# *PHDD: An Efficient Graph Representation for Floating Point Circuit Verification †

Yirng-An Chen                Randal E. Bryant
yachen@cs.cmu.edu                Randy.Bryant@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA 15213

## Abstract

*Data structures such as *BMDs, HDDs, and K*BMDs provide compact representations for functions which map Boolean vectors into integer values, but not floating point values. 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. We verified floating point multipliers before the rounding stage automatically.*

## 1 Introduction

Binary Moment Diagrams (BMDs) [3] 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 [4]. Clarke, *et al.* [7] 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.* [10] 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 [2]. But using rational numbers in the representation requires more space to store the numerator and denominator separately, and more computation to extract the rational numbers.

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 [4, 6, 8]. 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 [6]. The verification of these three sub-circuits was performed automatically by word-level SMV [8], 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.

†This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-96-C-0071. Appeared in ICCAD97.

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 [7], 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 practical for formats up to IEEE double precision.

Based on a hierarchical verification methodology [3, 4], 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 [6] show that the rounding stage itself can be handled.

## 2 BMDs, *BMDs and HDDs

For expressing functions Boolean variables into integer values, BMDs[3] use the moment decomposition of a function:

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

where $\cdot$,$+$ and $-$ denote multiplication, addition and subtraction, respectively. Term $f_x$ ($f_{\overline{x}}$) 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, $f_{\delta x} = f_x - f_{\overline{x}}$ is called the linear moment of $f$ with respect to $x$. This terminology arises by viewing $f$ as being a linear function with respect to its variables, and thus $f_{\delta x}$ is the partial derivative of $f$ with respect to $x$. The negative cofactor $f_{\overline{x}}$ 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 [10]. 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.

Clarke, *et al.* [7] 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:

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

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

| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 2 |
| 1 | 0 | 4 |
| 1 | 1 | 8 |

(a)

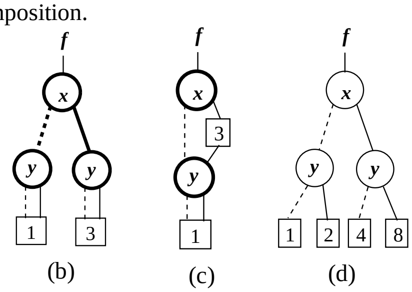


Figure 1: **An integer function with Boolean variables, $f = 1 + y + 3x + 3xy$, is represented by (a) Truth table, (b) BMDs, (c) *BMDs, (d) HDDs with Shannon decompositions.** The dashed-edges are 0-branches and the solid-edges are the 1-branches. The variables with Shannon and positive Davio decomposition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

As an example, Figure 1 show 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 [9]). 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 $f_{\overline{x}}$, $f_{\overline{x}}$, and $f_x$ ($f_x$, $f_{\delta x}$ and $f_{\delta\overline{x}}$) 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 $f_{\overline{x}} = 1 + y$, 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 $3 \times f_{\overline{x}}$. 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, $X$=$y$+2$x$, 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 and HDDs cannot represent the functions as $f = 2^{X-B}$, where $B$ is a constant, because they can only represent integer functions.

## 3 The *PHDD Data Structure

In this section, we introduce a new data structure, Multiplicative Power Hybrid Decision Diagrams (*PHDDs), 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.

### 3.1 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:

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

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.

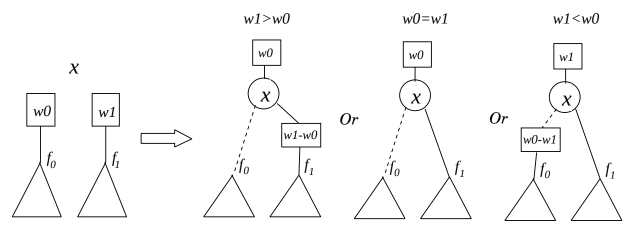


Figure 2: **Normalizing the edge weights.**

In addition to the HDD reduction rules [7], 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 [2]. 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 $f_1$. Base 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.

### 3.2 Negation Edge

Negation edges are commonly used in BDDs [1] and KFDDs [11], 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 $\overline{f}$ for Boolean functions.

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 [11], 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. These guarantee the canonical form for *PHDDs.

## 4 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.

### 4.1 Representation of Integers

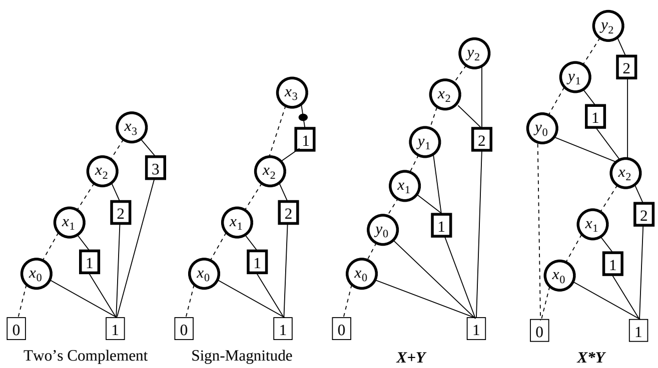


Figure 3: ***PHDD Representations of Integers and Integer operations.** Each variable uses positive Davio decomposition. The graphs grow linearly with word size.

*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: $x_{n-1}$, ..., $x_1$, $x_0$. These variables can be considered to represent an integer $X$ according to some encoding, e.g., unsigned binary or two's complement. Figure 3 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 3 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 $x_i 2^i Y$. In summary, while representing the integer functions, *PHDDs with positive Davio decompositions usually will get the most compact representation among these three decompositions.

### 4.2 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:

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

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

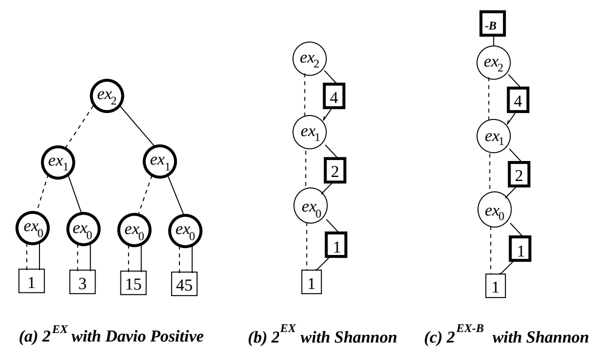


Figure 4: ***PHDD Representations of $2^{EX}$ and $2^{EX-B}$.** The graph grows linearly in the word size with Shannon, but grows exponentially with positive Davio.

Figure 4 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 $c^{2^i ex_i} = (c^{2^i})^{ex_i}$. In the graph with Shannon decompositions, a vertex labeled by variable $ex_i$ 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.

Figure 5 shows the *PHDD representations for the floating point

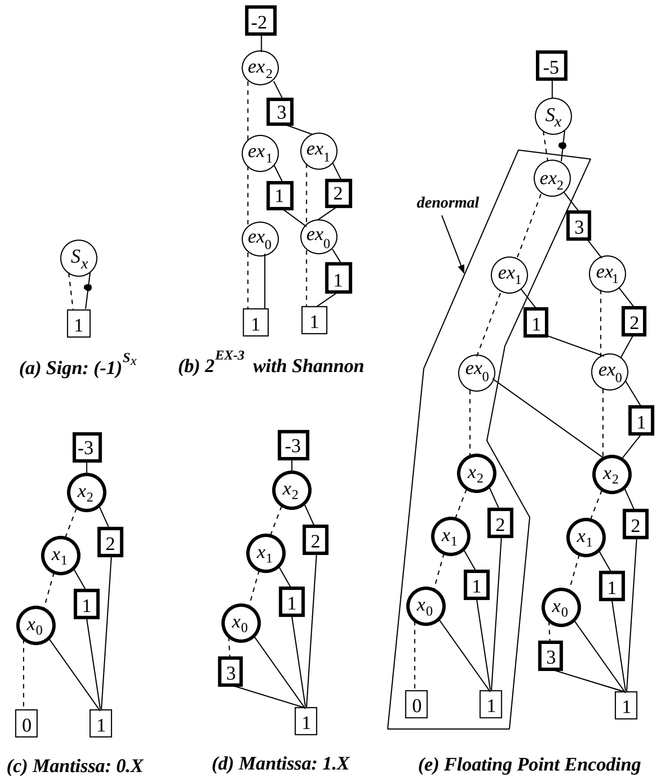


Figure 5: **Representations of floating point encodings.**

encoding, where $EX$ has 3 bits, $X$ has 4 bits and the bias $B$ is 3. The sign $S_x$ and $\vec{ex}$ variables use Shannon decomposition, while variables $\vec{x}$ use positive Davio. Figure 5.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 5.b shows the *PHDD representation for the exponent part $2^{EX-3}$. The graph is more complicated than Figure 4.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 $ex_2$, has two nodes: one to represent the denormal number case and another to represent normal number case. Figure 5.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 5.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 5.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.

## 4.3 Floating Point Multiplication and Addition

This section presents floating point multiplication and addition based on *PHDDs. Here, we show the representations of these

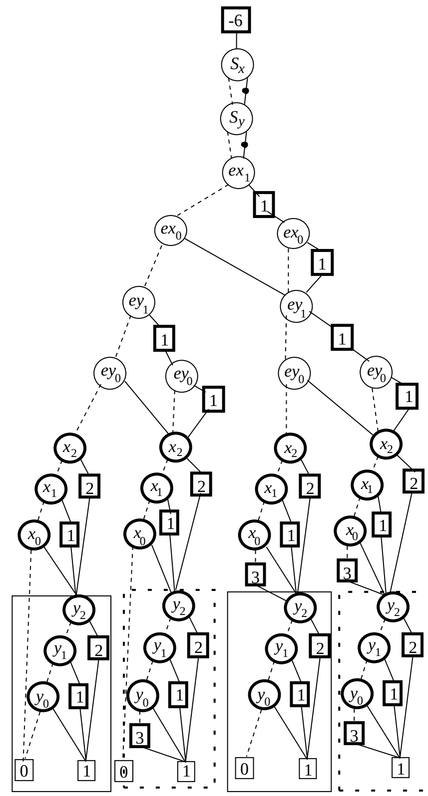


Figure 6: **Representation of floating point multiplication.**

operations before rounding. In other words, the resulting *PHDDs represent the precise results of the floating point operations. For floating point multiplication, the size of the resulting graph grows linearly with the word size. For floating point addition, the size of the resulting graph grows exponentially with the size of the exponent part.

Let $F_X = (-1)^{S_x} \times v_x.X \times 2^{EX-B}$ and $F_Y = (-1)^{S_y} \times v_y.X \times 2^{EY-B}$, 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: $(-1)^{S_x \oplus S_y} \times 2^{-2B} \times$

$$\begin{cases} 2^1 \times 2^1 \times (0.X \times 0.Y) & Case\ 0: EX = 0\ EY = 0 \\ 2^1 \times 2^{EY} \times (0.X \times 1.Y) & Case\ 1: EX = 0\ EY \neq 0 \\ 2^{EX} \times 2^1 \times (1.X \times 0.Y) & Case\ 2: EX \neq 0\ EY = 0 \\ 2^{EX} \times 2^{EY} \times (1.X \times 1.Y) & Case\ 3: EX \neq 0\ EY \neq 0 \end{cases}$$

Figure 6 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 6. The second and fourth products share the common sub-function $2^3 + Y$ shown by the dashed rectangles in Figure 6. In [5], we have proved that the size of the resulting graph of floating point multiplication is $6(n + m) + 3$ with the variable ordering given in Figure 6, where $n$ and $m$ are the number

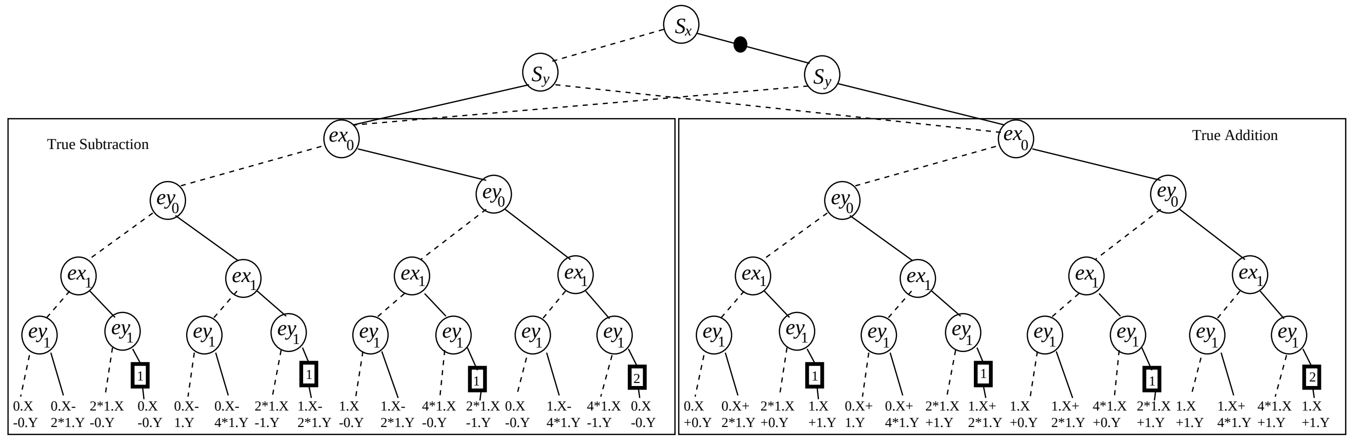


Figure 7: **Representation of floating point addition.** For simplicity, the graph only shows sign bits, exponent bits and the possible combinations of mantissa sums.

of bits in the exponent and mantissa parts.

For floating point addition, the size of the resulting graph grows exponentially with the size of the exponent part. In [5], we have proved that the number of distinct mantissa sums of $F_X + F_Y$ is $2^{n+3} - 10$, where $n$ is the number of bits in the exponent part. Figure 7 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. According to the sign bits of two words, the graphs can be divided into two sub-graphs: true addition and true subtraction which represent the addition and subtraction of two words, respectively. 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. Readers can refer to [5] for a detailed discussion of floating point addition. Floating point subtraction can be performed by the negation and addition operations. Therefore, it has the same complexity as addition.

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.

## 5 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 page[1]. 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.

### 5.1 Integer Multipliers

Table 1 shows the performance comparison between *BMD and *PHDD for different integer multipliers with different word sizes.

[1] http://www.cs.cmu.edu/∼yachen/home.html.

| Circuits | | CPU Time (Sec.) | | | Memory(MB) | | |
|---|---|---|---|---|---|---|---|
| | | 16 | 64 | 256 | 16 | 64 | 256 |
| Add-Step | *BMD | 1.40 | 15.38 | 354.38 | 0.67 | 0.77 | 1.12 |
| | *PHDD | 0.20 | 2.24 | 39.96 | 0.11 | 0.18 | 0.64 |
| | Ratio | 7.0 | 6.8 | 8.9 | 6.0 | 4.3 | 1.8 |
| CSA | *BMD | 1.61 | 26.91 | 591.70 | 0.67 | 0.80 | 2.09 |
| | *PHDD | 0.25 | 3.45 | 50.72 | 0.14 | 0.30 | 0.88 |
| | Ratio | 6.4 | 7.8 | 11.7 | 4.8 | 2.7 | 2.4 |
| Booth | *BMD | 2.05 | 34.09 | 782.20 | 0.70 | 0.86 | 1.84 |
| | *PHDD | 0.21 | 2.97 | 62.56 | 0.14 | 0.30 | 1.26 |
| | Ratio | 9.7 | 11.5 | 12.5 | 5.0 | 2.9 | 1.5 |
| Bit-Pair | *BMD | 1.21 | 17.35 | 378.64 | 0.70 | 0.86 | 2.34 |
| | *PHDD | 0.20 | 2.17 | 36.10 | 0.15 | 0.33 | 1.33 |
| | Ratio | 6.0 | 8.0 | 10.5 | 4.7 | 2.6 | 1.8 |

Table 1: **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.

For the CPU time, the complexity of *PHDDs for the multipliers still grows quadratically with the word size. Compared with *BMDs, *PHDDs are at least 6 times faster, since the edge weight manipulation of *PHDDs only requires integer addition and subtraction, but *BMDs require 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.

### 5.2 Floating Point Multipliers

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. Table 2 shows CPU times and memory requirements for verifying floating point multipliers with fixed exponent size 11. Observe that the complexity of verifying the 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 expo-

| 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 2: **Performance for different floating point multipliers.** Results are shown for three different mantissa word size with fixed exponent size 11.

nent adder are negligible. The memory requirement is also similar to that of the integer multiplier.

### 5.3 Floating Point Addition

| Exponent | No. of Nodes | | CPU Time (Sec.) | | Memory(MB) | |
|---|---|---|---|---|---|---|
| Bits | 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 3: **Performance for floating point additions.** Results are shown for three different exponent word size with fixed mantissa size 23 and 52 bits.

Table 3 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 = 1811.2MB memory. In order to verify IEEE extended precision addition, it is necessary to avoid the exponential growth of floating point addition.

## 6 Future Work

To verify circuit designs automatically, we would like to integrate the *PHDD package into word-level SMV [8] and extend word-level SMV, if needed, to handle the floating point arithmetic circuits. Then, we will look into the rounding stage and entire floating-point adders. Earlier results [6] 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. We also thank Manish Pandey, Alok Jain, Shipra Panda and Bwolen Yang for proofreading this paper.


## References


[1] BRACE, K., RUDELL, R., AND BRYANT, R. E. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 40–45.

[2] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic functions with binary moment diagrams. Tech. Rep. CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, 1994.

[3] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1995), pp. 535–541.

[4] CHEN, Y.-A., AND BRYANT, R. E. ACV: An arithmetic circuit verifier. In *Proceedings of the International Conference on Computer-Aided Design* (November 1996), pp. 361–365.

[5] CHEN, Y.-A., AND BRYANT, R. E. *PBHD: An efficient graph representation for floating point circuit verification. Tech. Rep. CMU-CS-97-134, School of Computer Science, Carnegie Mellon University, 1997.

[6] CHEN, Y.-A., CLARKE, E. M., HO, P.-H., HOSKOTE, Y., KAM, T., KHAIRA, M., O'LEARY, J., AND ZHAO, X. Verification of all circuits in a floating-point unit using word-level model checking. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 19–33.

[7] CLARKE, E. M., FUJITA, M., AND ZHAO, X. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 159–163.

[8] CLARKE, E. M., KHAIRA, M., AND ZHAO, X. Word level model checking – Avoiding the Pentium FDIV error. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996), pp. 645–648.

[9] CLARKE, E. M., MCMILLAN, K., ZHAO, X., FUJITA, M., AND YANG, J. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 54–60.

[10] DRECHSLER, R., BECKER, B., AND RUPPERTZ, S. K*BMDs: a new data struction for verification. In *Proceedings of European Design and Test Conference* (March 1996), pp. 2–8.

[11] DRECHSLER, R., SARABI, A., THEOBALD, M., BECKER, B., AND PERKOWSKI, M. A. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proceedings of the 31st ACM/IEEE Design Automation Conference* (June 1994), pp. 415–419.