 3 \times (2^{k+2} - 4) - (2^k - 1) - (2^k - 3) = 5 \times 2^{(k+1)+1} - 8 \times (k+1) - 9 = 5 \times 2^{n+1} - 8 \times n - 9$.

Similarly, the number of nodes of the exponent part for true subtraction is $5 \times 2^{n+1} - 8 \times n - 9$. Therefore, the size of the exponent part of the resulting graph is $5 \times 2^{n+2} - 16 \times n - 18$. \square

Theorem 3: For floating-point addition, the size of the resulting graph is $2^{n+1} \times (7m + 9) - 20m - 16n - 19$.

Proof: The size of the resulting graph is the sum of the nodes for the sign, exponent and mantissa parts. The nodes for the sign part are 3 as shown in Figure 8. Lemma 1 and 2 have shown the sizes of the mantissa and exponent parts respectively. Therefore, their sum is $2^{n+1} \times (7m + 9) - 20m - 16n - 19$. \square

In our experience, the sizes of the resulting graphs for multiplication and addition are hardly sensitive to the variables ordering of the exponent variables. They exhibit a linear growth for multiplication and exponential growth for addition for almost all possible ordering of the exponent variables. It is more logical to put the variables with Shannon decompositions on the top of the variables with the other decompositions.

VI. COMPARISONS WITH *BMD, HDD AND K*BMD

The major difference between *PHDD and the other three diagrams is in their ability to represent functions that map Boolean variables into floating-point values and their use of negation edges. Table I summarizes the differences between them.

Features	*PHDD	*BMD	HDD	K*BMD
Additive weight	No	No	No	Yes
Multiplicative weight	Powers of 2	GCD	No	GCD
# of decomp.	3	1	6	3
Negation edge	Yes	No	No	No

Table I. Differences among four different diagrams.

Compared to *BMDs [4], *PHDDs have three different decomposition types and a different method to represent and extract edge weights. These features enable *PHDDs to represent floating-point functions effectively. *BMD's edge weights are extracted as the greatest common divisor (GCD) of two children. In order to verify the multiplier with a size larger than 32 bits, *BMDs have to use multiple precision representation for integers to avoid the machine's

32-bit limit. This multiple precision representation and the GCD computation are expensive for *BMDs in terms of CPU time. Our powers of 2 method not only allows us to represent the floating-point functions but also improves the performance compared with *BMD's GCD method.

Compared with HDDs having six decompositions [10], *PHDDs have only three of them. In our experience, these three decompositions are sufficient to represent floating-point functions and verify floating-point arithmetic circuits. The other three decomposition types in HDDs may be useful for other application domains. Another difference is that *PHDDs have negation edges and edge weights, but HDDs do not. These features not only allow us to represent floating-point functions but also reduce the graph size.

*PHDDs have only multiplicative edge weights, but K*BMDs [15] allow additive and multiplicative weights at the same time. The method of extracting the multiplicative weights is also different in these two representations. *PHDDs extract the powers-of-2 and choose the minimum of two children, but K*BMDs extract the greatest common divisor of two children like *BMDs. The additive weight in K*BMDs can be distributed down to the leaf nodes in *PHDD by recursively distributing to one or two branches depending on the decomposition type of the node. In our experience, additive weights do not significantly improve the sharing in the circuits we verified. The sharing of the additive weight may occur in other application domains.

VII. EXPERIMENTAL RESULTS

We have implemented *PHDD with basic BDD functions and applied it to verify arithmetic circuits. Integer multiplier circuits and *BMD package can be obtained from Yirng-An Chen's WWW page¹. The circuit structure for four different types of multipliers are manually encoded in a C program which calls the BDD and *BMD operations. We also integrated our *PHDD package with the C program. Our measurements are obtained on Sun Sparc 10 with 256 MB memory. The memory measurements are the memory used by *BMD or *PHDD packages (i.e., the peak memory usage).

A. Integer Multipliers

To verify integer multipliers, we first use the hierarchical verification approach described in [4]. In this approach, the multiplier is partitioned into several sub-circuits and each sub-circuits is verified independently against its specification. Then, these sub-specifications are composed to check against the overall specification.

Based on this approach, Table II shows the performance comparison between *BMD and *PHDD for different integer multipliers with different word sizes. Based on these experiments, the complexity of *PHDDs for the multipliers for the CPU time still grows quadratically with the word size. Compared with *BMDs, *PHDDs are at least 5 times faster, since the edge weight manipulation of *PHDDs only requires integer addition and subtraction, but *BMDs re-

¹<http://www.cs.cmu.edu/~yachen/home.html>.

Circuits		CPU Time (Sec.)			Memory(MB)		
		16	64	256	16	64	256
Add-Step	*BMD	1.4	15.4	354.4	0.7	0.8	1.1
	*PHDD	0.2	2.2	40.0	0.1	0.2	0.6
	Ratio	7.0	7.0	8.9	7.0	4.0	1.8
CSA	*BMD	1.6	26.9	591.7	0.7	0.8	2.1
	*PHDD	0.3	3.5	50.7	0.1	0.3	0.9
	Ratio	5.3	7.7	11.7	7.0	2.7	2.3
Booth	*BMD	2.1	34.1	782.2	0.7	0.9	1.8
	*PHDD	0.2	3.0	62.6	0.1	0.3	1.3
	Ratio	10.5	11.4	12.5	7.0	3.0	1.4
Bit-Pair	*BMD	1.2	17.4	378.6	0.7	0.9	2.3
	*PHDD	0.2	2.2	36.1	0.2	0.3	1.3
	Ratio	6.0	7.9	10.5	3.5	3.0	1.8

Table II. Performance comparison between *BMD and *PHDD for different integer multipliers. Results are shown for three different words. The ratio is obtained by dividing the result of *BMD by that of *PHDD.

quire a multiple precision representation for integers and perform costly multiple precision multiplication, division, and GCD operations. While increasing the word size, the *PHDD's speedup is increasing, because *BMDs requires more time to perform multiple precision multiplication and division operations. Interestingly, *PHDDs also use less memory than *BMDs, since the edge weights in *BMDs are explicitly represented by extra nodes, while *PHDDs embed edge weights into the node structure. The node sizes for both packages are 20 bytes.

Bits	Mult A		Mult B		Mult C	
	CPU	MEM	CPU	MEM	CPU	MEM
4	0.1	6.2	0.1	6.1	0.1	7.6
8	0.2	6.9	0.3	6.9	0.3	7.8
16	1.2	8.1	1.2	8.3	1.1	8.2
32	6.4	8.7	10.6	9.2	5.9	9.1
64	37.8	10.7	133.9	13.4	47.8	13.0
order	$n^{2.5}$	$n^{1.1}$	$n^{3.5}$	$n^{1.3}$	$n^{2.8}$	$n^{1.2}$

Table III. Results of three different types of multipliers. Mult A is based on bit-pair and array multiplier. Mult B is based on Bit-Pair and Wallace-tree multiplier. Mult C is based on Booth-Radix4 and Wallace-tree Multiplier.

The drawback of the hierarchical verification approach is that the circuits must be partitioned into hierarchical forms. To overcome this constraint, Hamaguchi *et al* [18] proposed a backward substitution method to compute the output *BMDs of integer multipliers without any circuit knowledge. Keim *et al* [20] have shown that this approach with *BMDs is bounded by $O(n^4)$, in worse case. In [6], Chen *et al* have shown that the backward substitution method using *PHDDs is also bounded by $O(n^4)$, in worse case. Table III shows their performance measurements for different integer multipliers with different word sizes. These experiment results were performed on Sun Ultra-Sparc 60 (450MHz). Mult A is based on bit-pair and array multiplier. Mult B is based on Bit-Pair and Wallace-tree multiplier. Mult C is based on Booth-Radix4 and Wallace-tree Multiplier. Since different types of multipliers affect the behavior of computed cache table during backward substitution, the computation complexity of the actual mea-

surements varies from $O(n^{2.5})$ to $O(n^{3.5})$. Based on their experiments, for buggy multipliers, *PHDDs will blow up exponentially during the backward substitution.

B. Floating Point Multipliers

Circuits	CPU Time (Sec.)			Memory(MB)		
	16	64	256	16	64	256
Add-Step	0.24	2.29	39.77	0.13	0.18	0.65
CSA	0.29	3.08	53.98	0.14	0.30	0.88
Booth	0.25	3.85	67.38	0.16	0.30	1.26
Bit-Pair	0.21	2.10	38.54	0.15	0.33	1.33

Table IV. Performance for different floating-point multipliers. Results are shown for three different mantissa word size with fixed exponent size 11.

To perform floating-point multiplication operations before the rounding stage, we introduced an adder to perform the exponent addition and logic to perform the sign operation in the C program. Based on the hierarchical verification approach, Table IV shows CPU times and memory requirements for verifying floating-point multipliers with fixed exponent size 11. Based on these experimental results, the complexity of verifying floating-point multiplier before rounding still grows quadratically. In addition, the computation time is very close to the time of verifying integer multipliers, since the verification time of an 11-bit adder and the composition and verification times of a floating-point multiplier from integer mantissa multiplier and exponent adder are negligible. The memory requirement is also similar to that of the integer multiplier.

C. Floating Point Addition

Exp. Bits	No. of Nodes		CPU (Sec.)		Mem.(MB)	
	23	52	23	52	23	52
4	4961	10877	0.2	0.7	0.4	0.7
5	10449	22861	0.7	1.3	0.7	1.1
6	21441	46845	1.1	3.5	1.1	2.0
7	43441	94829	2.7	6.9	1.9	3.8
8	87457	190813	7.2	16.8	3.6	7.5
9	175505	382797	15.0	41.3	7.2	14.8
10	351617	766781	33.4	103.2	14.3	29.5
11	703857	1534765	72.8	262.4	26.5	54.9
12	1408353	3070749	163.2	573.7	54.1	110.9
13	2817361	6142733	398.3	1303.8	112.5	226.0

Table V. Performance for floating-point additions. Results are shown for 10 different exponent word size with fixed mantissa size 23 and 52 bits.

Table V shows the performance measurements of precise floating-point addition operations with different exponent bits and fixed mantissa sizes of 23 and 52 bits, respectively. Both the number of nodes and the required memory double, while increasing one extra exponent bit. For the same number of exponent bits, the measurements for the 52-bit mantissa are approximately twice the corresponding measurements for the 23-bit mantissa. In other words, the complexity grows linearly with the mantissa's word size. Due to the cache behavior, the CPU time is not doubling

(sometimes, around triple), while increasing one extra exponent bit. For the double precision of IEEE standard 754 (the numbers of exponent and mantissa bits are 11 and 52 respectively), it only requires 54.9MB and 262.4 seconds. These values indicate the possibility of the verification of an entire floating-point adder for IEEE double precision. For IEEE extended precision, floating-point addition will require at least $226.4 \times 8 = 1811.2$ MB memory. In order to verify IEEE extended precision addition, it is necessary to avoid the exponential growth of floating-point addition.

VIII. CONCLUSIONS AND FUTURE WORK

We have described a new representation, *PHDD, to represent functions that map Boolean variables into integer or floating-point values. We also applied *PHDDs to verify integer and floating-point multipliers. For integer multipliers, the CPU time of *PHDDs grows quadratically and is at least 6 times faster than *BMDs. For floating-point multipliers, the verification time is close to the verification time of integer multiplier. In addition, we showed that the *PHDD representation for floating-point multiplication grows linearly with the word size. For floating-point addition, we showed the graph size of the result grows exponentially with the word size of the exponent, but linearly with the size of the mantissa.

To verify circuit designs automatically, we would like to integrate the *PHDD package into word-level SMV [12] and extend word-level SMV, if needed, to handle floating-point arithmetic circuits. Then, we will look into the rounding stage and entire floating-point adders. Earlier results [9] show that the rounding stage itself can be handled with HDDs and therefore with *PHDDs. To verify entire floating-point adders, we need to develop some techniques to avoid the exponential growth. Our representation for floating-point addition represents the precise values of all possible combinations, but in the actual circuit design, there are only about 200 interesting mantissa sums. Based on this knowledge, we will develop a technique to avoid the exponential growth of floating-point addition. As mentioned in previous sections, we will further pursue handling infinite and NaN cases. We need to develop some techniques or introduce special symbols to handle these cases.

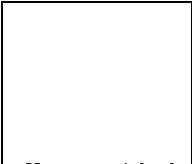
ACKNOWLEDGEMENT

We thank Xudong Zhao for valuable discussions on HDDs and verification of arithmetic circuits.

REFERENCES

- [1] K. Brace, R. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.
- [2] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 275–293, November 1996.
- [3] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, 1994.
- [4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the*

- [5] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *High Order Logic Theorem Proving and Its Applications*, September 1995.
- [6] J.-C. Chen and Y.-A. Chen. Equivalence checking of integer multipliers. In *Proceedings of ASP-DAC '2001*, pages 169–174, Yokohama, Japan, Feb. 2001.
- [7] Y.-A. Chen and R. E. Bryant. ACV: An arithmetic circuit verifier. In *Proceedings of the International Conference on Computer-Aided Design*, pages 361–365, November 1996.
- [8] Y.-A. Chen and R. E. Bryant. *PHDD: An efficient graph representation for floating point circuit verification. In *Proceedings of the International Conference on Computer-Aided Design*, pages 2–7, November 1997.
- [9] 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. In *Proceedings of the Formal Methods on Computer-Aided Design*, pages 19–33, November 1996.
- [10] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 159–163, November 1995.
- [11] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division using theorem proving techniques. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 111–122, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- [12] E. M. Clarke, M. Khaira, and X. Zhao. Word level model checking - Avoiding the Pentium FDIV error. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 645–648, June 1996.
- [13] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 54–60, June 1993.
- [14] T. Coe. Inside the Pentium Fdiv bug. *Dr. Dobbs Journal*, pages pp. 129–135, April 1996.
- [15] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: a new data structure for verification. In *Proceedings of European Design and Test Conference*, pages 2–8, March 1996.
- [16] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 415–419, June 1994.
- [17] L. M. Fisher. Flaw reported in new intel chip. *New York Times*, pages D, 4:3, May 6 1997.
- [18] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 78–82, November 1995.
- [19] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34, June 1996.
- [20] M. Keim, M. Martin, R. Drechsler, and P. Molitor. Polynomial formal verification of multipliers. In *Proceedings of 15th IEEE VLSI Test Symposium*, pages 150–155, 1997.
- [21] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the amd5_{K86}^{T^M} floating-point division program. In *IEEE Transactions on Computers*, pages 9:913–926, September 1998.



Yirng-An Chen received the B.S. and M.S. degrees in computer science from TungHai University, Taiwan, in 1987 and from National Tsing Hua University, Taiwan, in 1989, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University, PA, in 1998.

He was with the Department of Computer and Information Science of National Chiao Tung University, where he was an assistant professor from 1998 to 2001. Since then, he is a principal engineer at ~~Novas Software Inc.~~, San Jose, CA. His Research interests include formal verification, symbolic simulation and debugging.

Dr. Chen received Best Paper Award at 32nd Design Automation Conference in 1995.