Verification of Floating-Point Adders

Yirng-An Chen   Randal E. Bryant

April 1998
CMU-CS-98-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The floating-point division bug in Intel's Pentium processor and the overflow flag erratum of the FIST instruction in Intel's Pentium Pro and Pentium II processor have demonstrated the importance and the difficulty of verifying floating-point arithmetic circuits. In this paper, we present a "black box" version of verification of FP adders. In our approach, FP adders are verified by an extended word-level SMV using reusable specifications without knowing the circuit implementation. Word-level SMV is improved by using Multiplicative Power HDDs (*PHDDs), and by incorporating conditional symbolic simulation as well as a short-circuiting technique. Based on a case analysis, the adder specification is divided into several hundred implementation-independent sub-specifications. We applied our system and these specifications to verify the IEEE double precision FP adder in the Aurora III Chip from the University of Michigan. Our system found several design errors in this FP adder. Each specification can be checked in less than 5 minutes. A variant of the corrected FP adder was created to illustrate the ability of our system to handle different FP adder designs. For each adder, the verification task finished in 2 CPU hours on a Sun UltraSPARC-II server.

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

Keywords: Binary Moment Diagram, *BMD, Hybrid Decision Diagram, HDD, K*BMD, Multiplicative Power Hybrid Decision Diagram, *PHDD, Arithmetic Circuit, Floating-Point Adder, IEEE Standard, Formal Verification

1 Introduction

The floating-point (FP) division bug [11, 22] in Intel's Pentium processor and the overflow flag erratum of the FIST instruction (floating-point to integer conversion) [14] in Intel's Pentium Pro and Pentium II processors have demonstrated the importance and the difficulty of verifying floating-point arithmetic circuits and the high cost of an arithmetic bug. FP adders are the most common units in floating-point processors. Modern high-speed FP adders [20, 23] are very complicated, because they require many types of modules: a right shifter for alignment, a left shifter for normalization, a leading zero anticipator (LZA), an adder for mantissas, a rounding unit, etc. Exhaustive simulation or formal verification can be used to ensure the correctness of FP adders.

Most of the IEEE floating-point standard have been formalized by Carreño and Miner [5] in the HOL and PVS theorem provers. Theorem provers have been used to verify arithmetic circuits [18, 21]. However, theorem provers require users to make use of detailed circuit knowledge and the verification process for floating-point circuits is very tedious. Another drawback of theorem provers is that the proofs are implementation-dependent.

After the famous Pentium division bug [11], Intel researchers applied word-level SMV [9] with Hybrid Decision Diagrams (HDDs) [8] to verify the functionality of the floating-point unit in one of Intel's processors [7]. Due to the limitations of HDDs, the FP adder was partitioned into several sub-circuits whose specifications were expressed in terms of integer functions. Each sub-circuit was verified individually based on assumptions about its inputs. The correctness of the overall circuit had to be ascertained manually from the verified specifications of the sub-circuits. This partitioning approach is error prone, since mistakes could be introduced in any of the following steps: partitioning the circuit and the specification, performing case analysis for each sub-circuit, and proving the overall correctness of the circuit, which potentially could have used a theorem prover. Moreover, their specifications are highly dependent on the circuit implementations.

The combination of model checking and theorem prover techniques was used to verify a IEEE double precision floating-point multiplier [1]. The circuit was partitioned into several sub-circuits which can be verified by model checking. The theorem prover handled the completeness of the proofs by inference rules to compose the verified specifications. This approach combines the strengths of both techniques. However, the proofs based on this approach are still implementation-dependent.

To the best of our knowledge, only two types of arithmetic circuits can be verified by treating them as *black boxes* (i.e., the specifications contain only the inputs and outputs). First, an integer adder can be verified by using Binary Decision Diagrams (BDDs) [2]. Second, Hamaguchi *et al* [15] presented the verification of integer multipliers without knowing their implementations using Multiplicative Binary Moment Diagrams (*BMDs) [4]. However, their approach does not work for incorrect designs, because the *BMDs explode in size and counterexamples can not be generated for debugging. None of the previous approaches can verify FP adders without knowing their circuit implementations.

In this paper, we present a black box version of verification of FP adders. In our approach, a FP adder is treated as a black box and is verified by an extended version of word-level SMV with reusable specifications. Word-level SMV is improved by using Multiplicative Power HDDs (*PHDDs) [6] to represent the FP functions, and by incorporating *conditional symbolic simulation* as well as a *short-circuiting* technique. The FP adder specification is divided into several hundred sub-specifications based on the sign bits and the exponent differences. These sub-specifications are implementation-independent, since they use only the input and output signals of FP adders.

The concept of conditional symbolic simulation is to perform the symbolic simulation of the circuit with some conditions to restrict the behavior of the circuit. This approach can be viewed as dynamically extracting circuit behavior under the given conditions without modifying the actual circuit. Can we verify the specifications of FP adders using conditional symbolic simulation, avoiding any use of circuit knowledge? We identify a conflict in variable orderings between the mantissa comparator and mantissa adder, which causes the BDD explosion in conditional symbolic simulation. A short-circuiting technique to overcome this ordering conflict problem is presented and integrated into word-level SMV package. In general, this short-circuiting technique can be used when different parts of the circuit are used under different operating conditions.

We used our system and these specifications to verify the FP adder in the Aurora III Chip [16] at the University of Michigan. This FP adder is based on the design described in [20], and supports IEEE double precision and all 4 IEEE rounding modes. In this verification work, we verified the FP adder only in the round-to-nearest mode, because we believe that this is the most challenging rounding mode for verification. Our system found several design errors. Each specification can be checked in less than 3 minutes or 5 minutes including counterexample generation. A variant of the corrected FP adder was created and verified to illustrate the ability of our system to handle different FP adder designs. For each FP adder, verification took 2 CPU hours. We believe that our system and specifications can be applied to directly verify other FP adder designs and to help find design errors.

The overflow flag erratum of the FIST instruction (FP to integer conversion) [14] in Intel's Pentium Pro and Pentium II processors has illustrated the importance of verification of the conversion circuits which convert the data from one format to another format (e.g., IEEE single precision to double precision). Since these circuits are much simpler than FP adders and only have one input operand, we believe that our system can be used to verify the correctness of these circuits.

2 *PHDD Preliminary

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

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

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 the 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 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 [13]. 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.* [8] 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:

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

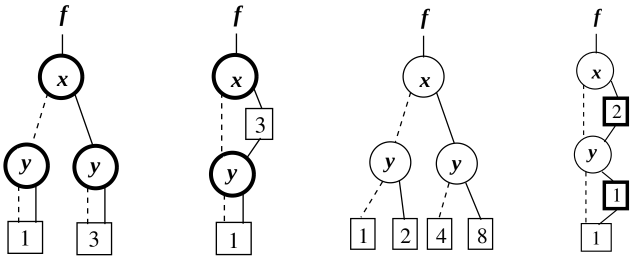
Here, $f_{\delta\overline{x}} = f_{\overline{x}} - f_x$ is the partial derivative of f with respect to $\overline{x}$. The BMD representation is a subset of HDDs. In other words, the HDD graph is the same as the BMD graph, if all of the variables use positive Davio decomposition.

Chen and Bryant [6] introduced multiplicative power-of-constant edge weights into HDDs to form a new representation called Multiplicative Power HDDs (*PHDDs). *PHDDs use three of HDD's six decompositions as expressed in Equation 2. Unlike the edge weights in *BMDs, the edge weights of *PHDDs are powers of a constant c. Thus, Equation 2 is rewritten as:

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

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 in the IEEE floating-point (FP) format is 2, we will consider only c = 2 for the remainder of this paper. Observe that w can be negative, allowing representation of rational numbers. The power edge weights enable us to represent functions mapping Boolean variables to FP values without using rational numbers in our representation.

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



(a) (b) (c) (d) (e)

Figure 1: **An integer function of Boolean variables, $f = 1 + y + 3x + 3xy$, is represented by (a) Truth table, (b) BMDs, (c) *BMDs, (d) HDDs with Shannon decompositions, (e) *PHDDs 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 the thin and thick vertices, respectively. The number i in the thin and thick boxes represents i and 2^i, respectively.

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 [10]). In our drawing, the variables with Shannon and positive Davio decomposition types are drawn in thin and thick vertices, respectively. The number i in the thin and thick boxes represents i and 2^i, respectively. A dashed line from a vertex with variable x points to the vertex represented function $f_{\overline{x}}$, $f_{\overline{x}}$, or f_x for the Shannon, positive Davio or negative Davio decompositions, respectively. Similarly, a solid line from a vertex with variable x points to the vertex represented function f_x, $f_{\delta x}$ or $f_{\delta\overline{x}}$ for the Shannon, positive Davio or negative Davio decompositions. 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 get $f_{\overline{x}} = 1 + y$, represented as the graph attached to the dashed-edge from vertex x, and $f_{\delta x} = 3 + 3y$, represented by the solid branch from 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 becomes Figure 1.c. This graph is called a Multiplicative BMD (*BMD), in which the greatest common divisor (GCD) is extracted 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 ($f_{\overline{x}}$) is constructed from the first two entries of the table, and the solid branch of vertex x (i.e., f_x) is constructed from the last two entries of the table. Observe that f_x is equal to $2^2 \times f_{\overline{x}}$. The *PHDDs can be constructed by extracting the powers-of-2 weights from the HDDs recursively.

Observe that if variables x and y are viewed as bits forming a 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 function. *BMDs can represent them efficiently due to 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. *PHDDs can represent this type of functions efficiently by adding the edge weight of $-B$ on the top the graph of 2^X. Therefore, *PHDDs can represent the FP encoding and operations efficiently. Readers can refer to [6] for more details of FP representation using *PHDDs.

3 Floating-Point Adders

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

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

where NaN denotes Not-a-Number and ∞ represents infinity. Let M_x = 1.N_x or 0.N_x. Let m be the number of mantissa bits including the bit on the left of the binary point and n be number of exponent bits. For IEEE double precision, m=53 and n=11.

Due to this encoding, an operation on two FP numbers cannot be rewritten as an arithmetic function of two inputs. For example, the addition of two FP numbers X (S_x, E_x, M_x) and Y (S_y, E_y, M_y) can not be expressed as X + Y, because of special cases when one of them is NaN or $\pm\infty$. Table 1 summarizes the

+		Y: $-\infty$	Y: F	Y: $+\infty$	Y: NaN
X	$-\infty$	$-\infty$	$-\infty$	*	NaN
	F	$-\infty$	$Round(X+Y)$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN

Table 1: **Summary of the FP addition of two numbers of X and Y.** F represents the normal and denormal numbers. * indicates FP invalid arithmetic operands.

possible results of the FP addition of two numbers X and Y, where F represents a normalized or denormalized number. The result can be expressed as $Round(X+Y)$ only when both operands have normal or denormal values. Otherwise, the result is determined by the case. When one operand is $+\infty$ and the other is $-\infty$, the FP adder should raise the FP invalid arithmetic operand exception.

Figure 2.a shows the block diagram of the SNAP FP adder designed at Stanford University [20]. This adder was designed for fast operation based on the following facts. First, the alignment (right shift) and normalization (left shift) needed for addition are mutually exclusive. When a massive right shift is performed

during alignment, the massive left shift is not needed. On the other hand, the massive left shift is required only when the mantissa adder performs subtraction and the absolute value of exponent difference is less than 2 (i.e. no massive right shift). Second, the rounding can be performed by having the mantissa adder generate $A+C$, $A+C+1$ and $A+C+2$, where A and C are the inputs of the mantissa adder, and using the final multiplexor to chose the correct output.

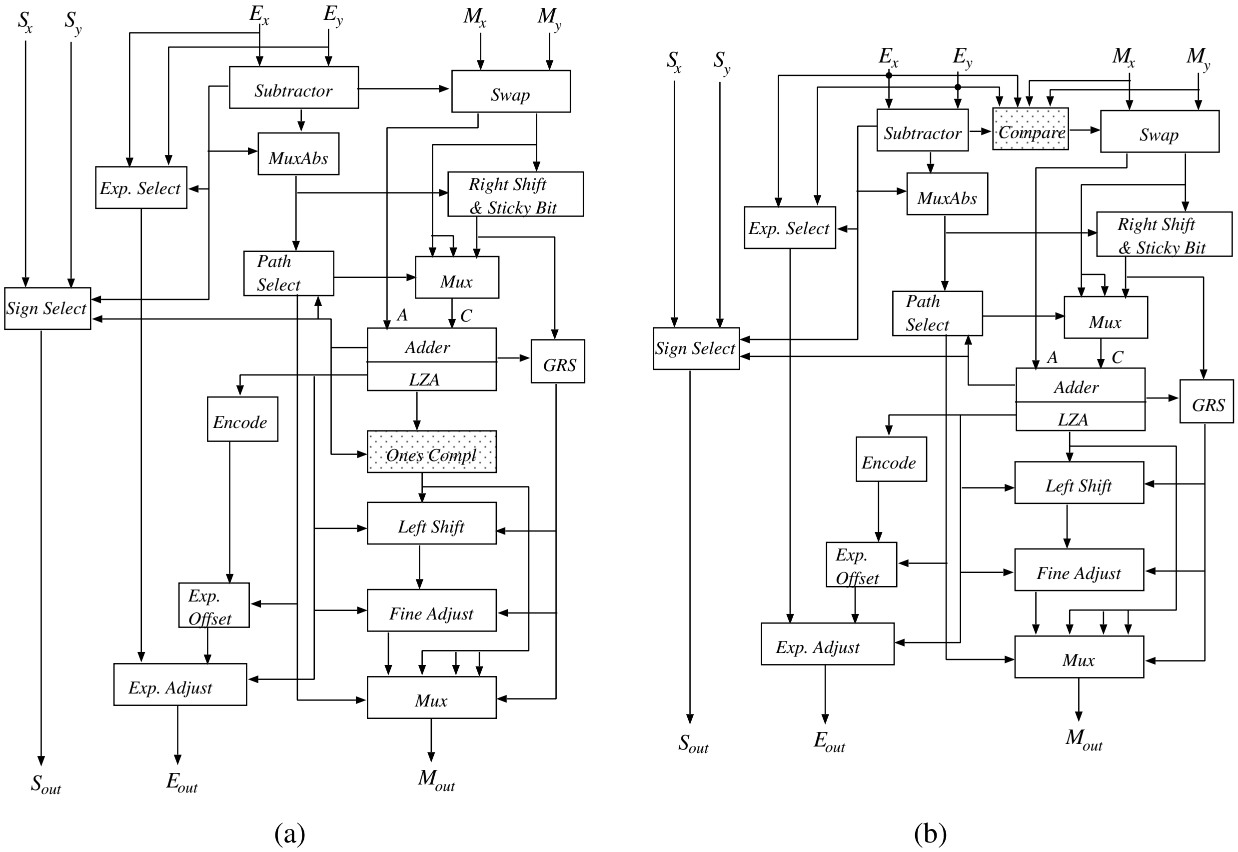


Figure 2: **The Stanford FP adder (a) and a variant (b).**

In the exponent path, the exponent subtractor computes the difference of the exponents. The *MuxAbs* unit computes the absolute value of the difference for alignment. The larger exponent is selected as the input to the exponent adjust adder. During normalization, the mantissa may need a right shift, no shift or a massive left shift. The exponent adjust adder is prepared to handle all of these cases.

In the mantissa path, the operands are swapped as needed depending on the result of the exponent subtractor. The inputs to the mantissa adder are: the mantissa with larger exponent (A) and one of the three versions of the mantissa with small exponent (C): unshifted, right shifted by 1, and right shifted by many bits. The *path select* unit chooses the correct version of C based on the value of exponent difference. The version right shifted by many bits is provided by the right shifter, which also computes the information needed for the sticky bit. The mantissa adder performs the addition or subtraction of its two inputs depending on the signs of both operands and the operation (add or subtract). If the adder performs subtraction, the mantissa with smaller exponent will first be complemented. The adder generates all possible outcomes ($A+C$, $A+C+1$, and $A+C+2$) needed to obtain the final, normalized and rounded result. The $A+C+2$ is required, because of the possible right shift during normalization. For example, when the most significant bits of A and C are 1, $A+C$ will have $m+1$ bits and must be right shifted by 1 bit. If the rounding logic decides to increase 1 in the least significant bit of the right shifted result, it means add 2 into $A+C$. When the operands have the same exponent and the operation of the mantissa adder is subtraction, the outputs of the adder could be negative. The ones complementer is used to adjust them to be positive. Then, one of these outputs is selected by the *GRS* unit to account for rounding. The *GRS* unit also computes the true guard (*G*), round(*R*), sticky (*S*) bits and the bit to be left shifted into the result during normalization. When the operands are close (the exponent difference is 0, 1, or -1) and the operation of the mantissa adder is subtraction, the result may need a massive left shift for normalization. The amount of left shift is predicted by the leading zero anticipator (*LZA*) unit in parallel with the mantissa adder. The predicted amount may differ by one from the correct amount, but this 1 bit shift is made up by a 1-bit *fine adjust* unit. Finally, one of the four possible results is selected to yield the final, rounded, and normalized result based on the outputs of the *path select* and *GRS* units.

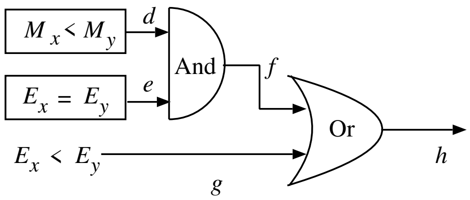


Figure 3: **Detail circuit of the compare unit**

As an alternative to the SNAP design, the ones complementer after the mantissa adder can be avoided, if we ensure that input C of the mantissa adder is smaller than or equal to input A, when the exponent difference is 0 and the operation of mantissa adder is subtraction. To ensure this property, a mantissa comparator and extra circuits, as shown in [23], are needed to swap the mantissas correctly. Figure 2.b shows a variant of the SNAP FP adder with this modification (the *compare* unit is added and the ones complementer is deleted). This $compare$ unit exists in many modern high-speed FP adder designs [23] and makes the verification harder described in Section 5.4. Figure 3 shows the detailed circuit of the *compare* unit which generates the signal to swap the mantissas. The signal $E_x < E_y$ comes from the exponent subtractor. When $E_x < E_y$ or $E_x = E_y$ and $M_x < M_y$ (i.e., h =1), A is M_y (i.e. the mantissas are swapped). Otherwise, A is M_x.

4 Specifications of FP Adders

In this section, we focus on the general specifications of the FP adder, especially when both operands have denormal or normal values. For the cases in which at least one of operands is a NaN or ∞, the specifications can be easily written at the bit level. For example, when both operands are NaN, the expected output is NaN (i.e. the exponent is all 1s and the mantissa is not equal to zero). The specification can be expressed as the "AND" of the exponent output bits is 1 and the "OR" of the mantissa output bits is 1.

When both operands have normal or denormal values, the ideal specification is $OUT = Round(X+Y)$. However, FP addition has exponential complexity with the word size of the exponent part for *PHDD. Thus, the specification must be divided into several sub-specifications for verification. According to the signs of both operands, the function $X+Y$ can be rewritten as Equation 3.

$$X+Y \quad = \quad (-1)^{S_x} \times \left\{ \begin{array}{ll} (2^{E_x-B} \times M_x + M_y \times 2^{E_y-B}) & S_x = S_y \ (true\ addition) \\ (2^{E_x-B} \times M_x - M_y \times 2^{E_y-B}) & S_x \neq S_y \ (true\ subtraction) \end{array} \right. \tag{3}$$

Similarly, for FP subtraction, the function $X - Y$ can be also rewritten as true addition when both operands have different signs and true subtraction when both operands have the same sign.

4.1 True Addition

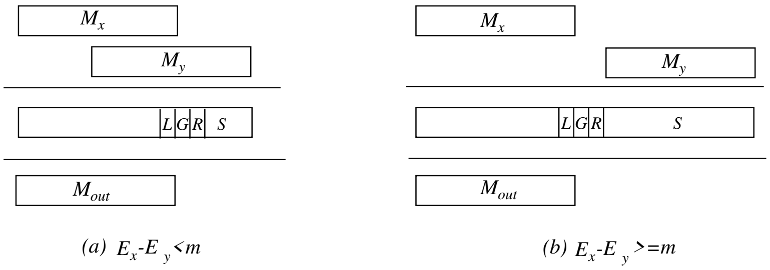


Figure 4: **Cases of true addition for the mantissa part.**

The *PHDDs for the true addition and subtraction still grow exponentially. Based on the sizes of the two exponents, the function $X + Y$ for true addition can be rewritten as Equation 4.

$$X + Y = (-1)^{S_x} \times \begin{cases} 2^{E_x - B} \times (M_x + (M_y >> (E_x - E_y))) & E_y \le E_x \\ 2^{E_y - B} \times (M_y + (M_x >> (E_y - E_x))) & E_y > E_x \end{cases} \quad (4)$$

When $E_y \le E_x$, the exponent is E_x and the mantissa is the sum of M_x and M_y right shifted by $(E_x - E_y)$ bits (i.e. $M_y >> (E_x - E_y)$ in the equation). $E_x - E_y$ can range from 0 to $2^n - 2$, but the number of mantissa bits in FP format is only m bits. Figure 4 illustrates the possible cases of true addition for $E_y \le E_x$ based on the values of $E_x - E_y$. In Figure 4.a, for $0 \le E_x - E_y < m$, the intermediate (precise) result contains more than m bits. The right portion of the result contains L, G, R and S bits, where L is the least signification bit of the mantissa. The rounding mode will use these bits to perform the rounding and generate the final result(M_{out}) in m-bit format. When $E_x - E_y \ge m$ as shown in Figure 4.b, the right shifted M_y only contributes to the intermediate result in the G, R and S bits. Depending the rounding mode, the output mantissa will be M_x or $M_x + 1 * 2^{-m+1}$. Therefore, we only need one specification in each rounding mode for the cases $E_x - E_y > m$. A similar analysis can be applied to the case $E_y > E_x$. Thus, the specifications for true addition with rounding can be written as:

$$\begin{cases} C_{a1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x + (M_y >> i))) & 0 \le i < m \\ C_{a2} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x + (M_y >> m))) & i \ge m \\ C_{a3}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_y - B} \times (M_y + (M_x >> i))) & 0 < i < m \\ C_{a4} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_y - B} \times (M_y + (M_x >> m))) & i \ge m \end{cases} \quad (5)$$

where $C_{a1}[i]$, C_{a2}, $C_{a3}[i]$ and C_{a4} are the conditions $Cond\_add \& E_x = E_y + i$, $Cond\_add \& \ E_x \ge E_y + m$, $Cond\_add \& E_y = E_x + i$, and $Cond\_add \& E_y \ge E_x + m$, respectively. $Cond\_add$ represents the condition for true addition and exponent range (i.e. normal and denormal numbers only). OUT is composed from the outputs S_{out}, E_{out} and M_{out}. Conditions $E_x - E_y = i$ and $E_x - E_y \ge m$ are represented by $2^{E_x} = 2^{E_y + i}$ and $2^{E_x} \ge 2^{E_y + m}$. Both sets of variables must use Shannon decomposition to represent the FP function efficiently in [6]. With this decomposition, the graph sizes of E_x and E_y are exponential in *PHDDs, but 2^{E_x} and 2^{E_y} will have linear size. While building BDDs and *PHDDs for OUT from the circuit, the function on left side of $\Rightarrow$ will be used to simplify the BDDs automatically by conditional forward simulation.

The number of specifications for true addition is $2m+1$. For instance, the value of m for IEEE double precision is 53, thus the number of specifications for true addition is 107. Since the specifications are very similar to one another, they can be generated by a looping construct in word-level SMV.

4.2 True Subtraction

The specification for true subtraction can be divided into two cases: *far* ($|E_x - E_y| > 1$) and *close* ($E_x - E_y$=0,1 or -1). For the *far* case, the result of mantissa subtraction does not require a massive left shift (i.e., LZA is not active). Similar to the true addition, the specifications for true subtraction can be written as Equation 6.

$$\begin{cases} C_{s1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> i))) & 2 \leq i \leq m \\ C_{s2} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> m))) & i > m \\ C_{s3}[i] \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> i))) & 2 \leq i \leq m \\ C_{s4} \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> m))) & i > m \end{cases} \tag{6}$$

where $C_{s1}[i], C_{s2}, C_{s3}[i]$ and C_{s4} are $Cond\_sub \& E_x = E_y + i$, $Cond\_sub\ \& E_x > E_y + m$, $Cond\_sub \& E_y = E_x + i$, and $Cond\_sub \& E_y > E_x + m$, respectively. $Cond\_sub$ represents the condition for true subtraction.

For the *close* case, the difference of the two mantissas may generate some leading zeroes such that normalization is required to product a result in IEEE format. For example, when $E_x - E_y$ = 0, $M_x - M_y$=0.0...01 must be left shifted by $m-1$ bits to 1.0...00. The number bits to left shift is computed in the LZA circuit and fed into the left shifter to perform normalization and into the subtractor to adjust the exponent. The number of bits to be left shifted ranges from 0 to m and is a function of M_x and M_y. The combination of left shifting and mantissa subtraction make the *PHDDs become irregular and grow exponentially. Therefore, the specifications for these cases must be divided further to take care of the exponential growth of *PHDD sizes.

Based on the number of leading zeroes in the intermediate result of mantissa subtraction, we divide the specifications for the true subtraction *close* case as shown in Equation 7.

$$\begin{cases} C_{c1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> 1))) & 0 \leq i < m \\ C_{c2}[i] \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> 1))) & 0 \leq i < m \\ C_{c3}[i]) \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - M_y)) & 1 \leq i < m \\ C_{c4}[i]) \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - M_x)) & 1 \leq i < m \end{cases} \tag{7}$$

where where $C_{c1}[i], C_{c2}[i]$,$C_{c3}[i]$, and $C_{c4}[i]$ are $Cond\_sub\ \&\ E_x = E_y + 1\ \&\ LS[i]$, $Cond\_sub\ \& E_y = E_x + 1\ \&\ LS[i]$, $Cond\_sub \& E_x = E_y \& M_x > M_y\ \&\ LS[i]$, and $Cond\_sub \&\ E_y = E_x + 1 \& M_x < M_y\ \&\ LS[i]$, respectively. $LS1[i], LS2[i], LS3[i]$ and $LS4[i]$ represent the conditions that the intermediate result has i leading zeroes to be left shifted. $LS1[i]$, $LS2[i]$, $LS3[i]$ and $LS4[i]$ are computed by $2^{m-i-1} \leq M_x - (M_y >> 1) < 2^{53-i}$, $2^{m-i-1} \leq M_y - (M_x >> 1) < 2^{m-i}$, $2^{m-i-1} \leq M_x - M_y < 2^{m-i}$, and $2^{m-i-1} \leq M_y - M_x < 2^{m-i}$), respectively. A special case is that the output is zero when E_x is equal to E_y and M_x is equal to M_y. The specification is as follows: $(Cond\_sub\ \& E_x = E_y\ \& M_x = M_y) \Rightarrow OUT = 0$.

4.3 Specification Coverage

Since the specifications of floating-point adders are split into several hundreds of sub-specifications, do these sub-specifications cover the entire input space? To answer this questions, someone would suggest to use theorem provers to handle the case splitting. In contrast, we propose a BDD approach to compute the coverage of our specifications.

Our approach is based on this observation that our specifications are in the form "$cond \Rightarrow out = expected\_result$" and $cond$ is only dependent on the inputs of the circuits. Thus, the union of the $conds$

of our specifications, which can be done by BDD operations, must be TRUE when our specifications cover the entire input space. In other words, the union of the $cond$s can be used to compute the percentage of input space covered by our specifications and to generate the cases which are not covered by our specifications.

5 Verification System: Extended Word-Level SMV with *PHDDs

Model checking is a technique to determine which states satisfy a given temporal logic formula for a given state-transition graph. In SMV [19], BDDs are used to represent the transition relations and set of states. The model checking process is performed iteratively on these BDDs. SMV has been widely used to verify control circuits in industry, but for arithmetic circuits, particularly for ones containing multipliers, the BDDs grows too large to be tractable. Furthermore, expressing desired behavior with Boolean formulas are not appropriated.

To verify arithmetic circuits, word-level SMV [9] with HDDs extended SMV to handle word level expressions in the specification formulas. In word-level SMV, the transition relation as well as those formulas that do not involve words are represented using BDDs. HDDs are used only to compute word-level expressions such as addition and multiplication. When a relational operation is performed on two HDDs, a BDD is used to represent the set of assignments that satisfies the relation. The BDDs for temporal formulas are computed in the same way as in SMV. For example, the evaluation the formula $AG(R = A + B)$, where R, A and B are word-level functions and AG is a temporal operator, is performed by first computing the HDDs for R, A, B and $A + B$, then generating BDDs for the relation $R = A + B$, and finally applying the AG operator to these BDDs. The reader can refer to [9] for the details of word-level SMV.

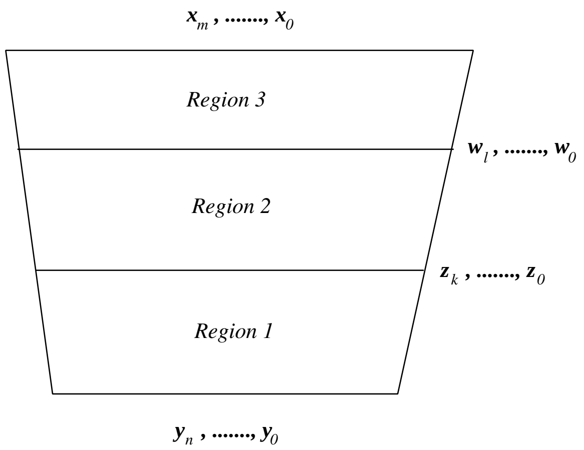


Figure 5: **Horizontal division of a combinational circuit,**

We have integrated *PHDDs into word-level SMV and introduced relational operators for floating-point numbers. As in word-level SMV, only the word-level functions are represented by *PHDDs and the rest of the functions are represented by BDDs.

Zhao's thesis [24] describes the layering backward substitution, a variant of Hamaguchi's backward substitution approach [15], although the public released version of word-level SMV does not implement this feature. We have implemented this feature in our system. The main idea of layering backward substitution is to virtually cut the circuit horizontally by introducing auxiliary variables to avoid the explosion of BDDs while symbolic evaluating bit level circuits. Figure 5 shows a horizontal division of a combinational circuit with primary inputs $x_0, \ldots, x_m$ and outputs $y_0, \ldots, y_n$. For $0 \leq i \leq n$, $y_i = f_i(x_0, \ldots, x_m)$ where f_i is a Boolean function, but it may not be feasible to be represented as a BDD. The circuit is divided into several layers by declaring some of the internal nodes as auxiliary variables. In this example, $y_i = f_{1i}(z_0, \ldots, z_k)$; $z_i = f_{2i}(w_0, \ldots, w_l)$; and $w_i = f_{3i}(x_0, \ldots, x_m)$. Since each f_{ji} is simpler than f_i, the BDD sizes to represent them are generally much smaller. When we try to compute *PHDD representation of the word $(y_0, \ldots, y_n)$ in terms of the variables $x_0, \ldots, x_m$, we first compute the *PHDD representation of the word in terms of variables $z_0, \ldots, z_k$ as $F = \sum_{i=0}^{N} 2^i \times f_{1i}(z_0, \ldots, z_k)$. Then we replace each z_i, one at a time, by $f_{2i}(w_0, \ldots, w_l)$. After this, we have obtained the *PHDD representation for the word in terms of variables $w_0, \ldots, w_l$. Likewise, we can replace each w_i by $f_{3i}(x_0, \ldots, x_n)$. In this way, the *PHDD representation of the word in terms of primary input can be computed without building BDDs for each output bit.

The main drawback of the backward substitution is that users still need to provide the information about the auxiliary variables (i.e., the virtual cuts). Another drawback is that the *PHDDs may grow exponentially during the substitution process, since the auxiliary variables may generalize the circuit behavior for some regions. For example, suppose that the internal nodes z_k and z_{k-1} under the original circuit have the relation that both of them can not be 0 at the same time and that the circuit of region 1 can only handle this case. After introducing the auxiliary variables, variables z_k and z_{k-1} can be 0 simultaneously. Hence, the word-level function F represents the function more general than the original circuit of region 1. This generalization may cause the *PHDD for F to blowup.

5.1 Conditional Symbolic Simulation

To overcome these drawbacks, we introduced a conditional symbolic simulation technique into word-level SMV. Symbolic simulation [3] performs the simulation with inputs having symbolic values (i.e., Boolean variables or Boolean functions). The simulation process builds BDDs for the circuits. If each input is a Boolean variable, this approach may cause the explosion of BDD sizes in the middle of the process, because it tries to simulate the entire circuit for all possible inputs at once. The concept of conditional symbolic simulation is to perform the simulation process under a restricted condition, expressed as a Boolean function over the inputs.

In [17], Jain and Gopalakrishnan encoded the conditions together with the original inputs as new inputs to the symbolic simulator using a parametric form of Boolean expressions, but it is hard to incorporate this approach into word-level SMV. Our approach is to apply the conditions directly during the symbolic simulation process. Right after building the BDD for a circuit gate, the condition is used to simplify the BDDs using the *restrict* [12] algorithm. Then, the simplified BDD is used as the input function for the gates connected to this one. This process is repeated until the outputs are reached. This approach can be viewed as dynamically extracting the circuit behavior under the specified condition without modifying the actual circuit.

5.2 Equalities and Inequalities with Conditions

To verify arithmetic circuits, it is very useful to compute the set of assignments that satisfy $F \sim G$, where F and G are word level functions represented by HDDs or *PHDDs, and $\sim$ can be any one of $=, \neq, \leq, \geq, <, >$. In general, the complexity of this problem is exponential. However, Clarke, *et al.* presented a branch-bound algorithm to efficiently solve this problem for a special class of HDDs, called linear expression functions using the positive Davio decomposition [8]. The basic idea of their algorithm is first to compute $H = F - G$ and then to compute the set of assignments satisfying $H \sim 0$ using branch-and-bound approach. The complexity of subtracting two HDDs is O($|F|\times|G|$) This algorithm only works well for the special class of HDDs (i.e., linear expression functions). However, the complexity of this algorithm for other classes of HDDs or *PHDDs can grows exponentially. In the verification of arithmetic circuits, HDDs and *PHDDs are not always in the class of linear expression functions. Thus, the $H \sim 0$ operations can not be computed for most cases. For example, "$OUT = Round(...)$" in some specifications of FP adders in Section 4.2 can not be finished after several CPU hours.

bdd cond_equal_0($< w_h, h >$, $cond$)
if $cond$ is FALSE, return FALSE;
if $< w_h, h >$ is a terminal node, return ($< w_h, h >$)= 0 ? TRUE : FALSE;
if the operation (cond_equal_0,$< w_h, h >$,$cond$) is in computed cache,
    return result found in cache;
$\tau \leftarrow$ top variable of h and $cond$;
$< w_{h_0}, h_0 >, < w_{h_1}, h_1 > \leftarrow$ 0- and 1-branch of $< w_h, h >$ with respect to variable τ ;
$cond_0, cond_1 \leftarrow$ 0- and 1-branch of $cond$ with respect to variable τ ;
bound_value($< w_{h_0}, h_0 >, upper_{h_0}, lower_{h_0}$); bound_value($< w_{h_1}, h_1 >, upper_{h_1}, lower_{h_1}$);
if (τ uses the Shannon decomposition) {
    if ($upper_{h_0} < 0 || lower_{h_0} > 0$) $res_0 \leftarrow$ FALSE;
    else $res_0 \leftarrow$ cond_equal_0($< w_{h_0}, h_0 >$,$cond_0$);
    res_1 is computed similar to res_0;
} else if (τ uses the positive Davio decomposition) {
    res_0 is computed the same as res_0 in Shannon decomposition;
    $upper_{h_1} \leftarrow upper_{h_1} + upper_{h_0}$; $lower_{h_1} \leftarrow lower_{h_1} + lower_{h_0}$;
    if ($upper_{h_1} < 0 || lower_{h_1} > 0$) $res_1 \leftarrow$ FALSE;
    else if ($cond_1$ is FALSE) $res_1 \leftarrow$ FALSE;
    else {
        $< w_{h_1}, h_0 > \leftarrow$ addition($< w_{h_1}, h_1 >, < w_{h_0}, h_0 >$);
        $res_1 \leftarrow$ cond_equal_0($< w_{h_1}, h_1 >$, $cond_1$);
    }
} else if (τ uses the negative Davio decomposition) {
    res_0 and res_1 computation are similar to them in positive Davio decomposition.
}
$result \leftarrow$ find BDD node (τ, res_0, res_1) in unique table, or create one if not exists;
insert (cond_equal_0, $< w_h, h >$, $cond$, *result*) into the computed cache;
return *result*;

Figure 6: **algorithm for H = 0 with conditions.** $H =< w_h, h >$.

To solve this problem, we introduce the relational operations with conditions, since the equality and inequality in our specifications must be hold only under the conditions. These operations take three arguments F, G and $cond$, where F and G are word level functions and $cond$ is a Boolean function. First, it computes $H = F - G$ and then computes the set of assignments satisfying $H \sim 0$ under the condition $cond$. For example, the algorithm for $H = 0$ under the condition $cond$ is given in Figure 6. This algorithm produces the BDDs satisfying $H = 0$ under the condition $cond$, and is similar to the algorithm in [8], except that it takes an extra BDD argument for the condition and uses the condition to stop the equality checking of the algorithm as soon as possible. To be conservative, when the condition is false, the returned result is false. In line 1, the condition is used to stop this algorithm, when the condition is false. In line 16, the condition is also used to stop the addition of two *PHDDs and the further equality checking in lines 18 and 19, respectively. The efficiency of this algorithm will depend on the BDDs for the condition. If the condition is always true, then this algorithm has the same behavior as Clarke's algorithm. If the condition is always false, then this algorithm will immediately return false regardless of how complex the *PHDD is. This new algorithm has reduced the computation time dramatically for the specifications in Section 4.2.

5.3 Equalities and Inequalities

The efficiency of Clarke's algorithm for relational operations of two HDDs depends on the complexity of computing $H = F - G$. The complexity of subtracting two HDDs is O($|F| \times |G|$), and similar algorithms can be used for these relational operators with *PHDDs. However, the complexity of subtracting two *PHDDs using disjunctive sets of supporting variables may grow exponentially. For example, the complexity of subtraction of two FP encodings represented by *PHDDs grows exponentially with the word size of exponent part [6]. Thus, Clarke's algorithm is not suitable for these operators with two *PHDDs having disjunctive sets of supporting variables. These operators are commonly used in our specifications of FP adders in Section 4.2 and 4.1.

We have developed algorithms for these relational operators with two *PHDDs having disjunctive sets of supporting variables. Figure 7 shows the new algorithm for computing BDDs for the set of assignments that satisfy $F > G$. Similar algorithms are used for other relational operators. The main concept of this algorithm is to directly apply the branch-and-bound approach without performing a subtraction, whose complexity could be exponential. First, if both arguments are constant, the algorithm returns the comparison result of the arguments. In line 2 and 3, weights w_f and w_g are adjusted by the minimum of them to increase the sharing of the operations, since $(2^{w_f} \times f) > (2^{w_g} \times g)$ is the same as $(2^{w_f - min} \times f) > (2^{w_g - min} \times g)$, where min is the minimum of w_f and w_g. Line 4 checks whether the comparison is in the computed cache and returns the result if it is found. In line 5 to 7, the top variable τ is chosen and the 0- and 1-branches of f and g are computed. In lines 8 and 9, function $bound\_value$ is used to compute the upper and lower bounds of these four sub-functions, The algorithm of $bound\_value$ is similar to that described in [8], except edge weights are handled. The complexity of $bound\_value$ is linear in the graph size. When τ uses the Shannon decomposition, lines 11 and 12 try to bound and finish the search for the 0-branch. If it is not successful, line 13 recursively calls this algorithm for 0-branch. The 1-branch is handled in a similar way. When τ uses the positive Davio decomposition, the computation for 0-branch is the same as that in Shannon decomposition. since $< w_{f_1}, f_1 >$ is the linear moment of $< w_f, f >$ and the 1-cofactor of $< w_f, f >$ is equal to $< w_{f_1}, f_1 > + < w_{f_0}, f_0 >$, the lower(upper) bound of the 1-cofactor of $< w_f, f >$ is bounded by the sum of lower (upper) bounds of $< w_{f_1}, f_1 >$ and $< w_{f_0}, f_0 >$. For the 1-branch, new upper and lower bounds for the 1-cofactors are recomputed in lines 17 and 18. In lines 19 and 20, new upper and lower bounds are used to bound and stop the further checking for 1-cofactor. If it is not successful, lines 21-24 add the constant and linear moments to get the 1-cofactors and recursively call this algorithm for the 1-cofactor case. For the negative Davio decomposition, the 0- and 1-branches are handled similar to the positive Davio decomposition. After generating res_0 and res_1 for 0- and 1-cofactors, the result BDD is built and this computed operation is inserted to the computed cache for future lookups.

bdd greater_than($< w_f, f >$, $< w_g, g >$)
if both $< w_f, f >$ and $< w_g, g >$ are terminal nodes,
return (($< w_f, f >$) $>$ ($< w_g, g >$)) ? TRUE : FALSE;
$min \leftarrow$ minimum(w_f, w_g);
$w_f \leftarrow w_f -$ min; $w_g \leftarrow w_g -$ min;
if the operation (greater_than, $< w_f, f >$, $< w_g, g >$) is in computed cache,
return *result* found in cache;
$\tau \leftarrow$ top variable of f and g
$< w_{f_0}, f_0 >, < w_{f_1}, f_1 > \leftarrow$ 0- and 1-branch of $< w_f, f >$ with respect to variable τ ;
$< w_{g_0}, g_0 >, < w_{g_1}, g_1 > \leftarrow$ 0- and 1-branch of $< w_g, g >$ with respect to variable τ ;
bound_value($< w_{f_0}, f_0 >, upper_{f_0}, lower_{f_0}$); bound_value($< w_{g_0}, g_0 >, upper_{g_0}, lower_{g_0}$);
bound_value($< w_{f_1}, f_1 >, upper_{f_1}, lower_{f_1}$); bound_value($< w_{g_1}, g_1 >, upper_{g_1}, lower_{g_1}$);
if (τ uses the Shannon decomposition) {
if ($upper_{f_0} \leq lower_{g_0}$) $res_0 \leftarrow$ FALSE;
else if ($lower_{f_0} > upper_{g_0}$) $res_0 \leftarrow$ TRUE;
else $res_0 \leftarrow$ greater_than($< w_{f_0}, f_0 >$,$< w_{g_0}, g_0 >$);
res_1 is computed similar to res_0;
} else if (τ uses the positive Davio decomposition){
res_0 is computed the same as res_0 in Shannon decomposition;
$upper_{f_1} \leftarrow upper_{f_1} + upper_{f_0}$; $upper_{g_1} \leftarrow upper_{g_1} + upper_{g_0}$;
$lower_{f_1} \leftarrow lower_{f_1} + lower_{f_0}$; $lower_{g_1} \leftarrow lower_{g_1} + lower_{g_0}$;
if ($upper_{f_1} \leq lower_{g_1}$) $res_1 \leftarrow$ FALSE;
else if ($lower_{f_1} > upper_{g_1}$) $res_1 \leftarrow$ TRUE;
else {
$< w_{f_1}, f_1 > \leftarrow$ addition($< w_{f_1}, f_1 >$, $< w_{f_0}, f_0 >$);
$< w_{g_1}, g_1 > \leftarrow$ addition($< w_{g_1}, g_1 >$, $< w_{g_0}, g_0 >$);
$res_1 \leftarrow$ greater_than($< w_{f_1}, f_1 >$,$< w_{g_1}, g_1 >$);
}
} else if (τ uses the negative Davio decomposition){
res_0 and res_1 are computed similar to positive Davio decomposition.
}
$result \leftarrow$ find BDD node (τ, *res*$_0$, *res*$_1$) in unique table, or create one if not exists.
insert (greater_than, $< w_f, f >$, $< w_g, g >$,*result*) into the computed cache
return *result*;

Figure 7: **Improved algorithm for $F > G$.** $F =< w_f, f >$ and $G =< w_g, g >$.

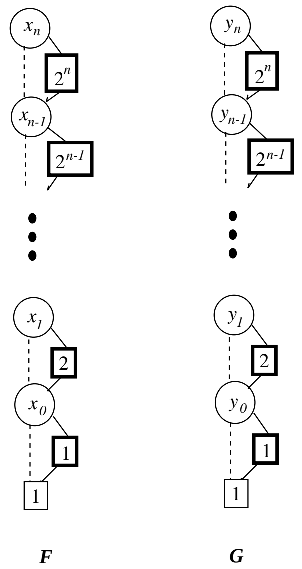


Figure 8: ***PHDDs for F and G.**

This algorithm works very well for two *PHDDs with disjunctive set of supporting variables, while Clarke's algorithm has exponential complexity. For example, let $F = \prod_{i=0}^{n} 2^{2^i \times x_i}$ and $G = \prod_{i=0}^{n} 2^{2^i \times y_i}$. The variable ordering is x_n, y_n, ..., x_0, y_0 and all variables use the Shannon decomposition. The *PHDDs for F and G have the structure shown in Figure 8. It can be proven that the complexity of this algorithm for this type of function is O(n) if the computed cache is a complete cache.

5.4 Short-Circuiting Technique

Can we verify the specifications of FP adders by conditional forward simulation? In our experience, all specifications for the FP adder design without a mantissa comparator, as in Figure 2.a, can be verified by conditional forward simulation, but not so for the FP adder containing a mantissa comparator, as in Figure 2.b. This is caused by a conflict of variable orderings for the mantissa adder and the mantissa comparator, which generates the signal $M_x < M_y$ (i.e. signal d in Figure 3). The best variable ordering for the comparator is to interleave the two vectors from the most significant bit to the least significant bit (i.e., x_{m-1}, y_{m-1}, ..., x_0, y_0). Table 2 shows the CPU time in seconds and the BDD size of the signal d under different variable orderings, where ordering offset represents the number of bit offset from the best ordering. For example, the ordering is x_{m-1}, ..., x_{m-6}, y_{m-1}, x_{m-7}, y_{m-2}, ..., x_0, y_5, ..., y_0, when the ordering offset is 5. Clearly, the BDD size grows exponentially with the offset. In contrast to the comparator, the best ordering for the mantissa adder is x_{m-1}, ..., x_{m-k-1}, y_{m-1}, x_{m-k-2}, y_{m-2}, ..., x_0, y_k, ..., y_0, when the exponent difference is k. We observed that the best ordering for the specification represented by *PHDDs is the same ordering as the best ordering for the mantissa adder. Thus, the extended word-level SMV can not build the BDDs for both the mantissa comparator and mantissa adder by conditional forward simulation, when the exponent difference is large.

Let us examine the *compare* unit carefully. We find that the signal d is used only when E_x = E_y. In other words, it is not necessary to build the BDDs for it, when $|E_x - E_y|$ is greater than 0. Based on this fact, we introduce a short-circuiting technique to eliminate unnecessary computations as early as possible. The word-level SMV and *PHDD packages are modified to incorporate this technique. In the *PHDD package, the BDD operators, such as *And* and *Or*, are modified to abort the operation and return a *special token* when

Ordering Offset	BDD Size	CPU Time (Sec.)
0	157	0.68
1	309	0.88
2	608	1.35
3	1195	2.11
4	2346	3.79
5	4601	7.16
6	9016	13.05
7	17655	26.69
8	34550	61.61
9	67573	135.22
10	132084	276.23

Table 2: **Performance measurements of a 52-bit comparator with different orderings.**

the number of newly created BDD nodes within this BDD call is greater than a size threshold. In word-level SMV, for an *And* gate with two inputs, if the first input evaluates 0, 0 will be returned without building the BDDs for the second input. Otherwise, the second input will be evaluated. If the second input evaluates to 0 and the first input evaluates to a *special token*, 0 is returned. Similar technique is applied to *Or* gates with two inputs. *Nand*(*Nor*) gates can be decomposed into *Not* and *And* (*Or*) gates and use the same technique to terminate earlier. For other logic gates with two inputs, the result is a *special token*, if any of the inputs evaluates to a *special token*. If the *special token* is propagated to the output of the circuit, then the size threshold is doubled and the output is recomputed. This process is repeated until the output BDD is built. For example, when the exponent difference is 30, the size threshold is 10000, the ordering is the best ordering of mantissa adder, and the evaluation sequence of the *compare* unit shown in Figure 3 is d, e, f, g and h, the values of signals d, e, f, g and h will be *special token*, 0, 0, 1, and 1, respectively, by conditional forward simulation. With these modification, the new system can verify all of the specifications for both types of FP adders by conditional forward simulation. We believe that this short-circuiting technique can be generalized and used in the verification which only exercises part of the circuits.

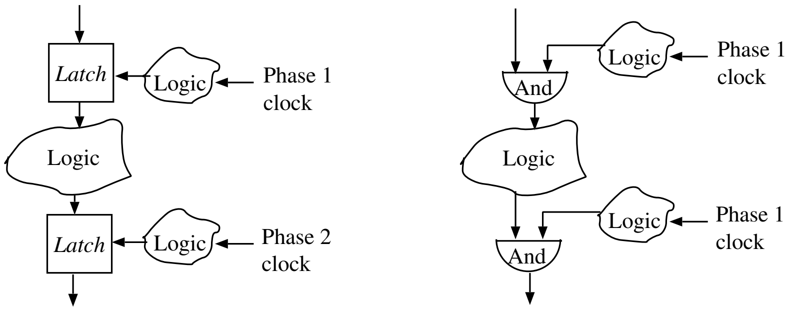
6 Verification of FP Adders

In this section, we used the FP adder in the Aurora III Chip [16], designed by Dr. Huff as part of his PhD dissertation at the University of Michigan, as an example to illustrate the verification of FP adders. This adder is based on the same approach as the SNAP FP adder [20] at Stanford University. Dr. Huff found several errors with the approach described in [20]. This FP adder only handles operands with normal values. When the result is a denormal value, it is truncated to 0. This adder supports IEEE double precision format and the 4 IEEE rounding modes. In this verification work, we verify the adder only in round to nearest mode, because we believe that the round to nearest mode is the hardest one to verify. All experiments were carried out on a Sun 248 MHz UltraSPARC-II server with 1.5 GB memory.

The FP adder is described in the Verilog language in a hierarchical manner. The circuit was synthesized into flattened, gate-level Verilog, which contains latches, multiplexors, and logic gates, by Dr. John Zhong at SGI. Then, a simple Perl script was used to translate the circuit from gate-level Verilog to SMV format.

6.1 Latch Removal

Huff's FP adder is a pipelined, two phase design with a latency of three clock cycles. We handled the latches during the translation from gate-level Verilog to SMV format. Figure 9.a shows the latches in the pipelined, two phase design. In the design, phase 2 clock is the complement of the phase 1 clock. Since we only verify the functional correctness of the design and the FP adder does not have any feedback loops, the latches can be replaced by And gates, as shown in Figure 9.b, without losing the functional behavior of the circuit. Since phase 2 clock is the complement of the phase 1 clock, we must replace the phase 2 clock by the phase 1 clock. Otherwise the circuit behavior will be incorrect.



(a) (b)

Figure 9: **Latch Removal.** (a) The pipelined, two phase design. (b) The design after latch removal.

6.2 Design with Bugs

In this section, we describe our experience with the verification of a FP adder with design errors. During the verification process, our system found several design errors in Huff's FP adder. These errors were not caught by random simulation performed by Dr. Huff.

The first error we found is the case when $A + C$ = 01.111...11, $A + C$ + 1=10.000...00, and the rounding logic decides to add 1 to the least significant bit (i.e., the result should be $A + C$ + 1), but the circuit design outputs A+C as the result. This error is caused by the incorrect logic in the *path select* unit, which categorized this case as a no shift case instead of a right shift by 1. While we were verifying the specification of true addition, our system generated a counterexample for this case in around 50 seconds. To ensure that this bug is not introduced by the translation, we have used Cadence's Verilog simulation to verify this bug in the original design by simulating the input pattern generated from our system. Another design error we found is in the sticky

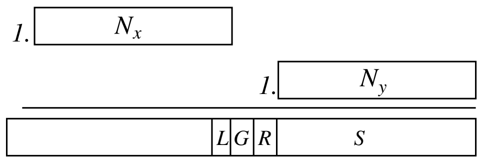


Figure 10: **Sticky bit generation, when $E_x - E_y$ = 54.**

bit generation. The sticky bit generation is based on the table given in page 10 of Quach's paper describing the SNAP FP adder [20]. The table only handles cases when the absolute value of the exponent difference is less than 54. The sticky bit is set 1 when the absolute value of the exponent difference is greater than 53 (for normal numbers only). The bug is that the sticky bit is not always 1 when the absolute value of the exponent difference is equal to 54. Figure 10 shows the sticky bit generation when $E_x - E_y$ = 54. Since N_x has 52 bits, the leading 1 will be the Round (R) bit and the sticky (S) bit is the OR of all of N_y bits, which may be 0. Therefore an entry for the case $|E_x - E_y| = 54$ is needed in the table of Quach's paper [20].

From our experience, the design in the mantissa path doesn't cause the *PHDD explosion problem. However, when the error is in the exponent path, the *PHDD may grow exponentially while building the output. A useful tip to overcome the *PHDD explosion problem is to reduce the exponent value to a smaller range by changing the exponent range condition in $Cond\_add$ or $Cond\_sub$ in Equation 5, 6 or 7.

6.3 Corrected Designs

After identifying the bugs, we fixed the circuit in the SMV format. In addition, we created another FP adder by adding the *compare* unit in Figure 2.b into Huff's FP adder. This new adder is equivalent to the FP adder in Figure 2.b, since the *ones complement* unit will not be active at any time.

To verify the FP adders, we combined the specifications for both addition and subtraction instructions into the specification of true addition and subtraction. We use the same specifications to verify both FP adders. Table 3 shows the CPU time in seconds and the maximum memory required for the verification of both FP adders. The CPU time is the total time for verifying all specifications. For example, the specifications of true addition are partitioned into 18 groups and the specifications in the same group use the same variable ordering. The CPU time is the sum of these 18 verification runs. The FP adder II can not be verified by conditional forward simulation without the short-circuiting technique. The maximum memory is the maximum memory requirement of these 18 runs. For both FP adders, the verification can be done within two hours and requires less than 55 MB. Each individual specification can be verified in less than 200 seconds.

	FP adder I		FP adder II	
Case	CPU Time (Sec.)	Max. Memory(MB)	CPU Time (Sec.)	Max. Memory(MB)
True addition	3283	49	3329	55
True subtraction(*far*)	2654	35	2668	35
True subtraction (*close*)	994	53	1002	48

Table 3: **Performance measurements of verification of FP adders.** FP adder I is Huff's FP adder with bugs fixed. FP adder II is FP adder I with the *compare* unit in Figure 2.b. For true subtraction, *far* represent cases $|E_x - E_y| > 1$, and *close* represent cases $|E_x - E_y| \leq 1$.

In our experience, the decomposition type of the subtrahend's variables for the true subtraction cases is very important to the verification time. For the true subtraction cases, the best decomposition type of the subtrahend's variables is *negative* Davio decomposition. If the subtrahend's variables use the *positive* Davio decomposition, the *PHDDs for OUT can not be built after a long CPU time (> 4 hours).

As for the coverage, the verified specifications cover 99.78% of the input space for the floating-point adders in IEEE round-to-nearest mode. The uncovered input space (0.22%) is caused by the unimplemented circuits for handling the cases of any operands with denormal, NaN or ∞ values, and the cases where the result of the true subtraction is denormal value.

Our results should not be compared with the results in [7], since the FP adders handle difference precision (i.e., their adder handles IEEE extended double precision) and the CPU performance ratio of two different machines is unknown (they used a HP 9000 workstation with 256MB memory). Moreover, their approach partitioned the circuit into sub-circuits which are verified individually based on the assumptions about their inputs, while our approach is implementation-independent.

7 Conversion Circuits

The overflow flag erratum of the FIST instruction (floating-point to integer conversion) [14] in Intel's Pentium Pro and Pentium II processors has illustrated the importance of verification of the conversion circuits [16] which convert the data from one format to another format (e.g., IEEE single precision to double precision). These circuits are another common unit in floating-point processors. For example, the MIPS processor supports conversions between any of the three number formats: integer, IEEE single precision, and IEEE double precision.

We believe that the verification of the conversion circuits is much easier than the verification of floating-point adders, since these circuits are much simple than the floating-point adders and only have one operand(i.e. less variables than FP adders). For example, the specification of the double-to-single (D2S) operation, which converts the data from double precision to single precision, can be written as *"(overflow_flag = expected_overflow) & (not overflow_flag ⇒ (output = expected_output))"*, where *overflow_flag* and *output* are directly from the circuit as well as *expected_overflow* and *expected_output* are computed in terms of the inputs, since some of the numbers represented in double precision cannot be represented in single precision. For example, *expected_output* is computed by $Round((-1)^S \times M \times 2^{E-B})$. Similarly, *expected_overflow* can be computed from the inputs.

For another example, the specification of the single-to-double (S2D) operation, which converts the data from single precision to double precision, can be written as *"output = input"*, since every number represented in single precision can be represented in double precision without rounding(i.e. the output represents the exact value of input).

8 Conclusions and Future Work

We presented extensions to word-level SMV to enable the verification of floating-point adders with implementation-independent specifications. Word-level SMV were improved by using the Multiplicative Power HDD (*PHDD) representation, by deriving efficient algorithms for equality and inequality operations between two *PHDDs, and by incorporating conditional symbolic simulation as well as a short-circuiting technique. Based on the case analysis of the signs and the relations of two exponents, the specifications of floating-point adders are divided into several hundreds of implementation-independent sub-specifications.

Conditional forward simulation has the advantage of implementation-independent specifications. We identified a conflict in the variable orderings between the mantissa comparator and mantissa adder which prevents the use of conditional forward simulation. We presented a short-circuiting technique to solve this problem. This short-circuiting technique can be generalized and used in the verification which only exercises part of the circuits.

We used our system and the implementation-independent specifications to verify a FP adder from University of Michigan. Our system found several bugs in Huff's FP adder and generated counterexamples within several minutes. After fixing the bugs, a variant of the Aurora III FP adder is created by introducing a mantissa comparator and extra circuits for demonstrating the capability of our system to handle different FP adder designs. For each of FP adders, the verification task finished in 2 CPU hours on a Sun UltraSPARC-II server for IEEE double precision. The verified specifications covered 99.78% of the entire input space. The uncovered input space (0.22%) We believe that our system and specifications can be applied to directly verify FP adders and to help finding design errors.

The overflow flag erratum of the FIST instruction (floating-point to integer conversion) [14] in Intel's Pentium Pro and Pentium II processors has illustrated the importance of verification of the conversion circuits which convert the data from one format to another format (e.g., IEEE single precision to double precision). Since these circuits are much simpler than floating-point adders and only have one input operand, we believe that our system can be used to verify the correctness of these circuits. We plan to verify the conversion circuits in the Aurora III chip.

Acknowledgement

We thank Prof. Brown, Dr. Huff and Mr. Riepe at University of Michigan for providing us with Huff's FP adder and valuable discussions. We thank Dr. John Zhong at SGI for helping us to synthesize the Huff's FP adder design into flattened, gate-level Verilog. We thank Henry A. Rowley for providing us the simple Perl script.

References

[1] AAGAARD, M. D., AND SEGER, C.-J. H. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 7–10.

[2] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers* (August 1986), pp. 8:677–691.

[3] BRYANT, R. E., BEATTY, D. L., AND SEGER, C.-J. H. Formal hardware verification by symbolic ternary trajectory. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), pp. 397–402.

[4] 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* (June 1995), pp. 535–541.

[5] CARREñO, V. A., AND MINER, P. S. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *High Order Logic Theorem Proving and Its Applications* (September 1995).

[6] CHEN, Y.-A., AND BRYANT, R. E. *PHDD: An efficient graph representation for floating point circuit verification. In *Proceedings of the International Conference on Computer-Aided Design* (November 1997), pp. 2–7.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] COE, T. Inside the Pentium Fdiv bug. *Dr. Dobbs Journal* (April 1996), pp. 129–135.

[12] COUDERT, O., AND MADRE, J. C. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design* (November 1990), pp. 126–129.

[13] 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.

[14] FISHER, L. M. Flaw reported in new intel chip. *New York Times* (May 6 1997), D, 4:3.

[15] HAMAGUCHI, K., MORITA, A., AND YAJIMA, S. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 78–82.

[16] HUFF, T. R. Architectural and circuit issues for a high clock rate floating-point processor. *PhD Dissertation in Electrical Engineering Department, University of Michigan* (1995).

[17] JAIN, P., AND GOPALAKRISHNAN, G. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (August 1994), pp. 1005–1015.

[18] LEESER, M., AND O'LEARY, J. Verification of a subtractive radix-2 square root algorithm and implementation. In *Proceedings of 1995 IEEE Internaational Conference on Computer Design: VLSI in Computer and Processors* (October 1995), pp. 526–531.

[19] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] QUACH, N., AND FLYNN, M. Design and implementation of the SNAP floating-point adder. Tech. Rep. CSL-TR-91-501, Stanford University, December 1991.

[21] RUEß, H., SHANKAR, N., AND SRIVAS, M. K. Modular verification of SRT division. In *Computer-Aided Verification, CAV '96* (New Brunswick, NJ, July/August 1996), R. Alur and T. A. Henzinger, Eds., no. 1102 in Lecture Notes in Computer Science, Springer-Verlag, pp. 123–134.

[22] SHARANGPANI, H. P., AND BARTON, M. L. Statistical analysis of floating point flag in the pentium processor(1994). Tech. rep., Intel Corporation, November 1994.

[23] SUZUKI, H., MORINAKA, H., HIROSHI MAKINO, NAKASE, Y., MASHIKO, K., AND SUMI, T. Leading-zero anticipatory logic for high-speed floating point addition. *IEEE Journal of Solid-State Circuits* (August 1996), pp. 1157–1164.

[24] ZHAO, X. Verification of arithmetic circuits. Tech. Rep. CMU-CS-96-149, School of Computer Science, Carnegie Mellon University, 1996.