Arithmetic Circuit Verification Based on Word-Level Decision Diagrams

Yirng-An Chen

May, 1998

CMU-CS-98-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Randal E. Bryant, Chair
Edmund M. Clarke
Rob. A. Rutenbar
Xudong Zhao, Intel Corporation

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

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

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Semiconductor Research Corporation, or the U.S. Government.

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, theorem proving, Binary Decision Diagram, MTBDD, EVBDD, FDD, KFDD, K*BMD, Model Checking, SMV

Abstract

The division bug in Intel's Pentium processor has demonstrated the importance and the difficulty of verifying arithmetic circuits and the high cost of an arithmetic bug. In this thesis, we develop verification methodologies and symbolic representations for functions mapping Boolean vectors to integer or floating-point values, and build verification systems for arithmetic circuits.

Our first approach is based on a hierarchical methodology and uses multiplicative binary moment diagrams (*BMDs) to represent functions symbolically for verification of integer circuits. *BMDs are particularly effective for representing and manipulating functions mapping Boolean vectors to integer values. Our hierarchical methodology exploits the modular structure of arithmetic circuits to speed up the verification task. Based on this approach, we have verified a wide range of integer circuits such as multipliers and dividers.

Our *BMD-based approach cannot be directly applied to verify floating-point (FP) circuits. The first challenge is that the existing word-level decision diagrams cannot represent floating-point functions efficiently. For this problem, we introduce Multiplicative Power Hybrid Decision Diagrams (*PHDDs) to represent floating-point functions efficiently. *PHDDs explode during the composition of specifications in the rounding module in the hierarchical approach. To overcome this problem, we choose to verify flattened floating-point circuits by using word-level SMV with these improvements: *PHDDs, conditional symbolic simulation and a short-circuiting technique.

Using extended word-level SMV, FP circuits are treated as black boxes and verified against reusable specifications. The FP adder in the Aurora III Chip at the University of Michigan was verified. Our system found several errors in the design and generated a counterexample for each error. A variant of the corrected FP adder was created and verified to illustrate the ability of our system to handle different 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. We believe that our system can be used to verify the correctness of conversion circuits which translate data from one format to another.

Acknowledgements

This thesis would have not been possible without my advisor, Randy Bryant. Six years ago, when I came to CMU, I had no knowledge of Decision Diagrams and Formal Verification. The transition was difficult, but Randy made it easy for me. He patiently initiated me into Binary Decision Diagrams and Formal Verification; he wisely guided me through the maze of graduate studies and research; he ceaselessly gave me confidence when I doubted myself. How many times have I gone into his office frustrated and discouraged about my research and come out smiling!

My committee members, Edmund Clarke, Rob Rutenbar, and Xudong Zhao have given me much help in my research and in writing my thesis. The work I have done on word-level SMV benefits greatly from conversations with and course taught by Edmund Clarke. My three summer jobs at Intel Strategic CAD Labs with Xudong Zhao have, without any doubt, widened my view of formal verification, especially for industrial circuits. Rob Rutenbar led me to obtain the floating-point circuit from the University of Michigan. Prof. Brown and Mr. Riepe at the University of Michigan and Dr. Huff at Intel provided the floating-point adder circuit and valuable discussion about the design. Chapter 6 would not exist without Rutenbar's suggestion and Huff's circuit.

I have enjoyed the School of Computer Science at Carnegie Mellon University since the first day I was here. My wonderful officemates, Bwolen Yang, Henry Rowley and James Thomas made my office a pleasant place to be in and made my time at work more delightful. They have helped in almost every aspect of my daily needs in the department, from Unix commands to paper proofreading. I have also learned much from my former officemate Rujith S. DeSilva. Other friends in CS and ECE departments also made my graduate studies pleasant: Hao-Chi Wong, Karen Z. Haigh, Arup Mukherjee, Claudson F. Bornstein, Manish Pandey, Alok Jain, Shipra Panda, Vishnu A. Patankar, Miroslav N. Velev, Yuan Lu, and many more. Special thanks go to Sharon Burks and Joan Maddamma, who have always greeted me with a pleasant smile when I walked into their office. They have taken care of every administrative detail concerning me and much more.

Pittsburgh is the city where I have lived the longest except for my home town Chia-Yi in Taiwan. Throughout the years, I have grown to love this city, to love many people I have had the pleasure to share my time with. I thank my host family, Ms. Maxine Matz and her family, for sharing their Thanksgiving and Easter holidays with me in my study period. I also thank Mr. and Mrs. Aston and their family, for sharing their Christmas holidays with me. I enjoyed the friendship offered by International Bible study group, especially Jody Jackson, David Palmer, and George Mazariegos.

I also want to thank my parents, who taught me the value of hard work by their own example. I have always felt their love and support despite the fact that they are thousands of miles away.

Finally, I thank my lovely wife, Mei-Ling Liao, who accompanied me for the first four years and gave me her full support in my studies. I could not go through this PhD program without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Microprocessors have been widely used in digital systems such as workstations, personal computers, aircraft control systems. In AD 2010, microprocessors running at frequencies over 1GHz will contain 90 million logic transistors and several hundred million cache transistors according to the Roadmap of Semiconductors [5]. Thus, more function units, such as digital signal processors, 3D graphic and multimedia instructions, will be added into microprocessors. The intense competition in the microprocessor field is resulting in ever shorter design cycles. To achieve it, more designers are added into design teams. To amortize the costs of development and manufacturing, microprocessors go to mass production a very short time after their introduction. Thus, a design error in a microprocessor can have a severe financial cost (e.g. the $475 million cost of Pentium DIV bug [36, 92]) and may lead to serious injuries or even loss of life when used in life-support or control systems.

Proving the correctness of a microprocessor design is an important task. Simulation is the most popular verification technique in the industry. However, increasing complexity makes simulation insufficient to verify modern microprocessors. It is impossible to simulate all possible combinations and sequences of inputs. For example, the Pentium microprocessors have been tested by over 1 trillion of test vectors before production [29]. Gwennap [48] summarized the reported design bugs in Intel Pentium and Pentium Pro processors after mass production. Several bugs in the Pentium Pro processors can cause data corruption or system hangs and thus are visible to end users. Thus, the industry is interested in formal verification techniques for circuit designs.

Arithmetic circuits, such as the Arithmetic Logic Unit (ALU) and the Floating-Point Unit (FPU) are important parts of microprocessors. These circuits performs data operations such as addition, multiplication, division, etc. The verification of arithmetic circuits is an important part of verification of microprocessors. The goal of our work is to develop techniques which enable the formal verification of arithmetic circuits.

Section 1.1 discusses verification of arithmetic circuits. Section 1.2 briefly surveys work in the area of formal verification, especially verification of arithmetic circuits. The goals of the thesis are summarized in Section 1.3. Finally, section 1.4 discusses the thesis organization, and a summary of each chapter.

1.1 Verification of arithmetic circuits

Verification of arithmetic circuits has always been an import part of processor verification. In modern processors, arithmetic circuits contains ALUs, integer multipliers, integer dividers, floating-point (FP) adder, FP multipliers, FP dividers, FP square roots, and most of multimedia instructions. These circuits form an important part of microprocessors. Because of area and performance constraints, these circuits are not synthesized by automatic synthesis tools, rather they are custom designed. They can occupy as much as 20%-50% of the processor chip area. In Intel's Pentium II processor [51], these circuits occupied 20% of the chip area, and they accumulated up to 50% of the chip area in Sun's SuperSparc-2 processor [47].

The importance and difficulty of arithmetic circuit verification has been illustrated by the famous FDIV bug in Intel's Pentium processor which cost Intel $475 million. This bug was not covered by the one trillion test vectors used for this processor [29]. Traditional approaches to verifying arithmetic circuits are based on simulation or emulation. However, these approaches can not exhaustively cover the input space of the circuits. For example, the whole input space of each IEEE double precision floating-point circuit with one rounding mode is 2^{128} test vectors, which is impossible to simulate in practice. Theorem proving approaches require verification experts to manually guide the systems to complete the proof. Thus, to automatically verify arithmetic circuits, we need to employ a formal technique which can handle large circuits. Among all the formal verification technique, decision diagram approach comes closest to meeting these requirements. However, there are still many fundamental and pragmatic issues to be resolved. These issues include the explosion problem of decision diagrams, and specification and efficient verification of these circuits. We have addressed these issues in this thesis.

1.2 Related work

In this section, we summarize the research work on decision diagrams, which have been used in many applications such verification, logic synthesis. Then, we discuss different verification approaches which can be used to verify arithmetic circuits. Since this discussion is general and quite broad, the specific work which is more closely related to our research will be discussed at the end of each chapter.

1.2.1 Decision Diagrams

Decision Diagrams are data structures to represent discrete functions and are derived from decision trees using reduction rules which produce a canonical form. The idea of representing Boolean functions as decision digrams can be traced back to Akers' paper [2], but the widespread use as a data structure for symbolic Boolean manipulation only started with the formulation of a set of algorithms to operate on these data structures by Bryant in 1986 [13]. After that, the basic ideas of BDDs have been extended to allow efficient representation of other classes of functions. As mentioned in [94], we encountered more than 43 different decision diagrams in the past several years. Good surveys of decision diagram related work can be found in [15, 16, 94].

In this section, we summarize a few decision diagrams which have had strong impact, especially in the area of formal verification. Based on the range of the function values, decision diagrams can be divided into two classes: Bit-Level diagrams which have Boolean values and Word-Level diagrams which have integer or floating-point values.

Bit-Level Diagrams
Binary Decision Diagrams (BDDs) [13] represent switching functions $f : B^n \to B$, where n is the number of input variables. BDDs are based on a decomposition of Boolean functions called the "Shannon expansion". A function f can be in terms of a variable x as

$$f = \overline{x} \wedge f_{\overline{x}} \vee x \wedge f_x \tag{1.1}$$

where $\wedge$, $\vee$ and overline represent Boolean product , sum and complement. Term f_x (respectively, $f_{\overline{x}}$) 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. This decomposition is the basis for the BDD representation.

Two alternative function decompositions can be expressed in terms of the XOR (exclusive-or) operation:

$$f = f_{\overline{x}} \oplus x \wedge f_{\delta x} \tag{1.2}$$

$$= f_x \oplus \overline{x} \wedge f_{\delta x} \tag{1.3}$$

where $f_{\delta x}$ denotes the Boolean difference of function f with respect to variables x, i.e. $f_{\delta x} = f_x \oplus f_{\overline{x}}$. Equation 1.2 is commonly referred to as the "positive Davio" or "Reed-Muller" expansion, while Equation 1.3 is referred to as the "negative Davio" expansion. Functional Decision Diagrams (FDDs) [68] use the positive Davio expression as the basis of the graph representation of Boolean functions. For some classes of functions, FDDs are exponentially more compact than BDDs, but the reverse can also hold. To obtain the advantage of each, Drechsler *et al* have proposed a hybrid form called Kronecker FDDs (KFDDs) [42]. In this representation, each variable has an associated decomposition, which can be any one of the three given by Equations 1.1- 1.3.

Minato has developed another variant of BDDs, called "Zero-suppressed" BDDs (ZBDDs) [58], for combinatorial problems that can be solved by representing and manipulating sparse sets of bit vectors of length n [58]. The data for a problem are encoded as bit vectors of length n. Then any subset of the vectors can be represented by a Boolean function over n variables yielding 1 when the vector corresponding to the variable assignment is in the set. Minato has shown that a number of combinatorial problems can be solved efficiently using a ZBDD representation [78]. It can be shown that ZBDDs reduce the size of the representation of a set of n-bit vectors over BDDs by at most a factor of n [90]. In practice, the reduction is large enough to have a significant impact.

Bit-level diagrams are not suitable for the verification of complex arithmetic circuits. First, it is very difficult to write the specification for each output bit as a Boolean function. Second, bit-level diagrams usually explode in size when representing arithmetic circuits. For example, BDD representations for integer multiplication have been shown to be exponential in the number of input bits [14]. Yang *et. al* reported that the number of BDD nodes to represent integer multiplication grows exponential at a factor of about 2.87 per bit of word size [97]. For a 16-bit multiplier, building the BDDs for the output bits requires about 3.8GB memory on a 64-bit machine (i.e. 1.9GB on a 32-bit machine).

Word-Level Diagrams

Building on the success of BDDs, there have been several efforts to extend the concept to represent functions over Boolean variables, with non-Boolean ranges, such as integers and real numbers. For example, it is very useful for verification of arithmetic circuits to represent vectors of Boolean functions as word-level functions such as integer or floating-point functions. A vector of m Boolean functions $(f_0, f_1, ..., f_{m-1})$ can be interpreted as a integer function F whose value at $\vec{x} = (x_0, x_1, ..., x_{n-1})$ is $F(\vec{x}) = \sum_{i=0}^{m-1} f_i(x_0, ..., x_{n-1}) \times 2^i$. Keeping the variables Boolean allows the use of a branching structure similar to BDDs. The challenge becomes finding a compact way to encode the numeric function values.

One straightforward way to represent numeric-valued functions is to use a decision diagrams like a BDD, but to allow arbitrary values on the terminal nodes. This representation is called Multi-Terminal BDDs (MTBDDs) [34] or Algebraic Decision Diagrams (ADDs) [4]. For expressing functions having numeric range, the Boole-Shannon expansion can be generalized as:

$$f \;=\; (1-x)\cdot f_{\overline{x}} \;+\; x\cdot f_x \tag{1.4}$$

where $\cdot$, $+$, and $-$ denote multiplication, addition, and subtraction, respectively. Note that this expansion relies on the assumption that variable x is Boolean, i.e., it will evaluate to either 0 or 1. Both MTBDDs and ADDs are based on such a pointwise decomposition. For some applications, the number of possible values is small enough that the graph size is not too big. In such applications, the simplicity of the representation makes MTBDDs viable candidate. However, these diagrams grow exponentially with the number of Boolean variables for the common integer encodings such as unsigned binary, ones complement and two's complement. For the case of unsigned binary numbers of length n, there are 2^n possible values and hence the MTBDD representation must have 2^n leaf nodes.

For applications where the number of possible function values is too high for MTBDD, Edge-Valued BDDs (EVBDDs) are introduced by incorporating numeric weights on the edges in order to allow greater sharing of subgraphs [72, 73]. The edge weights are combined additively. The common integer encodings can be represented in linear size of EVBDDs. For two integers X and Y represented by EVBDDs, the sum and difference also have linear complexity. However, for the multiplication, the complexity and size of EVBDDs grows exponentially. Multiplicative edge weights are added into EVBDDs to yield another representation called Factored EVBDDs (FEVBDDs). However, these diagrams still cannot represent $X \cdot Y$ in polynomial size.

To overcome this exponential growth, we proposed Binary Moment Diagrams (BMDs) which provides a compact representation for these integer encodings and operations. BMDs use a function decomposition with respect to the input variables in a manner analogous to FDDs. The function decomposition used by BMDs [20, 21], is obtained by rearranging the terms of Equation 1.4:

$$f \;=\; f_{\overline{x}} \;+\; x\cdot f_{\delta x} \tag{1.5}$$

where $f_{\delta x} = f_x - f_{\overline{x}}$ is called the *linear moment* of f with respect to x. This terminology arises from 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 will be termed the *constant moment*, i.e., it denotes the portion of function f that remains constant with respect to x.

An extension of BMDs is to incorporate weights on the edges, yielding a representation called Multiplicative BMDs (*BMDs) [21]. These edge weights combine multiplicatively, rather than additively as with EVDDs. With *BMDs, word-level functions such as $X+Y$, $X-Y$, $X\cdot Y$ and 2^X all have linear-sized representations. The development of *BMDs enables us to verify arithmetic circuits such as multipliers, dividers, etc.

Adapting the idea of KFDDs, Clarke, *et al* have developed a hybrid between MTBDDs and BMDs, which is called Hybrid Decision Diagrams (HDDs) [30]. In their representations, each variable can use one of six different decompositions, including Shannon, positive Davio and negative Davio decompositions. In their experience, the variables for the control signals should use Shannon decomposition to achieve smaller graph sizes.

Adding both additive and multiplicative weights into HDDs yields another representation called Kronecker *BMDs (K*BMDs) [41]. In this representation, the variables can only use one of Shannon, positive Davio and negative Davio decompositions. Both HDDs and K*BMDs are superset of MTBDDs and BMDs. However, we do not find the additive edge weight useful for the verification of arithmetic circuits.

For the word-level diagrams mentioned above, functions mapping Boolean variables into floating-point values can not be represented efficiently without introducing rational weights on the edges or the leaf nodes. The overhead of storing and manipulating the rational edge weights make them less attraction for representing floating-point functions.

We introduced Multiplicative Power HDDs (*PHDDs) [28] to provide a compact representation for integer and floating-point functions by extracting powers of 2 for the edge weights rather than greatest common divisors in *BMDs. The edge weights only record the powers. In other word, the edge weight k represents 2^k and the edge weights are combined multiplicatively in the same way as *BMDs. With this feature, *PHDDs can represent and manipulate integer and floating-point functions efficiently and can be used in the verification of floating-point circuits such as adders.

1.2.2 Verification Techniques

Theorem Proving

In a theorem proving approach, the circuit is described as a hierarchy of components, and there is a behavioral description of each component in the hierarchy. The proof of the correctness of a design is based on the proofs of the correctness of its components, which is obtained by composing and inferring the proofs of the components at lower levels. Theorem provers, HOL [46], Boyer-Moore [8], PVS [80] and ACL2 [67], have been successfully used to verify several hardware systems.

HOL, developed at Cambridge University, is one of the best know theorem provers applied to hardware verification [46]. Joyce [64] and Melham [77] also used HOL to perform verification of circuits. A significant application of the Boyer-Moore theorem prover is the verification of the FM8501 microprocessor by Hunt [57]. PVS provides a specification language integrated with a theorem prover, and support procedures to ease the burden of developing proofs. Srivas and his colleagues have used PVS to verify several hardware designs such as a commercial processor [93].

Theorem provers have been used to verify arithmetic circuits and algorithms. Most of the IEEE floating point standard has been formalized by Carreño and Miner [26] in the HOL and PVS theorem provers. Verkest *et al* verified a nonrestoring division algorithm and hardware implementation using the Boyer-Moore theorem prover. Leeser *et al* verified a radix-2 square root algorithm and hardware implementation [74].

In response to Intel's famous DIV bug in Pentium processor based on SRT algorithm [76, 87, 96], the correctness of the SRT algorithm and implementation has been verified by Clarke *et al* [31] using a theorem prover, Analytica [35], and by Rueß *et al* [89] using PVS. Miner and Leathrum [79] generalized Rueß's verification work to encompasses many division algorithms and to includes a formal path relating the algorithms to the IEEE standard. AMD hired the CLI company to prove the correctness of the AMD 5K86's floating-point division algorithm using ACL2 [11]. They reported that over 1600 definitions and lemmas were involved in this proof. Kapur *et al* [66] used a theorem prover to prove the correctness of a family of integer multipliers based on Wallace trees.

However, the basic weakness of the theorem proving approach is that it requires a large amount of user intervention to create specifications and to perform proofs, which makes them unsuitable for automation. Attempts at automation of proofs have not been particularly successful, and proofs still require substantial interaction and guidance from skilled users.

Model Checking

Model checking is an automatic verification methodology to verify circuits. In this approach, a circuit is described as a state machine with transitions to describe the circuit behavior. The specifications to be checked are described as properties that the machine should or should not satisfy. Based on the data structure for representing state transitions, the model checkers can be categorized into: 1) pure BDD-based model checkers such as SMV [75], VIS [10] and COSPAN [53], and 2) other model checkers such as SPIN [55], Murphi [40] and COSPAN [53]. Note that COSPAN can use either BDD-based or explicit-state enumeration algorithms. COSPAN is the core engine of commercial verification tool $FormalCheck^{TM}$ from Lucent Technology Inc.

Traditionally, model checkers used explicit representations of the state transition graph, which made their use impractical for all but the smallest state machines. To overcome this capacity limitation, BDDs are used to represent the state transition graphs and thus allows model checkers (SMV, VIS and COSPAN) to verify systems with as many as 10^{100} states, much larger than can be verified using an explicit state representation technique. However, these model checkers still have the state explosion problem (i.e., BDD size explosion) while verifying large circuits. A number of approaches [23, 45] have focused on the use of partitioned transition relations to reduce the BDD size during state machine exploration, but the capacity limitation is still a major problem preventing model checkers from verifying industrial circuits without abstraction.

Model checkers, SPIN, Murphi and COSPAN, use other techniques to improve the capacity of the model checking without using BDDs. SPIN uses an optimized depth-first-search graph traversal method to perform the verification task. To avoid a purely exhaustive search, SPIN uses a number of special purpose algorithms such as partial order reduction [54], state compression, and sequential bitstate hashing. Murphi explicitly generates the states and stores them in a hash table. To increase its capacity, Murphi uses many state reduction techniques including symmetry reduction [59] and exploitation of reversible rules [60]. To increase its capacity and performance, COSPAN uses several caching and hashing options, and a state minimization algorithm in its explicit-state enumeration algorithm.

In general, these model checkers can verify systems with up to 500 latches. To handle the real circuit designs in industry, the circuits must be simplified by manual abstraction. For example, the verification of the circuit to handle the cache coherence among 16 processors must be abstracted to a model that contains fewer processors (e.g. 4 processors) and a smaller word size (e.g. 1 or 2 data bits instead of 32 or 64). A number of approaches [32, 84] have been proposed to perform the abstraction automatically.

To verify arithmetic circuits, these model checkers have the following difficulties. First, their specification languages are not powerful enough to express arithmetic properties. For arithmetic circuits, the specifications must be expressed as Boolean functions, which is not suitable for complex circuits. Second, these model checkers cannot represent arithmetic circuits efficiently in their models. For example, SMV will have the BDD explosion problem for representing integer multipliers.

In order to overcome these problems, Clarke *et al* presented word-level SMV based on BDDs for Boolean functions and HDDs for integer functions [30, 33]. The specifications of arithmetic circuits are expressed in word-level and represented by HDDs. Chen *et al* [29] have applied word-level SMV to verify arithmetic circuits in one of Intel's processors. In this work, floating-point circuits were partitioned into several sub-circuits whose specifications could be expressed in terms of integer operations, because HDDs can not represent floating-point functions efficiently. The main drawback of this partitioning approach is that the specifications are implementation dependent and cannot be reused for for different implementations. For example, two different implementations of the floating-point adder can yield different partitions, and thus the specifications for the sub-circuits for one design are different from those in another. Another drawback is that this approach requires user intervention to partition the circuits and reason about the correctness of the overall specifications from the verified sub-specifications.

Symbolic Simulation

Symbolic simulation is a well know technique for simulation and verification of digital circuits and is an extension of conventional digital simulation, where a simulator evaluates circuit behavior using symbolic Boolean variables to encode a range of circuit operating conditions. The initial internal state variables as well as the input values can be Boolean expressions, which are usually expressed in parametric form [12]. This makes each run of a symbolic simulator equivalent multiple runs of a conventional simulator. Jain *et al* [63] presented an efficient method to generate the parametric form.

After the introduction of BDDs by Bryant, symbolic simulation became more practical. Both COSMOS [18] developed at Carnegie Mellon University and Voss [91] developed at the University of British Columbia use BDDs as the representations for Boolean functions. These two simulators have been the framework for verification of different classes of circuits [6, 19, 61, 81, 82]. Beatty *et al* [6] verified a microprocessor using COSMOS. Voss has been used to verify memory arrays in the PowerPC processor [82]. Bose and Fisher [7] have used symbolic simulation to a verify pipelined hardware system.

Compared with model checking, the symbolic simulation technique can handle much larger circuits, because this approach can only cover some of the input spaces in each simulation run. However, symbolic simulation can not used to completely verify arithmetic circuits such as integer multipliers, dividers, etc, because the input spaces of these circuits are very large and the BDDs blow up exponentially for these circuits. An exhaustive simulation to cover the entire input space is almost impossible for large integer multipliers, dividers and floating-point circuits. Another problem for symbolic simulation is that the specifications must be expressed as Boolean functions, which are very complicated for arithmetic circuits.

Equivalence Checking

In recent years, many CAD vendors offered equivalence checking tools for design verification. For example, the partial list of equivalence checkers are Formality (from Synopsys), Design Verifyer (from Chrysalis), VFormal (from Compass), Verity (from IBM). These tools perform logic equivalence checking of two circuits based structural analysis and BDD techniques.

Some equivalence checking techniques have been described in [37, 38, 70]. The common assumption used in the equivalence checking is that two circuits have identical state encodings (latches) [70]. With this assumption, only the equivalence of the combinational portions of two circuits must be checked. Coudert *et al* [37, 38] and Cabodi *et al* [24] use a symbolic breadth first exploration of the product machine state graph to do equivalence checking for two circuits without identical state encodings. These tools can handle the large designs with similar structures. However, these tools can not handle the equivalent designs with little structure similarity. For example, an array multiplier and a booth-encoding multiplier can not be proved to be equivalent using these tools. Another drawback of equivalence checkers is that they all need “golden” circuits as the reference to be compared with. The correctness of “golden” circuits is still questionable.

Hybrid approaches: Theorem prover with model checking or simulation
Another approach to verifying circuits is combining a theorem prover with a model checking or symbolic simulation tool [85, 65]. In this approach, theorem provers handle the high-level proofs, while the low-level properties are handled by the model checking or symbolic simulation tool.

Camilleri [25] used simulation as an aid to perform verification of circuits using the HOL theorem prover. Kurshan *et al* [71] verified local properties of the low-level circuits of a multiplier using COSPAN and verified the correctness of the whole multiplier by a theorem prover to compose of the verified local properties. Aagaard *et al* [1] used Voss and a theorem prover to verify a IEEE double-precision floating-point multiplier. Compared with the theorem proving approach, this approach is much more automatic, but still requires user guidance.

Other Approaches
Burch [22] has implemented a BDD-based technique for verifying certain classes of integer multipliers. His method effectively creates multiple copies of the multiplier variables, leading to BDDs that grow cubically with the word size. The limiting factor in dealing with larger word sizes would be the cubic growth in memory requirement. Furthermore, this approach cannot handle multipliers that use multiplier recoding techniques, although Burch describes extensions to handle some forms of recoding.

Jain *et al* [62] have used Indexed Binary Decision Diagrams (IBDDs) to verify several multiplier circuits. This form of BDD allows multiple repetitions of a variable along a path from root to leaf. They were able to verify C6288 (a 16-bit multiplier) in 22 minutes of CPU time on a SUN-4/280, generating a total of 149,498 graph vertices. They were also able to verify a multiplier using Booth encoding, but this required almost 4 hours of CPU time and generated over 1 million vertices in the graphs.

Based on the Chinese remainder theorem, Kimura [69] introduced residue BDDs, which have bounded size, to verify a 16×16 integer multiplier. In [86], Ravi *et al* discuss how to choose a good modulus and also show how to build residue BDDs for complex circuits involving function blocks. They have shown that this approach can detect the bugs efficiently. However, this approach cannot be extended to verify larger integer multipliers such as 64-bit multipliers.

Bryant [17] has used BDD to check the properties and invariants that one iteration of the SRT circuits must preserve for the circuit to correctly divide. To do the verification, he needed to construct a gate-level checker-circuit (much larger than the verified circuit) to describe the desired behavior of the verified circuit, which is not the ideal level of specification.

1.3 Scope of the thesis

Approaches based on decision diagrams have been used to verify arithmetic circuits. However, direct application of decision diagrams in this domain is not without challenges. We classify these challenges into two categories – *representation* and *methodology*. The main representation challenges are to overcome the explosion problem of the existing decision diagrams and to provide a compact representation for integer and floating-point functions such that the specification can be easily be expressed in word level. The methodology related challenges includes problems like having a framework to specify and verify arithmetic properties, automating the verification process, and making the specification reusable. We have built on earlier work on decision diagrams and methodologies to overcome these challenges. The principal contributions of this thesis are the following:

- *BMD-based hierarchical verification for integer circuits
  We have developed *BMDs to represent integer functions efficiently and thus have enabled the verification of integer circuits. Based on *BMDs and a hierarchical verification methodology, we have built a system to verify several integer circuits such as integer multipliers, dividers and square roots.

- Representation for floating-point functions
  We have developed *PHDDs to represent floating-point functions efficiently. We also have analyzed the complexities of representing floating-point addition and floating-point multiplication using *PHDDs.

- Methodologies for verification of floating-point circuits
  We have developed several methodologies for the verification of floating-point circuits, especially floating-point adders. These methodologies have been integrated into word-level SMV.

- Verification of floating-point adders and conversion circuits
  A FP adder designed by Dr. Huff at the University of Michigan was verified by our approach with reusable specifications. Several bugs were found in the design. A counterexample for each bug can be generated within 5 minutes. The reusability of our specifications is demonstrated by the verification of a variant of Huff's FP adder.

1.4 Thesis overview

Chapter 2 presents Multiplicative Binary Moment Diagrams (*BMDs) which enable the verification of arithmetic circuits. The data structures and algorithms for *BMDs provide a compact representation for integers functions.

Chapter 3 discusses the verification of integer circuits such as multipliers, dividers and square roots. A hierarchical verification method based on *BMDs is presented for verification of integer circuits along with other methods. Based on these methodologies, the Arithmetic Circuit Verifier (ACV) was built to verify integer multipliers, dividers and square roots.

Chapter 4 presents Multiplicative Power Hybrid Decision Diagrams (*PHDDs) for the verification of floating-point circuits. First, we discuss the limitations of *BMDs and HDDs and thus the need for a new diagram to represent floating-point functions. A performance comparison between *BMDs and *PHDDs is discussed.

Chapter 5 discusses methodologies for verification of floating-point circuits, especially floating-point adders. First, we discuss the drawbacks of ACV which lead us to verify flattened designs of floating-point circuits. Then, we present several improvements of word-level SMV to enable the verification of floating-point circuits.

Chapter 6 presents the verification work of a floating-point adder obtained from the University of Michigan. We discuss the circuit design of this FP adder and the reusable specifications for FP adders. Several bugs were found in this design by our system. A variant of this FP adder is created and verified to illustrate that our approach is implementation independent.

Chapter 7 rounds off the thesis with an evaluation of the work and possible future research directions.

The appendix shows the complexity analysis of representing floating-point addition and multiplication using *PHDDs.

Chapter 2

Representations for Integer Functions

To verify integer arithmetic circuits, we must have concise representations for word-level functions that map Boolean vectors to the integer values. Such Boolean vectors correspond to the input operands and the final result. In this chapter, we discuss Multiplicative Binary Moment Diagrams (*BMDs) which provide a compact representation for integer functions. *BMDs have efficient representations for common integer encodings as well as operations such as addition and multiplication, and enable us to easily verify integer circuits such as multipliers and dividers. The verification of these circuits will be described in Chapter 3.

Section 2.1 presents the data structure for *BMDs. The *BMD representations of integer functions and operations are shown in Section 2.2. Then, the algorithms for *BMDs are presented in Section 2.3. Section 2.4 describes the related work.

2.1 The *BMD Data Structure

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

2.1.1 Function Decompositions

To illustrate ways of decomposing a function, consider the function F over a set of Boolean variables y and z, yielding the integer values shown in the table of Figure 2.1. BDDs are

Function

y	z	F
0	0	8
0	1	-12
1	0	10
1	1	-6

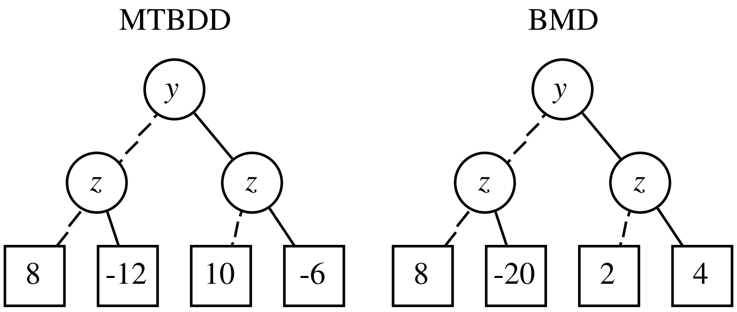


Figure 2.1: **Example Function Decompositions.** MTBDDs are based on a pointwise decomposition (left), while BMDs are based on a linear decomposition (right).

based on a pointwise decomposition, characterizing a function by its value for every possible set of argument values. By extending BDDs to allow numeric leaf values, the pointwise decomposition leads to a “Multi-Terminal” BDD (MTBDD) representation of a function [34] (also called “ADD” [4]), as shown on the left side of Figure 2.1. In this drawing, the dashed line from a vertex denotes the case where the vertex variable is 0, and the solid line denotes the case where the variable is 1. Observe that the leaf values correspond directly to the entries in the function table.

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

$$\begin{aligned} F(x,y) &= \left[\begin{array}{cccc} 8 & (1-y) & (1-z) & + \\ -12 & (1-y) & z & + \\ 10 & y & (1-z) & + \\ -6 & y & z & \end{array} \right] \\ &= 8 - 20z + 2y + 4yz \end{aligned}$$

This expansion leads to the BMD representation of a function, as shown on the right side of Figure 2.1. In our drawings of graphs based on a moment decomposition, the dashed line from a vertex indicates the case where the function is independent of the vertex variable, while the solid line indicates the case where the function varies linearly. Observe that the leaf values correspond to the coefficients in the linear expansion.

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

Boolean function f can be decomposed in terms of variable x in terms of an expansion (variously credited to Shannon and to Boole): $f \;=\; \overline{x} \wedge f_{\overline{x}} \;\vee\; x \wedge f_x$. In this equation we use $\wedge$ and $\vee$ to represent Boolean sum and product, and overline to represent Boolean complement. Term f_x (respectively, $f_{\overline{x}}$) denotes the positive (resp., negative) *cofactor* of f with respect to variable x, i.e., the function resulting when constant 1, (resp., 0) is substituted for x. This decomposition is the basis for the BDD representation.

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

$$f \;=\; (1-x)\cdot f_{\overline{x}} \;+\; x \cdot f_x \tag{2.1}$$

where $\cdot$, $+$, and $-$ denote multiplication, addition, and subtraction, respectively. Note that this expansion relies on the assumption that variable x is Boolean, i.e., it will evaluate to either 0 or 1. Both MTBDDs and EVBDDs [73] are based on such a pointwise decomposition. As with BDDs, each vertex describes a function in terms of its decomposition with respect to the variable labeling the vertex. The two outgoing arcs denote the negative and positive cofactors with respect to this variable.

The moment decomposition of a function is obtained by rearranging the terms of Equation 2.1:

$$\begin{aligned} f &= f_{\overline{x}} \;+\; x \cdot (f_x - f_{\overline{x}}) \\ &= f_{\overline{x}} \;+\; x \cdot f_{\delta x} \end{aligned} \tag{2.2}$$

where $f_{\delta x} \;=\; f_x \;-\; f_{\overline{x}}$ is called the *linear moment* of f with respect to x. This terminology arises by viewing f as being a linear function with respect to its variables, and thus $f_{\delta x}$ is the partial derivative of f with respect to x. Since we are interested in the value of the function for only two values of x, we can always extend it to a linear form. The negative cofactor will be termed the *constant moment*, i.e., it denotes the portion of function f that remains constant with respect to x. Each vertex of a BMD describes a function in terms of its moment decomposition with respect to the variable labeling the vertex. The two outgoing arcs denote the constant and linear moments of the function with respect to the variable.

The moment decomposition of Equation 2.2 is analogous to the Reed-Muller expansion for Boolean functions: $f \;=\; f_{\overline{x}} \;\oplus\; x \wedge (f_x \oplus f_{\overline{x}})$. The expression $f_x \oplus f_{\overline{x}}$ is commonly known as the *Boolean difference* of f with respect to x, and in many ways is analogous to our linear moment. Other researchers [68] have explored the use of graphs for Boolean functions based on this expansion, calling them Functional Decision Diagrams (FDDs). By our terminology, we would refer to such a graph as a "moment" diagram rather than a "decision" diagram.

2.1.2 Edge Weights

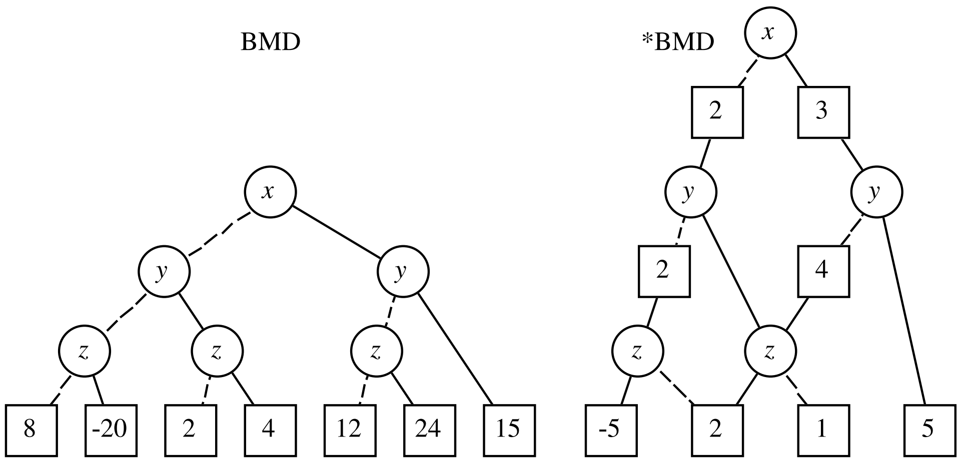


Figure 2.2: **Example of BMD vs. *BMD.** Both represent the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. *BMDs have weights on the edges that combine multiplicatively.

The BMD data structure encodes numeric values only in the terminal vertices. As a second refinement, we adopt the concept of edge weights, similar to those used in EVBDDs. In our case, however, edge weights combine multiplicatively, and hence we call these data structures *BMDs. As an illustration, Figure 2.2 shows representations of the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$. In the BMD representation, leaf values correspond to the coefficients in the linear expansion. As the figure shows, the BMD data structure misses some opportunities for sharing of common subexpressions. For example, the terms $2y + 4yz$ and $12x + 24xz$ can be factored as $2y(1 + 2z)$ and $12x(1 + 2z)$, respectively. The representation could therefore save space by sharing the subexpression $1 + 2z$. For more complex functions, one might expect more opportunities for such sharing.

The *BMD shown in Figure 2.2 indicates how *BMDs are able to exploit the sharing of common subexpressions. In our drawings of *BMDs, we indicate the weight of an edge in a square box. Unlabeled edges have weight 1. In evaluating the function for a set of arguments, the weights are multiplied together when traversing downward. Several rules for manipulating edge weights can be formulated that guarantee the resulting graph form is canonical. For representing functions with integer ranges, we require that the edge weights for the two branches leaving a vertex be relatively prime. We also require that weight 0 only appear as a terminal value, and that when a node has one such branch, the other branch has weight 1. This property is maintained by the way in which the *BMDs are generated, in a manner analogous to BDD generation methods. For the remainder of the presentation we will consider only *BMDs, The effort required to implement weighted edges is justified by the savings in graph sizes.

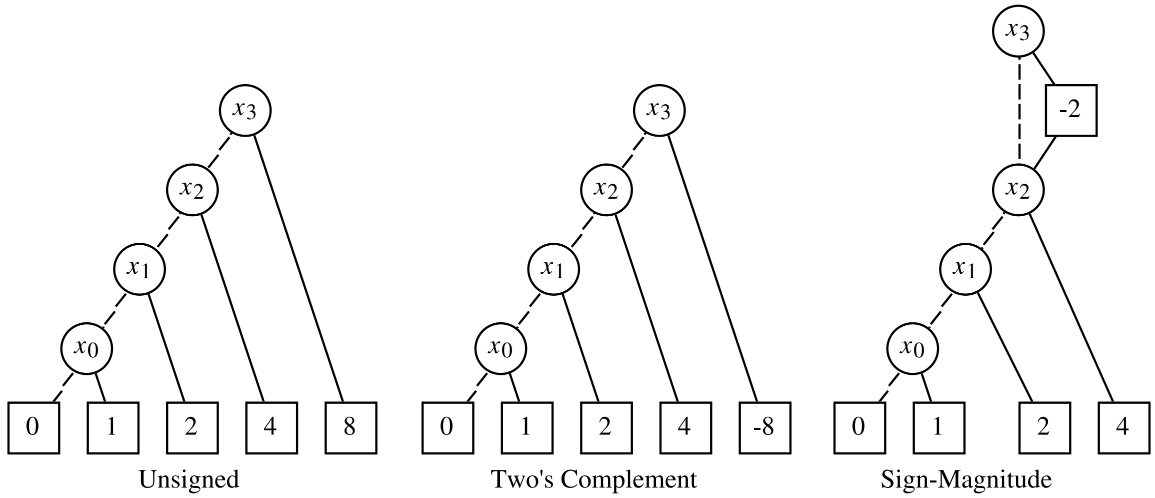


Figure 2.3: **Representations of Integers.** All commonly used encodings can be represented with linear complexity.

2.2 Representation of Integer Functions

*BMDs provide a concise representation of functions defined over "words" of data, i.e., vectors of bits having a numeric interpretation. Let $\vec{x}$ represent a vector of Boolean variables: $x_{n-1}, \ldots, x_1, x_0$. These variables can be considered to represent an integer X according to some encoding, e.g., unsigned binary, two's complement, BCD, etc. Figure 2.3 illustrates the *BMD representations of several common encodings for integers. An unsigned number is encoded as a sum of weighted bits. The *BMD representation has a simple linear structure with the different weights forming the leaf values. For representing signed numbers, we assume x_{n-1} is the sign bit. The two's complement encoding has a *BMD representation similar to that for unsigned integers, but with bit x_{n-1} having weight -2^{n-1}. For a one's complement encoding (not shown), the sign bit has weight $-2^{n-1} + 1$. Sign-magnitude integers also have *BMD representations of linear complexity, but with the constant moment with respect to x_{n-1} scaling the remaining unsigned number by 1, and the linear moment scaling the number by -2. In evaluating the function for $x_{n-1} = 1$, we would add these two moments effectively scaling the number by -1.

2.2.1 Integer Operations

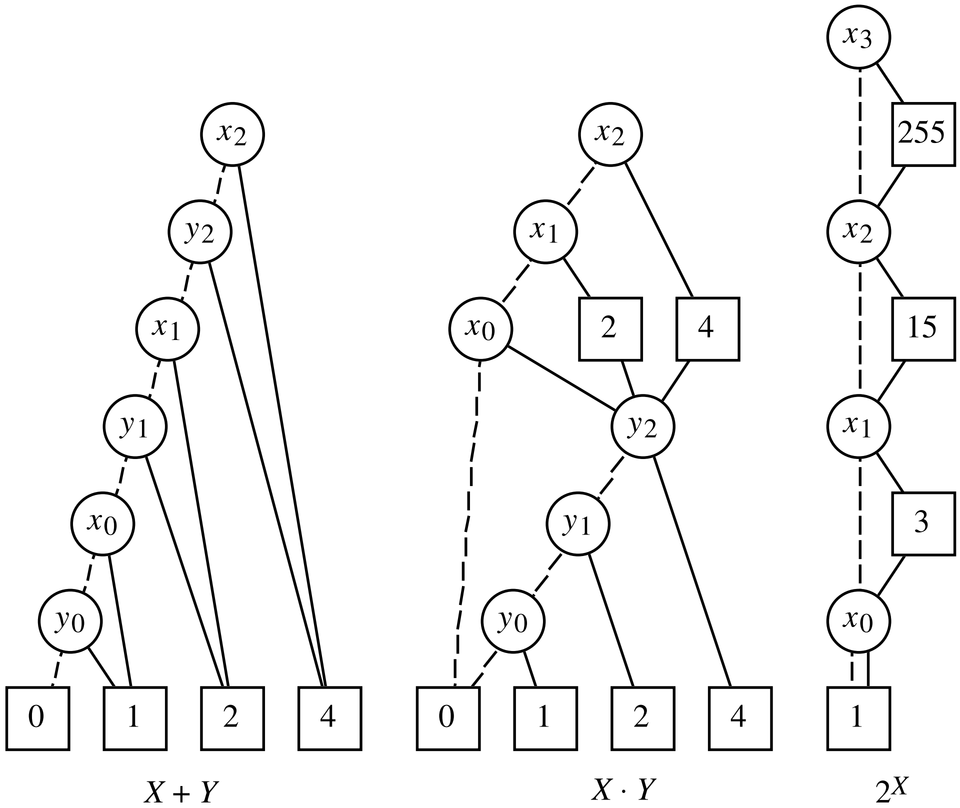


Figure 2.4: **Representations of Word-Level Sum, Product, and Exponentiation.** The graphs grow linearly with word size.

Figure 2.4 illustrates the *BMD representations of several common arithmetic operations on word-level data. Observe that the sizes of the graphs grow only linearly with the word size n. Word-level addition can be viewed as summing a set of weighted bits, where bits x_i and y_i both have weight 2^i. Word-level multiplication can be viewed as summing a set of partial products of the form $x_i 2^i Y$. In representing the function c^X (in this case $c = 2$), the *BMD expresses the function as a product of factors of the form $c^{2^i x_i} = (c^{2^i})^{x_i}$. In the graph, a vertex labeled by variable x_i has outgoing edges with weights 1 and $c^{2^i} - 1$ both leading to a common vertex denoting the product of the remaining factors. Interestingly, the sizes of these representations are hardly sensitive to the variable ordering—they remain of linear complexity in all cases. We have found that variable ordering is much less of a concern when representing word-level functions with *BMDs than it is when representing Boolean functions with BDDs.

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

2.2.2 Representation of Boolean Functions

In verifying arithmetic circuits, we abstract from the bit-level representation of a circuit, where each signal is binary-valued, to a word level, where bundles of signals encode words of data. In performing this transformation we must represent both Boolean and word-level functions. Hence we require our data structure to be suitable for representing Boolean functions as well.

Boolean functions are just a special case of numeric functions having a restricted range. Therefore such functions can be represented as *BMDs. Figure 2.5 illustrates the *BMD representations of several common Boolean functions over multiple variables, namely their Boolean product and sum, as well as their exclusive-or sum. As this figure shows, the *BMD of Boolean functions may have values other than 0 or 1 for edge weights and leaf values. Under all variable assignments, however, the function will evaluate to 0 or to 1. As can be seen in the figure, these functions all have representations that grow linearly with the number of variables, as is the case for their BDD representations.

Figure 2.6 shows the the bit-level representation of a 3-bit adder. It represents the 4 adder outputs as a single *BMD having multiple roots, much as is done with a shared BDD representation. The complexity of this representation grows linearly with the word size. Observe the relation between the word-level representation (Figure 2.4, left) and the bit-level representation of addition. Both are functions over variables representing the adder inputs, but the former is a

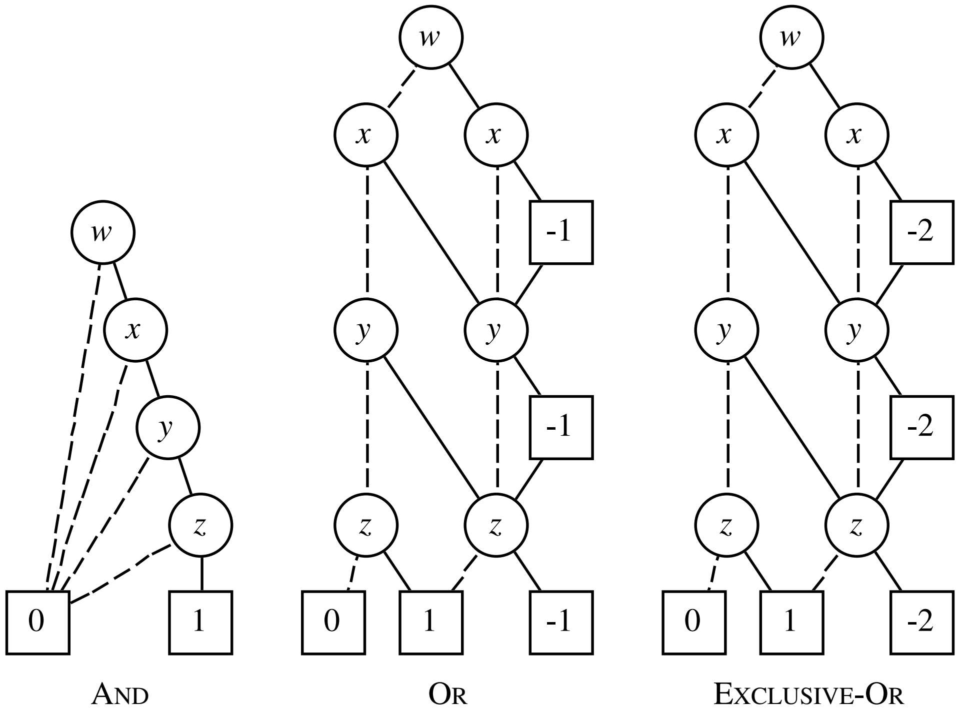


Figure 2.5: **Representations of Boolean Functions.** Representations as *BMDs are comparable in size to BDDs.

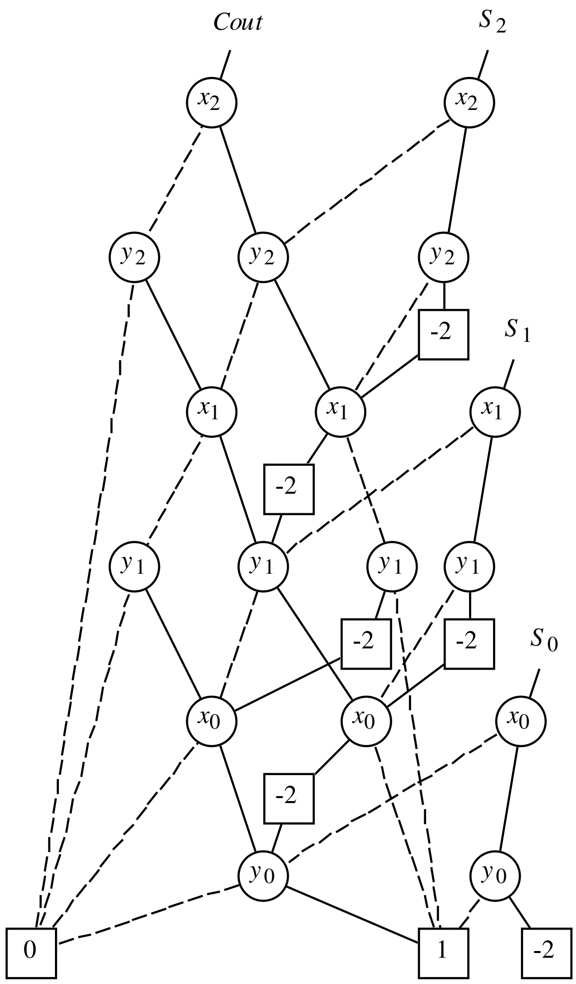


Figure 2.6: **Bit-Level Representation of Addition Functions.** The graph represents all four outputs of a 3-bit adder.

single function yielding an integer value, while the latter is a set of Boolean functions: one for each circuit output. The relation between these two representations will be discussed more fully in our development of a verification methodology.

In all of the examples shown, the *BMD representation of a Boolean function is of comparable size to its BDD representation. In general this will not always be the case. Enders [43] has characterized a number of different function representations and shown that *BMDs can be exponentially more complex than BDDs, and vice-versa. The two representations are based on different expansions of the function, and hence their complexity for a given function can differ dramatically. In our experience, *BMDs generally behave almost as well as BDDs when representing Boolean functions.

2.3 Algorithms for *BMDs

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

2.3.1 Representation of *BMDs

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

Var(v): The vertex variable.

Hi(v): The pair designating the linear moment.

Lo(v): The pair designating the constant moment.

Uid(v): Unique identifier for vertex.

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

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

vertex *UniqueVertex*(**variable** *x*, **pair** $\langle w_l, v_l\rangle$, **pair** $\langle w_h, v_h\rangle$)
{ Maintain set of graph vertices such that no duplicates created }
  key ← $[x, w_l,$ Uid$(v_{\rm l}), w_{\rm h},$ Uid$(v_{\rm h})]$
  found, *v* ← *LookUp*(*UTable*, *key*)
  if *found* then return *v*
  v ← *New*(vertex)
  Var(*v*) ← *x*; Uid(*v*) ← *Unid*();
  Lo(*v*) ← $\langle w_l, v_l\rangle$; Hi(*v*) ← $\langle w_h, v_h\rangle$
  Insert(*UTable*, *key*, *v*)
  return *v*

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

Figure 2.7: **Algorithms for Maintaining *BMD.** These algorithms preserve a strong canonical form.

2.3.2 Maintaining Canonical Form

The functions to be represented are maintained as a single graph in *strong canonical form*. That is, pairs $\langle w_1, v_1\rangle$ and $\langle w_2, v_2\rangle$ denote the same function if and only if $w_1 = w_2$ and $v_1 = v_2$. We assume that the set of variables is totally ordered, and that all of the vertices constructed

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

Figure 2.8: **Algorithm for Multiplying Function by Weight.** This algorithm ensures that edge to a nonterminal vertex has weight 0.

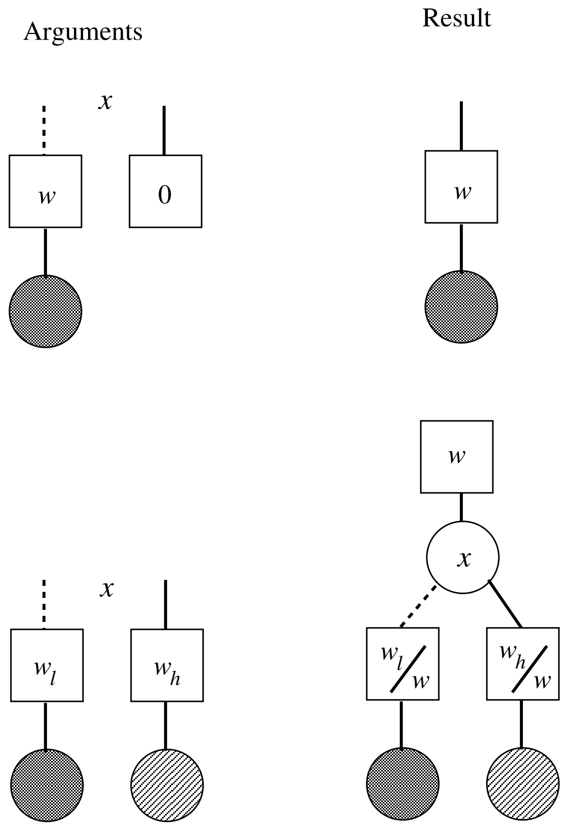


Figure 2.9: **Normalizing Transformations Made by** ***MakeBranch*****.** These transformations enforce the rules on branch weights.

obey this ordering. That is, for any vertex v, its variable $\mathrm{Var(v)}$ must be less than any variable appearing in the subgraphs $\mathrm{Lo(v)}$ and $\mathrm{Hi(v)}$.

Maintaining a canonical form requires obeying a set of conventions for vertex creation and for weight manipulation. These conventions are expressed by the code shown in Figures 2.7 and 2.8. The *MakeBranch* algorithm provides the primary means of creating and reusing vertices in the graph. It is given as arguments a variable and two moments, each represented as weighted pairs. It returns a pair representing the function given by Equation 2.2. It assumes that the argument variable is less than any variable in the argument subgraphs. The steps performed by *MakeBranch* are illustrated in Figure 2.9. In this figure two moments are drawn as weighted pointers.

When the linear moment is the constant 0, we can simply return the constant moment as the result, since this function is independent of variable x. Observe that this rule differs from the reduction rule for a graph based on a pointwise decomposition such as BDDs. In such cases a vertex can be eliminated when both of its children are identical. This reflects the difference between the two different function decompositions. Our rule for *BMDs is similar to that for FDDs [42, 68].

For other values of the linear moment, the routine first factors out some weight w by calling function *NormWeight*, adjusting the weights of the two arguments accordingly. We want to extract any common factor while ensuring that all weights are integers. Hence we take the greatest common divisor (gcd) of the argument weights. In addition, we adopt the convention that the sign of the extracted weight matches that of the constant moment. This assumes that gcd always returns a nonnegative value.

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

Figure 2.8 shows the code for a function *ApplyWeight* to multiply a function, given as a weighted pair, by a constant value, given as a weight w'. This procedure simply adjusts the pair weight, detecting the special case where the multiplicative constant is 0.

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

Termination conditions

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

Table 2.1: **Termination Cases for Apply Algorithms.** Each line indicates an operation, a set of terminations, and the returned result.

Rearrangements

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

Table 2.2: **Rearrangements for Apply Algorithms.** These rearrangements increase the likelihood of reusing a previously-computed result.

2.3.3 The Apply Operations

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

Figure 2.10 shows the fundamental algorithm for adding two functions. The function *PlusApply* takes two weighted pairs indicating the argument functions and returns a weighted pair indicating the result function. This algorithm can also be used for subtraction by first multiplying the second argument by weight -1. This code closely follows the Apply algorithm for BDDs [9].

pair *PlusApply*(**pair** $\langle w_1, v_1\rangle$, **pair** $\langle w_2, v_2\rangle$): **pair**
{ Compute sum of two functions }
    done, $\langle w, v\rangle \leftarrow$ *TermCheck*($+$, $\langle w_1, v_1\rangle$, $\langle w_2, v_2\rangle$)
    if *done* then return $\langle w, v\rangle$
    w', $\langle w_1, v_1\rangle$, $\langle w_2, v_2\rangle \leftarrow$ *Rearrange*($+$, $\langle w_1, v_1\rangle$, $\langle w_2, v_2\rangle$)
    key $\leftarrow [+, w_1, \mathrm{Uid}(v_1), w_2, \mathrm{Uid}(v_2)]$
    found, $\langle w, v\rangle \leftarrow$ *LookUp*(*OpTable*, *key*)
    if *found* then return *ApplyWeight*(w', $\langle w, v\rangle$)
    x $\leftarrow$ *Min*(Var(v_1), Var(v_2))
    { Begin recursive section }
    $\langle w_{1l}, v_{1l}\rangle \leftarrow$ *SimpleMoment*($\langle w_1, v_1\rangle$, *x*, 0)
    $\langle w_{2l}, v_{2l}\rangle \leftarrow$ *SimpleMoment*($\langle w_2, v_2\rangle$, *x*, 0)
    $\langle w_{1h}, v_{1h}\rangle \leftarrow$ *SimpleMoment*($\langle w_1, v_1\rangle$, *x*, 1)
    $\langle w_{2h}, v_{2h}\rangle \leftarrow$ *SimpleMoment*($\langle w_2, v_2\rangle$, *x*, 1)
    $\langle w_l, v_l\rangle \leftarrow$ *PlusApply*($\langle w_{1l}, v_{1l}\rangle$, $\langle w_{2l}, v_{2l}\rangle$)
    $\langle w_h, v_h\rangle \leftarrow$ *PlusApply*($\langle w_{1h}, v_{1h}\rangle$, $\langle w_{2h}, v_{2h}\rangle$))
    { End recursive section }
    $\langle w, v\rangle \leftarrow$ *MakeBranch*(*x*, $\langle w_l, v_l\rangle$, $\langle w_h, v_h\rangle$)
    Insert(*OpTable*, *key*, $\langle w, v\rangle$)
    return *ApplyWeight*(w', $\langle w, v\rangle$)

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

    if Var(v) $\neq x$
        if $b = 0$
            then return $\langle w, v\rangle$
            else return $\langle 0, \Lambda\rangle$
    if $b = 0$
            then return *ApplyWeight*(*w*, Lo(*v*))
            else return *ApplyWeight*(*w*, Hi(*v*))

Figure 2.10: **Apply Algorithm for Adding Two Functions.** The algorithm is similar to the counterpart for BDDs.

It utilizes a combination of recursive descent and "memoizing," where all computed results are stored in a table and reused whenever possible. The recursion is based on the property that taking moments of functions commutes with addition. That is, for functions f and g and for variable x:

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

This routine, like the other Apply algorithms, first checks a set of termination conditions to determine whether it can return a result immediately. This test is indicated as a call to function *TermCheck* having as arguments the operation and the arguments of the operation. This function returns two values: a Boolean value *done* indicating whether immediate termination is possible, and a weighted pair indicating the result to return in the event of termination. Some sample termination conditions are shown in Table 2.1. For the case of addition, the algorithm can terminate if either argument represents the constant 0, or if the two arguments are multiples of each other, indicated by weighted pairs having the same vertex element.

Failing the termination test, the routine attempts to reuse a previously computed result. To maximize possible reuse it first rearranges the arguments and extracts a common weight w'. This process is indicated as a call to the function *Rearrange* having the same arguments as *TermCheck*. This function returns three values: the extracted weight and the modified arguments to the operation. Some sample rearrangements are shown in Table 2.2. For the case of addition rearranging involves normalizing the weights according to the same conditions used in *MakeBranch* and ordering the arguments so that the first has greater weight. For example, suppose at some point we compute $6y - 9z$. We will extract weight -3 (assuming integer weights) and rearrange the arguments as $3z$ and $-2y$. If we later attempt to compute $15z - 10y$, we will be able to reuse this previous result with extracted weight 5.

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

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

The Apply algorithms for multiplication has a similar overall structure to that for addition, but differing in the recursive evaluation. Comments in the code of Figure 2.10 delimit the "recursive section" of the routine. In this section recursive calls are made to create a pair of weighted pointers $\langle w_l, v_l \rangle$ and $\langle w_h, v_h \rangle$ from which the returned result is constructed. For the multiplication algorithm we show only its recursive section.

The multiplication of functions f and g, denoted $f \cdot g$ can be defined recursively as follows. If these functions evaluate to constants a and b, respectively, then their product is simply $f \cdot g = a \cdot b$. Otherwise assume the functions are given by their moment expansions (Equation 2.2) with respect to some variable x. The product of the functions can then be defined as:

$$f \cdot g \;\; = f_{\overline{x}} \cdot g_{\overline{x}} + x(f_{\overline{x}} \cdot g_{\delta x} + f_{\delta x} \cdot g_{\overline{x}}) + x^2 f_{\delta x} \cdot g_{\delta x} \tag{2.3}$$

Under the *Boolean domain restriction*, i.e., considering only variable assignments ϕ such that $\phi(x) \in \{0, 1\}$, we are guaranteed that $x = x^2$. Equation 2.3 can be rewritten as following:

$$f \cdot g \;\; = f_{\overline{x}} \cdot g_{\overline{x}} + x(f_{\overline{x}} \cdot g_{\delta x} + f_{\delta x} \cdot g_{\overline{x}} + f_{\delta x} \cdot g_{\delta x}) \tag{2.4}$$

Figure 2.11 shows the recursive section for multiplying a pair of functions, using the formulation of linear product given by Equation 2.4. Each call to *MultApply* requires four recursive calls, plus two calls to *PlusApply*. With the rearrangements shown in Table 2.2, we can always extract the weights from the arguments. Hence if the arguments have *BMD representations of m_f and m_g vertices, respectively, no more than $m_f m_g$ calls will be made to *MultApply*. Unfortunately, this bound on the calls does not suffice to show a polynomial bound on the complexity of the algorithm. The calls to *PlusApply* may blow up exponentially.

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

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

Figure 2.11: **Recursive Section for Apply Operation for Multiplying Functions.** This operation exploits the ring properties of linear product.

2.4 Related Work

Enders [43] has shown that the representation size for Boolean functions may differ exponentially for BMD, EVBDD and FDD representations. He also proved that the multiplication of BMDs and *BMDs as well as the addition of *BMDs may have exponential operations in the worst case. Arditi [3] used *BMDs for verification of arithmetic assembly instructions to delay the use of theorem provers. Rotter *et al* [88] used *BMDs to represent polynomial functions. Their result shows that *BMDs have better performance than ZBDDs in terms of the number of nodes and CPU time.

Clarke, *et al.* [30] 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 2.2 is generalized to the following three equations according the variable's decomposition type:

$$f \quad = \quad \begin{cases} (1-x)\cdot f_{\overline{x}} + x\cdot f_x & (Shannon) \\ f_{\overline{x}} + x\cdot f_{\delta x} & (Positive\ Davio) \\ f_x + (1-x)\cdot f_{\delta\overline{x}} & (Negative\ Davio) \end{cases} \tag{2.5}$$

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. In their experience, the variables for the control signals should use Shannon decomposition to achieve better performance.

Multiplicative edge weights are added into EVBDDs to yield another representation called Factored EVBDDs (FEVBDDs). However, these diagrams still cannot represent $X \cdot Y$ in polynomial size.

Adding both additive and multiplicative weights into HDDs yields another representation called Kronecker *BMDs (K*BMDs). In their representation, the variables can only use one of Shannon, positive Davio and negative Davio decompositions. Both HDDs and K*BMDs are the superset of MTBDDs and BMDs. However, we do not find any usefulness of the additive edge weight for the verification of arithmetic circuits.

Table 2.3 summarizes the complexity of different word-level decision diagrams to represent the integer functions and operations such as X, $X+Y$, $X*Y$, X^2 and 2^X. Note that *BMDs and K*BMDs have more compact representations than others.

Form	X	$X+Y$	$X*Y$	X^2	2^X
MTBDD	exponential	exponential	exponential	exponential	exponential
EVBDD	linear	linear	exponential	exponential	exponential
FEVBDD	linear	linear	exponential	exponential	exponential
BMD	linear	linear	quadratic	quadratic	exponential
HDD	linear	linear	quadratic	quadratic	exponential
*BMD	linear	linear	linear	quadratic	linear
K*BMD	linear	linear	linear	quadratic	linear

Table 2.3: **Word-Level Operation Complexity.** Expressed in how the graph sizes grow relative to the word size.

Chapter 3

Verification of Integer Circuits

*BMDs can serve as the basis for a hierarchical methodology for verifying integer circuits such as multipliers and dividers. At the low level, we have a set of component modules such as add steppers, Booth steppers, and carry save adders described at both the bit level (in terms of logic gates) and at the word level (as algebraic expressions). Using a methodology proposed by Lai and Vrudhula [73], we verify that the bit-level implementation of each block implements its word-level specification. At the higher level (or levels), a system is described as an interconnection of components having word-level representations, and the specification is also given at the word-level. We then verify that the composition of the block functions corresponds to the system specification. Using this technique we can verify systems such as multipliers that cannot be represented efficiently at the bit level. We also can handle a more abstract level of specification than can methodologies that work entirely at the bit level. Based on *BMDs and this hierarchical verification methodology, we have built an Arithmetic Circuit Verifier (ACV) to verify integer multipliers, dividers, and square roots successfully.

The outline of this chapter is organized as following. Section 3.1 describes our hierarchical verification methodology using *BMDs as the underlying representation. Section 3.2 describes the verification system based on a hierarchical verification methodology. Additional techniques are introduced in Section 3.3 to handle some special circuits such as multipliers using carry-save adders.

3.1 Hierarchical Verification

Figure 3.1 illustrates schematically an approach to circuit verification originally formulated by Lai and Vrudhula [73]. The overall goal is to prove a correspondence between a logic circuit,

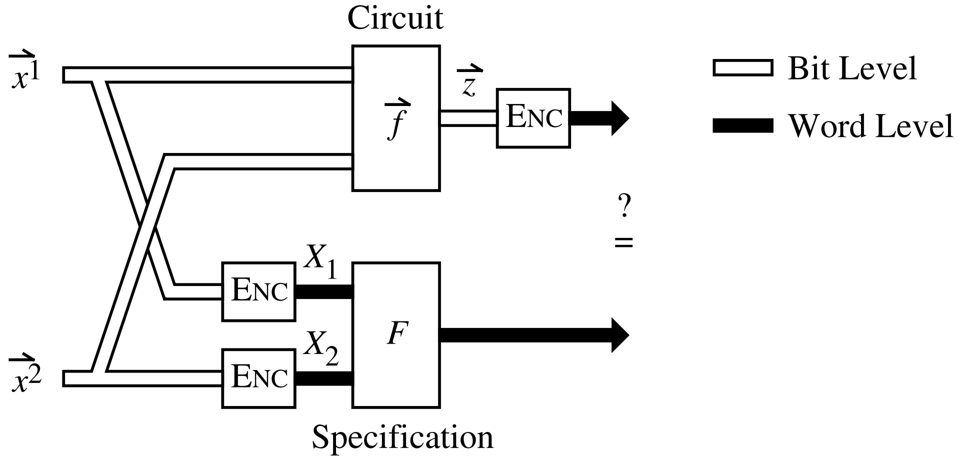


Figure 3.1: **Formulation of Verification Problem.** The goal of verification is to prove a correspondence between a bit-level circuit and a word-level specification

represented by a vector of Boolean functions $\vec{f}$, and the specification, represented by the arithmetic function F. The Boolean functions can correspond to the outputs of a combinational circuit in terms of the primary inputs, or to the outputs of a sequential circuit operated for a fixed number of steps, in terms of the initial state and the input values at each step. Assume that the circuit inputs (and possibly initial state) are partitioned into vectors of binary signals $\vec{x}^1, \ldots, \vec{x}^k$ (in the figure $k = 2$). For each set of signals $\vec{x}^i$, we are given an encoding function ENC_i describing a word-level interpretation of the signals. An example of such an encoding function would be as a 16-bit two's complement integer. The circuit implements a set of Boolean functions over the inputs, denoted by the vector of functions $\vec{f}(\vec{x}^1, \ldots, \vec{x}^k)$. Typically this circuit is given in the form of a network of logic gates. Furthermore, we are given an encoding function ENC_o defining a word-level interpretation of the output. Finally, we are given as specification a arithmetic function $F(X_1, \ldots, X_k)$, where $X_i = Enc_i(\vec{x}^i)$. The task of verification is then to prove the equivalence:

$$\text{ENC}_o(\vec{f}(\vec{x}^1, \ldots, \vec{x}^k)) \quad = \quad F(\text{ENC}_1(\vec{x}^1), \ldots, \text{ENC}_k(\vec{x}^k)) \tag{3.1}$$

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

*BMDs provide a suitable data structure for this form of verification, because they can represent both bit-level and word-level functions efficiently. EVBDDs can also be used for this purpose, but only for the limited class of circuit functions having efficient word-level representations as EVBDDs. By contrast, BDDs can only represent bit-level functions, and hence the specification

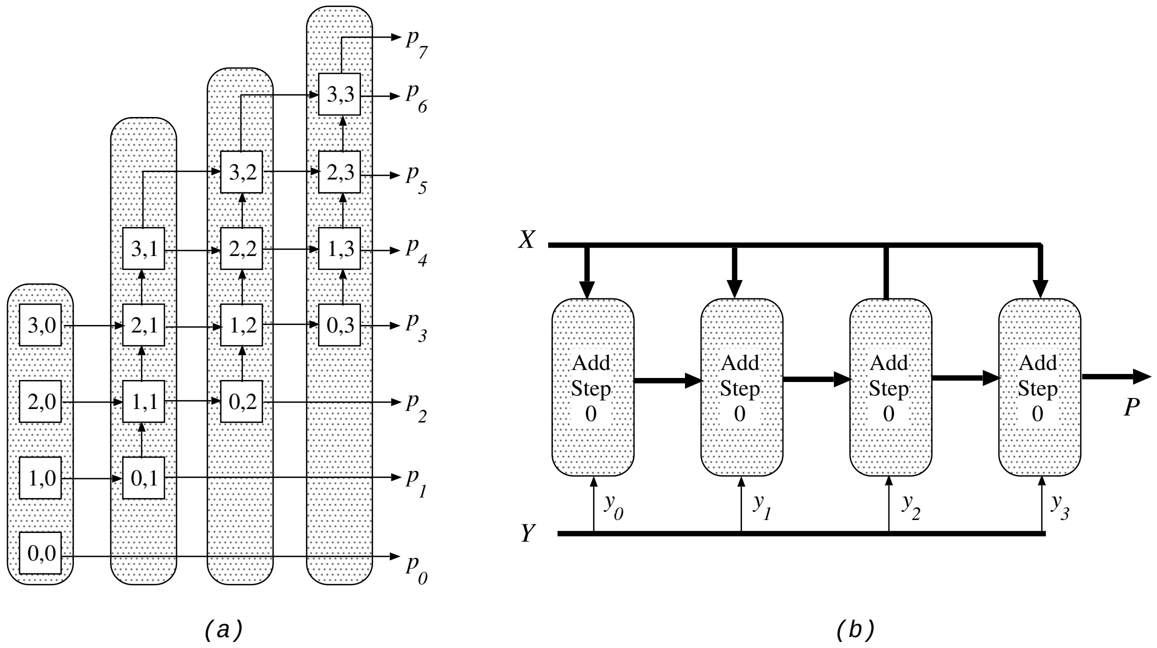


Figure 3.2: **Multiplier Circuit Different Levels of Abstraction** Each square contains an AND gate and a full adder. The vertical rectangles indicate the word-level partitioning yielding the representations shown on the right.

must be expanded into bit-level form. While this can be done readily for standard functions such as binary addition, a more complex function such as binary to BCD conversion would be difficult to specify at the bit level.

3.1.1 Hierarchical Verification

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

Figure 3.2 illustrates the design of two different 4-bit multipliers. Each box labeled i, j in the figure represents a "cell" consisting of an AND gate to form the partial product $x_i \wedge y_j$, and a full adder to add this bit into the product. The vertical rectangles in the figure indicate a word-level partitioning of the circuits, yielding the component interconnection structure shown on the upper right. All word-level data in the circuit uses an unsigned binary encoding. Considering the design labeled "Add-Step", each "Add Step i" component has as input the multiplicand word X, one bit of the multiplier y_i, and a (possibly 0) partial sum input word PI_i. It generates a partial sum word PO_i, where the functionality is specified as $PO_i = PI_i + 2^i \cdot y_i \cdot X$.

Verifying the multiplier therefore involves two steps. First, we must prove that each component implements its specification. Second, we must prove that the composition of the word-level functions matches that of integer multiplication, i.e.,

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

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

3.1.2 Component Verification

The component partitioning allows us to efficiently represent both their bit-level and word-level functions. This allows the test of Equation 3.1 to be implemented directly. As an example, consider the adder circuit having bit-level functions given by the *BMD of Figure 2.6, where this *BMD is obtained by evaluating BDDs from a gate-level representation of the circuit and then translating BDDs into *BMDs. The word-level specification is given by the left-hand *BMD of Figure 2.4. In generating the *BMD from the specification we are also incorporating the requirement that input words X and Y have an unsigned binary encoding. Given that the output is also to have an unsigned binary encoding, we would use our Apply algorithms to convert the bit-level circuit outputs to the word level as:

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

We would then compare the *BMD for P to the one shown on the left-hand side of Figure 2.4.

3.2 Verification System: Arithmetic Circuit Verifier

Based on *BMDs and hierarchical verification methodology, we have built an Arithmetic Circuit Verifier (ACV) to verify integer circuits such as multipliers, dividers, etc. To support

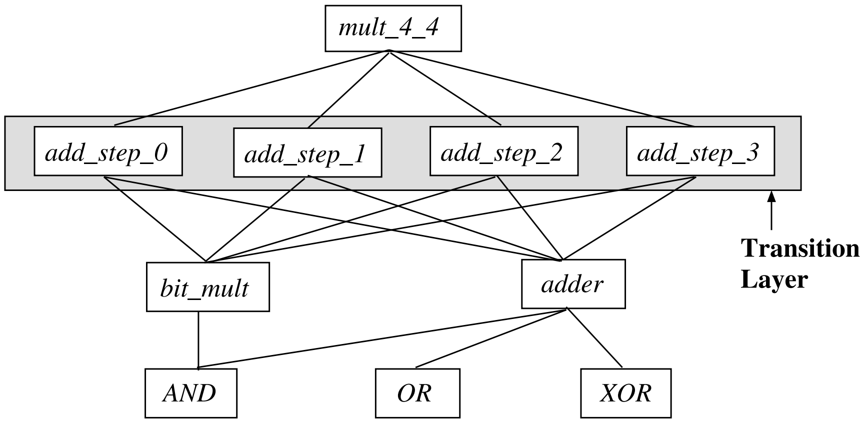


Figure 3.3: **Module hierarchy of 4×4 multiplier.** Each module in the transition layer is the last module with word-level specifications on the path down from the root.

the hierarchical verification methodology, we devised a hardware description language, also called ACV, to describe circuits and their specifications in a hierarchical manner. Each module is composed structurally from other modules and primitive logic gates. In addition, a module can be given the word-level specification consisting of definitions of the numeric encodings of inputs and outputs, as well as the module functionality in terms of arithmetic expressions relating input and output values.

We use a 4×4 array multiplier to illustrate the ACV language and system. This multiplier can be represented by the module hierarchy shown in Figure 3.3. We define the "transition layer", shown as the shaded box in Figure 3.3, as the collection of modules which are the last modules with word-level specifications on the paths down from the root. Modules in or above the transition layer must declare their word-level specifications, as well as their structural definitions. Modules below the transition layer just declare their structural definitions. Modules in the transition layer abstract from the bit-level, where the structure consists of logic gates (sub-module will be evaluated recursively), to a word-level representation, where the structure consists of blocks interconnected by bit-vectors encoding numeric values.

Figure 3.4 shows the ACV description of the top module of a 4×4 array multiplier. The definition of a module is encompassed between keywords "MODULE" and "ENDMODULE". First, the module name and the names of signals visible from outside of this module must be given as shown in the first line of Figure 3.4. The module is declared as $mult\_4\_4$ with three signals x, y, and p. Then, the width of these signals are declared in the VAR section. Both x and y are declared as 4 bits wide, and p are 8 bits.

For each module, section INTERNAL and STRUCTURE define the circuit connections among

MODULE $mult\_4\_4(x, y, p)$	
VAR	p[8],x[4],y[4];
ENCODING	P = (unsigned) p;
	X = (unsigned)x;
	Y = (unsigned)y;
FUNCTION	P == X*Y;
VERIFY	P == X*Y;
ORDERING	x,y;
INTERNAL	$s1$[4],$s2$[6],$s3$[7];
STRUCTURE	add_step_0(y[0],x,$s1$);
	add_step_1(y[1],x,$s1$,$s2$);
	add_step_2(y[2],x,$s2$,$s3$);
	add_step_3(y[3],x,$s3$,p);
ENDMODULE	

Figure 3.4: **ACV code for Module $mult\_4\_4$ of a 4$\times$4 multiplier.**

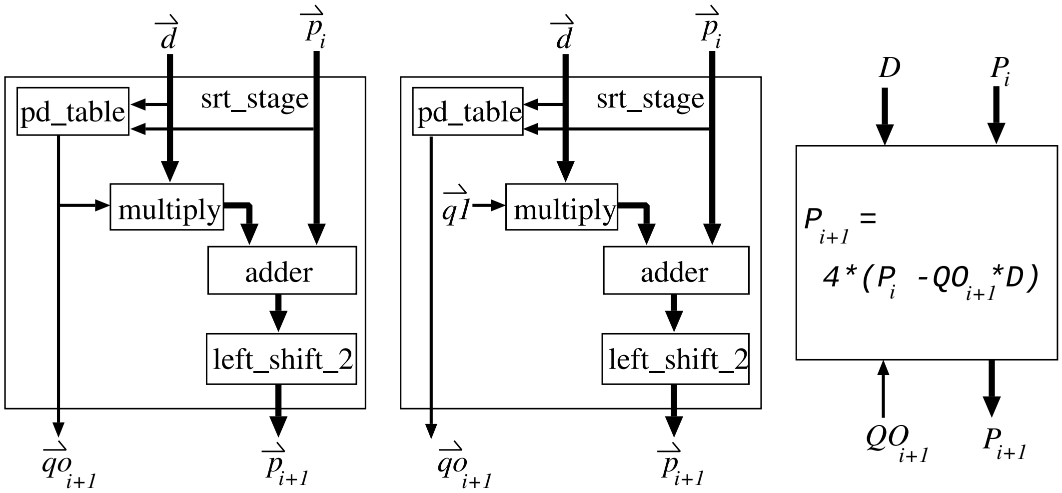
logic gates and sub-modules. The INTERNAL section declares the names and widths of internal vector signals used in the STRUCTURE section to connect the circuit. Vector $s1$, $s2$ and $s3$ are declared as 4, 6 and 7 bits, respectively. There are two types of statements in the STRUCTURE section. First, the assignment statements, shown in lines 1, 2, 4, 5 and 6 in the STRUCTURE section of Figure 3.6, are used to rename part of a signal vector, or to connect the output of a primitive logic gate. Second, the module instantiation statements, shown in the STRUCTURE section of Figure 3.4, declare which signals are connected to the referenced modules. Note that we do not distinguish inputs from outputs in module instantiation statements and module definitions. As we shall see, it is often advantageous to shift the roles of inputs and outputs as we move up in the module hierarchy. The ACV program will distinguish them during the verification process based on the information given in the specification sections.

To give the word-level specification for a module, sections ENCODING, FUNCTION, VERIFY and ORDERING are required in the module definition. The ENCODING section gives the numeric encodings of the signals declared in the VAR section. For example, vector p is declared as having an unsigned encoding and its word-level value is denoted by P. The allowed encoding types are: unsigned, two's complement, one's complement, and sign-magnitude. The FUNCTION section gives the word-level arithmetic expressions for how this module should be viewed by modules higher in the hierarchy. For example, if module $mult\_4\_4$ were used by a higher level module, its function would be to compute output P as the product of inputs X and Y. In general, the variable on the left side of "==" will be treated as output and the variables on the right side will be treated as inputs. The VERIFY section declares the specification which will be verified against its circuit implementation. In the multiplier example, the module specification is the same as its function. In other cases, such as the SRT divider example in next section, these two may differ to allow a shifting of viewpoints as we move up in the hierarchy. The ORDERING section not only specifies the BDD variable ordering for the inputs but also defines which signals should be treated as inputs during the verification of this module. The variable ordering is very important to verification, because our program does not currently do dynamic variable reordering.

The ACV program proceeds recursively beginning with the top-level module. It performs four tasks for each module. First, it verifies the sub-modules if they have word-level specifications. Second, it evaluates the statements in the STRUCTURE section in the order of their appearance to compute the output functions. For a module in the transition layer, this involves first computing a BDD representation of the individual module output bits by recursively evaluating the sub-module's statements given in their STRUCTURE sections. These BDDs are then converted to a vector of bit-level *BMDs, and then a single word-level *BMD is derived by applying the declared output encoding. For a module above the transition layer, evaluation involves composing the submodule functions given in their FUNCTION sections. Third, ACV checks whether the module specification given in the VERIFY section is satisfied, Finally, it checks whether the specification given in the VERIFY section implies the module function given in the FUNCTION section. A flag is maintained for each module indicating whether this module has been verified. Thus, even if a module is instantiated multiple times in the hierarchy, it will be verified only once.

For example, the verification of the 4-bit array multiplier in Figure 3.4 begins with the verification of the four add_step modules. For each one, the structural definition as well as the structural definitions it references are evaluated recursively using BDDs to derive a bit-level representation of the module output. These BDDs are converted to *BMDs, and then a word-level *BMD is derived by computing the weighted sum of the bits. ACV checks whether the circuit matches the specification given in its VERIFY section. The specification of add_step module i is $Out = In + 2^i * y * X$, where In is a partial sum input (0 for i=0), y is a bit of the multiplier and Out is the output of the module.

Assuming the four add_step modules are verified correctly, ACV derives a *BMD representation of the multiplier output. It first creates *BMD variables for the bit vectors x and y (4 each), and computes *BMD representations of X and Y by computing weighted sums of these bits. It evaluates the add_step instantiations to derive a word-level representation of module output. First, it computes $s1$ by evaluating the FUNCTION statement $Out = y * X$ of module add_step_0 for the bindings $y = y_0$ and $X = X$. Then it computes $s2$ by evaluating the FUNCTION statement $Out = In + 2 * y * X$ of module add_step_1 for the bindings $In = s1$, $y = y_1$, and $X = X$. This process continues for the other two modules, yielding a *BMD for



(a) (b) (c)

Figure 3.5: **Block level representation of SRT divider stage from different perspectives.** (a) The original circuit design. (b) The abstract view of the module, while verifying it. (c) The abstract view of the module, when it is referenced.

P equivalent to $P = (((y_0 * X) + 2 * y_1 * X) + 2^2 * y_2 * X) + 2^3 * y_3 * X$. Note that whether a module argument is an input or an output is determined by whether it has a binding at the time of module instantiation. ACV then compares the *BMD for P to the one computed by evaluating $X * Y$ and finds that they are identical. Finally, checking whether the specification in the VERIFY section implies the functionality given in the FUNCTION section is trivial for this case, since they are identical.

3.3 Additional Techniques

In order to verify the integer circuits, which generate more than 1 word-level output such as carry-save adders(CSA) and dividers, we developed some techniques within hierarchical verification approach. We use radix-4 SRT division as an example to explain several additional verification methodologies, and to illustrate the use of these methodologies in ACV language.

A divider based on the radix-4 SRT algorithm is an iterative design maintaining two words of state: a partial remainder and a partial quotient, initialized to the dividend and 0, respectively. Each iteration extracts two bits worth of quotient, subtracts the correspondingly weighted value of the divider from the partial remainder, and shifts the partial remainder left by 2 bit positions. The logic implementing one iteration is shown in Figure 3.5.a, where we do not show two registers storing partial remainder and partial quotient. The inputs are divisor $\vec{d}$ and partial remainder $\vec{p_i}$, and the outputs are the extracted quotient digit $\vec{qo}_{i+1}$ (ranging from -2 to 2) and the updated partial remainder $\vec{p}_{i+1}$. The PD table, used to look up the quotient digits based on the truncated values of the divisor and the partial remainder, is implemented in logic gates derived from a sum of products form. After the iterations, the set of obtained quotient digits is converted into the actual quotient by a quotient conversion circuit.

First, we prove the correctness of one iteration of the circuit. The specification is given in [36] and is shown as Equation 3.2. This specification states that for all legal inputs (i.e., satisfying the range constraint) the outputs also satisfy the range constraint, and that the inputs and outputs are properly related. This specification captures the essence of the SRT algorithm.

$$\begin{aligned}&(-8D \leq 3P_i \leq 8D) \rightarrow \\ &\{(-8D \leq 3P_{i+1} \leq 8D) \wedge [P_{i+1} == 4(P_i - QO_{i+1} * D)]\}\end{aligned} \tag{3.2}$$

This specification contains word-level function comparisons such as $\leq$ and == as well as Boolean connectives $\wedge$ and $\rightarrow$. In [30], a branch-and-bound algorithm is proposed to do word-level comparison operations for HDDs. It takes two word-level functions and generates a BDD representing the set of assignments satisfying the comparison operation. We adapted their algorithm for *BMDs to allow ACV to perform the word-level comparisons. Once these "predicates" are converted to BDDs, we use BDD operations to evaluate the logic expression.

If Equation 3.2 is used to verify this module, the running time will grow exponentially with the word size, because the time to convert output $\vec{p}_{i+1}$ in Figure 3.5(a) from a vector of Boolean functions into a word-level function grows exponentially with the word size. The reason is that $\vec{p}_{i+1}$ depends on output vector $\vec{qo}_{i+1}$ which itself has a complex function. We overcome this problem by cutting off the dependence of $\vec{p}_{i+1}$ on $\vec{qo}_{i+1}$ by introducing an auxiliary vector of variables $\vec{q1}$, shown in Figure 3.5(b). One can view this as a cutting of the connection from the PD table to the multiply component in the circuit design. Now, the task of verifying this module becomes to prove that Equation 3.3 holds:

$$\begin{aligned}&(-8D \leq 3P_i \leq 8D \wedge QO_{i+1} == Q1) \rightarrow \\ &\{(-8D \leq 3P_{i+1} \leq 8D) \wedge [P_{i+1} == 4(P_i - Q1 * D)]\}\end{aligned} \tag{3.3}$$

In the actual design, the requirement that $QO_{i+1} == Q1$ is guaranteed by the circuit structure. Hence Equation 3.3 is simply an alternate definition of the module behavior. By this methodology, the computing time of verifying this specification is reduced dramatically with a little overhead (the computing time of performing $QO_{i+1} == Q1$ and an extra AND operation). The major difference between this cutting methodology and the hierarchical partitioning is that the latter decomposes the specification into several sub-specifications, but the former only

```
MODULE srt_stage(p, d, qo, p1)
VAR          qo[3],q1[3],p[9],d[6],p1[9];
EQUIVALENT   (qo,q1);
ENCODING     P = (twocomp) p;
             P1 = (twocomp)p1;
             D = (unsigned)d;
             QO = (signmag)qo;
             Q1 = (signmag)q1;
FUNCTION     P1 == 4*(P-QO*D);
VERIFY
      (3*P ≤ 8*D & 3*P ≥ -8*D & Q1 == QO & D ≥ 2**5)
      → (3*P1 ≤ 8*D & 3*P1 ≥ -8*D & P1 == 4*(P-Q1*D));
ORDERING     q1,p,d;
INTERNAL     ph[7],dh[4],t[9],nqd[9],r[10];
STRUCTURE    ph = p[2 .. 8];
             dh = d[1 .. 4];
             pd_table(ph, dh, qo);
             w1 = qo[0];
             w2 = qo[1];
             neg = not(qo[2]);
             shifter(d, w1, w2, t);
             negater(t, neg, nqd);
             adder(p, nqd, neg, r);
             left_shift_2(r, p1);
ENDMODULE
```

Figure 3.6: **ACV code for Module srt_stage.**

introduces auxiliary variables to simplify the computation. We can also apply this methodology to verify the iteration stage of such similar circuits as restoring division, restoring square root and radix-4 SRT square root.

Module srt_stage, shown in Figure 3.6, implements the function of one SRT iteration for a 6×6 divider using the ACV language. Vector variables, p, d, qo and $p1$ in Figure 3.6, represent signal vectors, $\vec{p}_i$, $\vec{d}$, $\vec{qo}_{i+1}$ and $\vec{p}_{i+1}$ in Figure 3.5(a), respectively. Their encoding and ordering information is given in the relevant sections. Modules $shifter$ and $negater$ implements module multiply in Figure 3.6(a). Since *BMDs can only represent integers, we must scale all numbers so that binary point is at the right. We specify one additional condition in the specification:

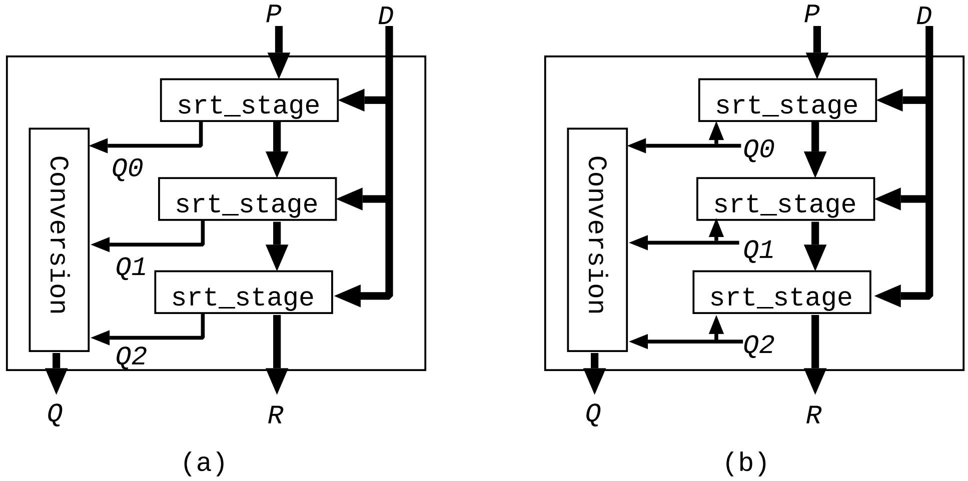


Figure 3.7: **Block level representation of a 6×6 SRT divider from two different perspectives.** (a) The original circuit design. (b) the abstract view of the module, while verifying it.

that the most significant bit of the divider must be 1, by the term $D \geq 2$**5.

The support for our "cutting" methodology arises in several places. First, vector $q1$ is declared in the VAR and ORDERING sections with the same size as qo, and is therefore treated as a "pseudo input", i.e., an input invisible to the outside. Then, the equivalence of signals qo and $q1$ is declared in the EQUIVALENT section. The original signal qo must appear first in the pair. While evaluating the statements in the STRUCTURE section, ACV automatically uses $q1$'s value instead of qo's value for signal qo once signal qo has been assigned its value. For example, all appearances of signal qo after the pd_table instantiation in Figure 3.6 will use $q1$'s value (a *BMD $Q1$ using three Boolean variables) instead of its original value (a *BMD function of inputs P and D) when evaluating these statements. Finally, the encoding method of $q1$ is declared the same as q0 and Equation 3.3 is used in the VERIFY section instead of Equation 3.2.

Figure 3.7(a) shows the block level representation of a 6×6 SRT divider. Since module srt_stage performs a cycle of SRT division, we instantiate it multiple times, effectively unrolling the sequential SRT division into a combinational one, and compose them with another module $Conversion$ which takes the set of quotient digits generated from the stages and converts them into a quotient vector with an unsigned binary representation. The divider takes two inputs P and D, goes through 3 srt_stage and 1 $Conversion$ modules, and generates the outputs Q and R. Module $Conversion$ takes a set of quotient digits, generated from the

MODULE $srt\_div\_6\_6(p, d, q, r)$
VAR $p[6]$, $d[6]$, $q[6]$, $r[9]$, $q0[3]$, $q1[3]$, $q2[3]$;
ENCODING P = (unsigned) p;
D = (unsigned) d;
Q = (unsigned) q;
R = (twocomp) r;
$Q0$ = (signmag) $q0$;
$Q1$ = (signmag) $q1$;
$Q2$ = (signmag) $q2$;
FUNCTION R == 2**6 * P - 4*D*Q;
VERIFY (3*$P \leq$ 8*D & 3*$P \geq$ -8*D & $D \geq$ 2**5) $\rightarrow$
((2**6 * P) == 4*D*Q + R);
ORDERING $p, d, q2, q1, q0$;
INTERNAL $p0[9]$, $p1[9]$, $p2[9]$;
STRUCTURE $p0[0 .. 5]$= p;
$p0[6 .. 8]$= 0;
$srt\_stage(p0, d, q0, p1)$;
$srt\_stage(p1, d, q1, p2)$;
$srt\_stage(p2, d, q2, r)$;
$Conversion(q, q0, q1, q2)$;
ENDMODULE

Figure 3.8: **ACV description of Module $srt\_div\_6\_6$.**

srt_stages, and converts them into a vector in the unsigned binary form. Assume module $Conversion$ takes inputs $\vec{q_0}$, ..., $\vec{q_n}$, and produces the output $\vec{q}$. The specification of this module is $Q = Q_n + 4 * Q_{n-1} + ... + 4^n * Q_0$, where Q and Q_i are the word-level representations of $\vec{q}$ and $\vec{q_i}$, $0 \leq i \leq n$.

With the partitioning shown in Figure 3.7(a), we cannot directly apply hierarchical verification, because the outputs of module srt_stage do not have unique functional definitions. The redundant encoding of the quotient digits in the SRT algorithm allows, in several cases, a choice of values for the quotient digits. Fortunately, we do know the relation between inputs and outputs: $P_{i+1} = 4 * (P_i - QO_{i+1} * D)$. We exploit the fact that the correctness of the overall circuit behavior does not depend on the individual output functions, but rather on their relation. Therefore we can apply a technique similar to one used to verify circuits with carry-save adders[21] treating the quotient output as an input when this module is instantiated. Figure 3.5(c) shows this abstract view of the srt_stage module when it is referenced. The abstract view of the SRT divider is then changed as shown in Figure 3.7(b), and described in ACV as shown in Figure 3.8. The quotient output vectors $\vec{q2}$, $\vec{q1}$ and $\vec{q0}$ (denoted by $Q2$, $Q1$ and $Q0$ for the word-level representation) of three srt_stage modules are changed to pseudo inputs by declaring them in the VAR, ENCODING and ORDERING sections. With this additional information, the circuit is effectively changed from Figure 3.7(a) to Figure 3.7(b) without modifying the physical connections.

Assume both srt_stage and $conversion$ modules are verified. During verification of module $srt\_div\_6\_6$, when ACV evaluates the first srt_stage statement, vector $\vec{q0}$ has its word value $Q0$ and is treated as an input to module srt_stage to compute the value of vector $p1$. Therefore, the value of vector $\vec{p1}$ is $4 * (P - Q0 * D)$ and this becomes an input to the second srt_stage. ACV repeats the same procedure for the other srt_stage statements to compute the value of R which now depends on P, D, $Q0$, $Q1$ and $Q2$. It also computes the value of Q, which depends on $Q0$, $Q1$ and $Q2$, from module $Conversion$. The specification of this 6×6 SRT Radix-4 divider we verified is: $(-8D \leq 3P \leq 8D \wedge D \geq 2^5) \rightarrow (P * 2^6 == 4 * Q * D + R)$. The constraints , $-8D \leq 3P \leq 8D$ and $D \geq 2^5$, required for the first srt_stage, specify the input range constraints. Under these input constraints, the circuit performs the division, specified by the relation $P * 2^6 = 4 * Q * D + R$. Since $Q0$, $Q1$ and $Q2$ can be arbitrary values, we cannot verify the divider's output range constraint: $-8D \leq 3R \leq 8D$. It can be deduced manually from the initial condition and the input and output constraints of the srt_stage modules.

3.4 Experimental Results

All of our results were executed on a Sun Sparc Station 10. Performance is expressed as the number of CPU seconds and the peak number of megabytes (MB) of memory required.

Table 3.1 shows the results of verifying a number of multiplier circuits with different word sizes. Observe that the computational requirements grow quadratically, caused by quadratical growth of the circuit size, except Design "seq" which is linear. The design labeled "CSA" is based on the logic design of ISCAS'85 benchmark C6288 which is a 16-bit version of the circuit. These results are especially appealing in light of prior results on multiplier verification. A brute force approach based on BDDs cannot get beyond even modest word sizes. Yang *et al* [97] have successfully built the OBDDs for a 16-bit multiplier, requiring over 40 million vertices. Increasing the word size by one bit causes the number of vertices to increase by a factor of approximately 2.87, and hence even more powerful computers will not be able to get much beyond this point.

Compared with other multipliers, the verification of CSA multiplier is slower, because the verification of a carry-save adder is slower than a carry-propagate adder. The designs labeled

Sizes	16x16	32x32	64x64	128x128	256x256
CSA	4.68(sec)	20.08	78.55	351.18	1474.55
	0.83(MB)	1.19	2.31	6.34	21.41
Booth	2.37	8.18	27.47	128.87	535.18
	0.77	1.09	2.12	5.94	20.41
BitPair	1.90	5.76	15.43	69.68	288.70
	0.74	0.93	1.53	3.56	11.12
Seq	1.08	2.41	5.30	14.35	36.13
	0.70	0.76	0.96	1.41	2.75

Table 3.1: **Verification Results of Multipliers.** Results are shown in seconds and Mega Bytes.

Sizes	16x16	32x32	64x64	128x128	256x256
srt-div	16.25(sec)	23.58	40.40	109.63	398.68
	1.16(MB)	1.47	2.19	4.47	10.47
r-div	5.53	26.02	153.13	1131.82	8927.18
	0.71	0.89	1.56	4.22	15.34
r-sqrt	8.35	54.85	320.60	2623.11	20991.35
	0.77	1.12	3.12	14.97	98.31

Table 3.2: **Verification Results of Dividers and Square Roots.** Results are shown in seconds and Mega Bytes.

"Booth" and "BitPair" are based on the Booth and the modified Booth algorithms, respectively. Verifying the BitPair circuits takes less time than the Booth circuits, because it has only half the stages. Comparing these results with the results given in [21], we achieve around 3 to 4 times speedup, because we exploited the sharing in the module hierarchy.

Design "Seq" is an unrolled sequential multiplier obtained by defining a module corresponding to one cycle of operation and then instantiating this module multiple times. The performance of Design "Seq" is another example to demonstrate the advantage of sharing in our verification methodology. The complexity of verifying this multiplier is linear in the word size, since the same stage is repeated many times.

Table 3.2 shows the computing time and memory requirement of verifying divider and square root circuits for a variety of sizes. We have verified divider circuits based on a restoring method and the radix-4 SRT method. For the radix-4 SRT divider, the computing time grows quadratically, because we exploit the sharing property of the design and apply hierarchical verification as much as we can. For both restoring divide and square root, the computing time grows cubically in the word size. This complexity is caused by verifying the subtracter. While converting the vector of BDD functions into word-level *BMD function for the output of the subtracter, the intermediate *BMD size and operations grow cubically, although, the size of final *BMD function is linear.

3.5 Related Work

Our approach has to partition the circuits into hierarchical forms. However, some design may not have hierarchical structures for us to verify. For example, the optimized design usually is in the flattened netlist format. To overcome this constraint, Hamaguchi *et al* [52] proposed a backward substitution method to compute the output of integer multipliers without any circuit knowledge. For a 64×64 multiplier, they reported 22,340 seconds of CPU time, while our approach only requires 27.47 seconds.

Clarke *et al* [30, 33] presented word-level SMV based on BDDs for Boolean functions, HDDs for integer functions and a layered backward substitution method (a variant of hamaguchi's method) [29]. For integer multipliers, their complexity grows cubically, but the constant factor is much smaller than Hamaguchi's. Chen *et al* [29] have applied word-level SMV to verify arithmetic circuits in one of Intel's processors. In this work, floating-point circuits were partitioned into several sub-circuits whose specifications can be expressed in terms of integer operations, because HDDs can not represent floating-point functions efficiently. Each sub-circuits were verified in a flattened manner. They reported 508 seconds to verify a 64 bit multiplier and 194 seconds to verify a 64-bit sequential divider on a HP 9000 workstation with 256MB, which is at least 2.5 times faster than Sun Sparc 10.

Both Hamaguchi's and layered backward substitution approach have cubical growth for the correct multipliers, while our approach has quadratic growth. In general, compared with approaches with backward substitution methods, our approach achieves greater speedup for the larger circuits. For the incorrect multipliers, both backward substitution methods cannot build *BMDs or HDDs for the outputs, because *BMDs or HDDs explodes exponentially in size. However, our approach can easily detect the bugs while verifying the lower modules of the design.

Chapter 4

Representation for Floating-Point Functions

When applied to verify floating-point circuits, our hierarchical approach with *BMDs has two major problems. First, the decision diagram explodes in size, when the output of the rounding module is computed from the word-level functions obtained from the previous module of the circuit. This problem will be illustrated more detail in Chapter 5. Second, *BMD and/or HDDs cannot represented floating-point functions without the use of rational edge weights. Thus, a new representation and verification methodology is needed to verify the floating-point circuits such as adders and multipliers. In this chapter, we present a representation, multiplicative Power HDD (*PHDD), to address the first problem. *PHDDs can represent floating-point functions efficiently and can be used to verify the floating-point circuits.

The rest of this chapter is organized as followings. Section 4.1 illustrates the limitations of *BMDs and HDDs in representing floating-point functions. The data structure for *PHDDs are described in Section 4.2. The *PHDD representations for integer and floating-point functions are shown in Section 4.3. Section 4.4 shows experimental results to illustrate the advantages of *PHDDs compared with *BMDs.

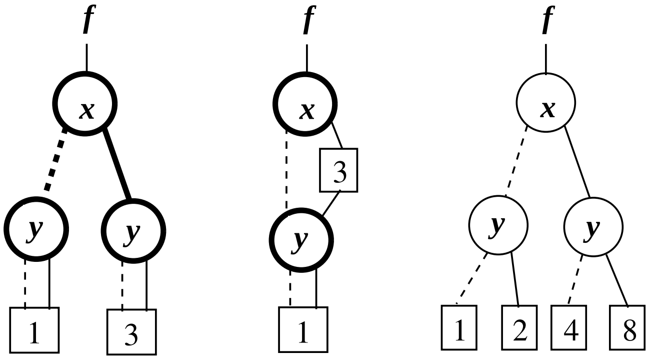
4.1 Reasons for A New Diagrams

To verify floating-point circuits, we must have a word-level diagram that can represents floating-point functions efficiently, especially for IEEE floating-point encoding: $(-1)^{sign} \cdot 2^{(X-bias)} \cdot M$, where X and M use unsign binary encodings and bias is a constant. As summarized in

Section 1.2.1, many word-level decision diagrams have been proposed to represent word-level functions efficiently. However, these diagrams do not have compact representations for floating-point functions.

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

(a)



(b) (c) (d)

Figure 4.1: **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.

We used Figure 4.1 and 4.2 to illustrate the limitations of BMDs and HDDs. Figure 4.1 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 [34]). Observe that if variables x and y are viewed as bits forming 2-bit binary number, X=y+2x, then the function f can be rewritten as $f = 2^{(y+2x)} = 2^X$. 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 4.1.b shows the BMD representation. To construct this graph, we apply Equation 2.2 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 4.1.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 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 $f = 2^{X-bias}$, where $bias$ is a constant, because they can only represent integer functions without introducing the rational numbers in the edges or leaf nodes as shown in Figure 4.2. However, the overhead of storing and manipulating the rational numbers in the edges or leaf nodes make them less attractive for representing floating-point functions.

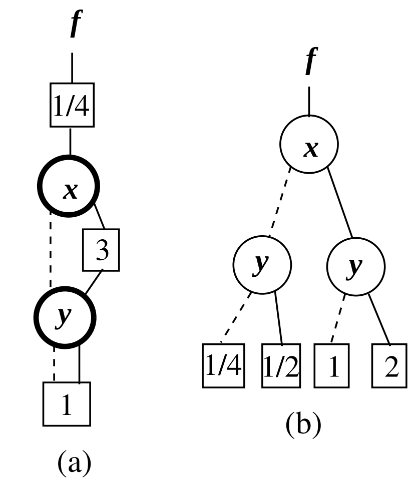


Figure 4.2: ***BMDs and HDDs for function** $f = 2^{X-2}$**, where** $X = x + 2y$**.** (a) *BMDs, (d) HDDs.

4.2 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.

4.2.1 Edge Weights

*PHDDs use three of HDD's six decompositions as expressed in Equation 2.5. 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 2.5 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.

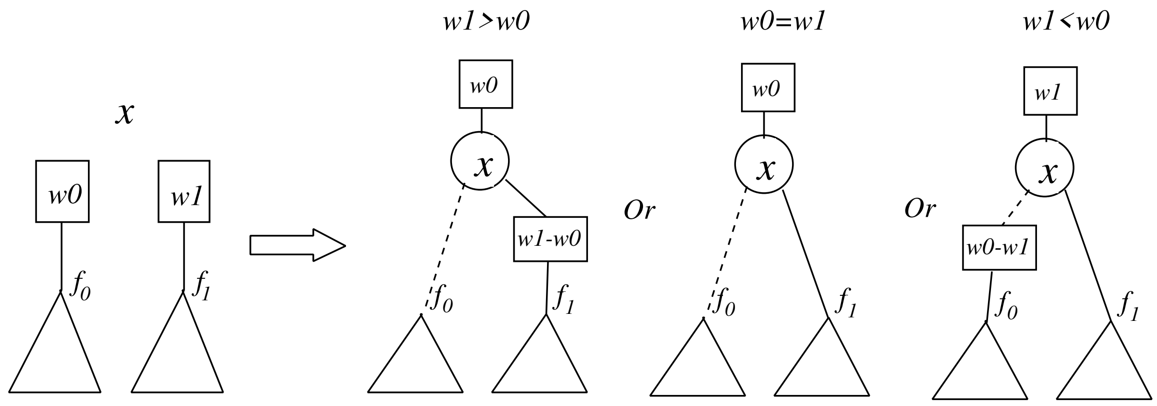


Figure 4.3: **Normalizing the edge weights.**

In addition to the HDD reduction rules [30], 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 denote 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 [20]. Figure 4.3 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 4.3. 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.

4.2.2 Negation Edge

Negation edges are commonly used in BDDs [9] and KFDDs [42], 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 the *PHDD data structure, we use the low order bit of each pointer to denote negation, as is done with the complement edges of BDDs. To maintain a canonical form, we must constrain the use of negation edges. Unlike KFDDs [42], 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 a canonical form for *PHDDs.

4.3 Representation of Word-Level Functions

*PHDDs can effectively represent word-level 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.

4.3.1 Representation of Integer Functions

*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: $x_{n-1}, \ldots, x_1$, x_0. These variables can be considered to represent an integer X according to some encoding, e.g., unsigned binary or two's complement. Figure 4.4 illustrates the *PHDD representations of several common encodings for integers. In our drawing of *PHDDs, we indicate the edge weight and leaf node in square boxes with thick and thin lines, respectively. Edge weight i represents 2^i and Unlabeled edges have weight 0 (2^0). An unsigned number is encoded as a sum of weighted bits. The *PHDD representation has a simple linear structure where the leaf values are formed by the corresponding edge weight and leaf 1 or 0. For representing signed numbers, we assume x_{n-1} is the sign bit. The two's complement encoding has a *PHDD representation similar to that of unsigned integers, but with bit x_{n-1} having weight -2^{n-1} represented by the edge weight $n-1$ and the negation edge. Sign-magnitude integers also have *PHDD representations of linear complexity, but with the constant moment with respect to x_{n-1} scaling the remaining unsigned number by 1, and the linear moment scaling the number by -2 represented by edge weight 1 and the negation edge. In evaluating the function for $x_{n-1} = 1$, we would add these two moments, effectively scaling the number by -1. Note that it is more logical to use Shannon decomposition for the sign bit.

Figure 4.4 also illustrates the *PHDD representations of several common arithmetic operations on integer data. Observe that the sizes of the graphs grow only linearly with the word size n. Integer addition can be viewed as summing a set of weighted bits, where bits x_i and y_i both have weight 2^i represented by edge weight i. Integer multiplication can be viewed as summing a set of partial products of the form $x_i 2^i Y$. In summary, while representing the integer functions, *PHDDs with positive Davio decompositions usually will get the most compact representation among these three decompositions.

4.3.2 Representation of Floating-Point Numbers

Let us consider the representation of floating-point numbers by IEEE standard 754. For example, 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:

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

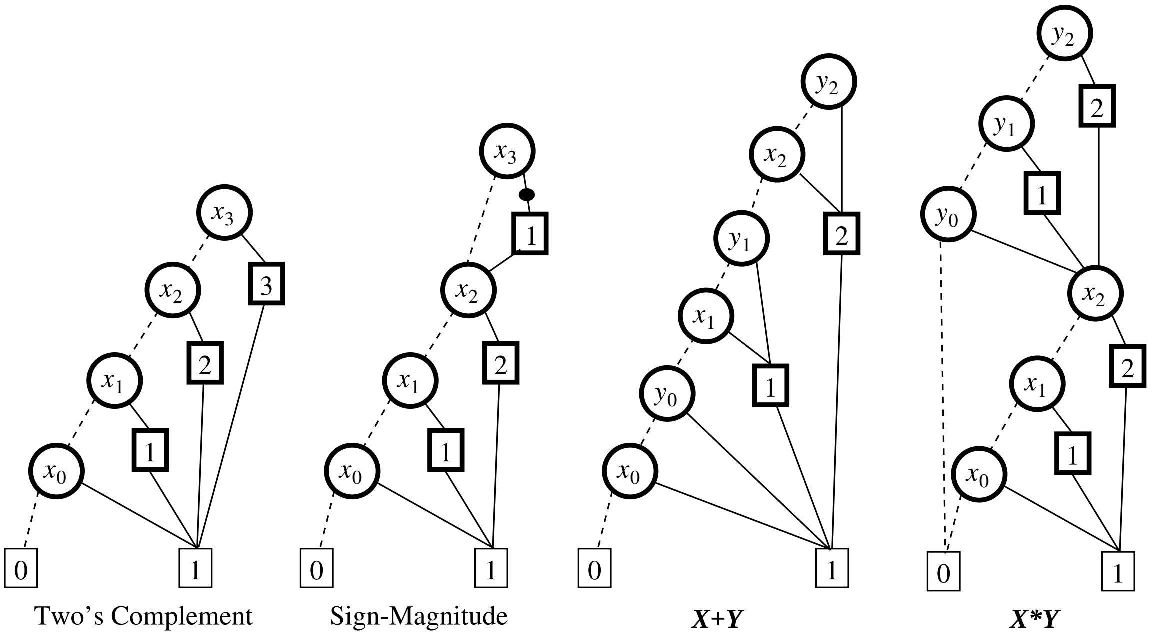


Figure 4.4: ***PHDD Representations of Integers and Integer operations.** Each variable uses positive Davio decomposition. The graphs grow linearly with word size.

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

Figure 4.5 shows *PHDD representations for 2^{EX} and 2^{EX-B} using different decompositions. To represent function c^{EX} (in this case $c = 2$), *PHDDs express the function as a product of factors of the form $c^{2^i ex_i} = (c^{2^i})^{ex_i}$. 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.

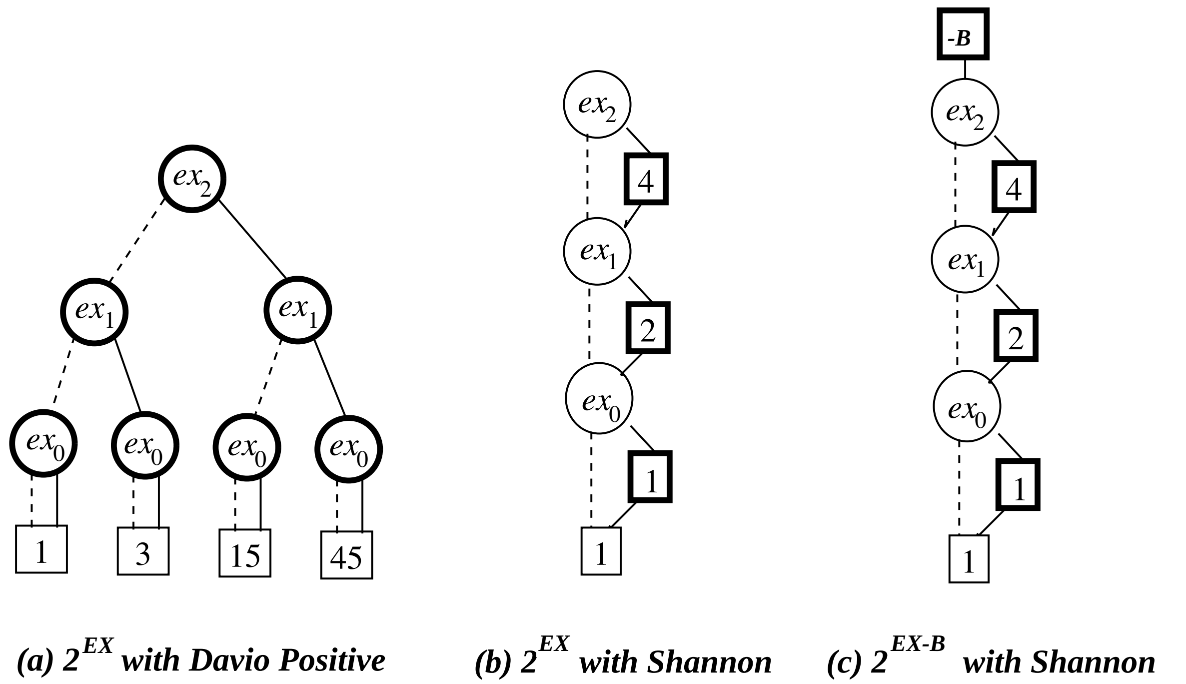


Figure 4.5: ***PHDD Representations of 2^{EX} and 2^{EX-B}.** The graph grows linearly in the word size with Shannon, but grows exponentially with positive Davio.

Figure 4.6 shows the *PHDD representations for the floating-point encoding, where EX has 3 bits, X has 4 bits and the bias B is 3. The sign S_x and $\vec{ex}$ variables use Shannon decomposition, while variables $\vec{x}$ use positive Davio. Figure 4.6.a shows the *PHDD representation for the sign bit $(-1)^{S_x}$. When S_x is 0, the value is 1; otherwise, the value is -1 represented by the negation edge and leaf node 1. Figure 4.6.b shows the *PHDD representation for the exponent part 2^{EX-3}. The graph is more complicated than Figure 4.5.c, because, in the floating-point encoding, when $EX = 0$, the value of the exponent is $1 - B$, instead of $-B$. Observe that each exponent variable, except the top variable ex_2, has two nodes: one to represent the denormal number case and another to represent normal number case. Figure 4.6.c shows the representation for the mantissa part $0.X$ obtained by dividing X by 2^{-3}. Again, the division by powers of 2 requires just adding the edge weight on top of the original graph. Figure 4.6.d shows the representation for the mantissa part $1.X$ which is the sum of $0.X$ and 1. The weight (2^{-3}) of the least significant bit is extracted to the top and the leading bit 1 is represented by the path with all variables set to 0. Finally, Figure 4.6.e shows the *PHDD representation for the complete floating-point encoding. Observe that negation edges reduce the graph size by half. The outlined region in the figure denotes the representation for denormal numbers. The rest of the graph represents normal numbers. Assume the exponent is n bits and the mantissa is m bits. Note that the edge weights are encoded into the node structure in our implementation, but the top edge weight requires an extra node. It can be shown that the total number of *PHDD nodes for the floating point encoding is $2(n + m) + 3$. Therefore, the size of the graph grows linearly with word size. In our experience, it is best to use Shannon decompositions for the sign and exponent bits, and positive Davio decompositions for the mantissa bits.

4.3.3 Floating-Point Multiplication and Addition

This section presents floating-point multiplication and addition based on *PHDDs. Here, we show the representations of these 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: $(-1)^{S_x \oplus S_y} \times 2^{-2B} \times$

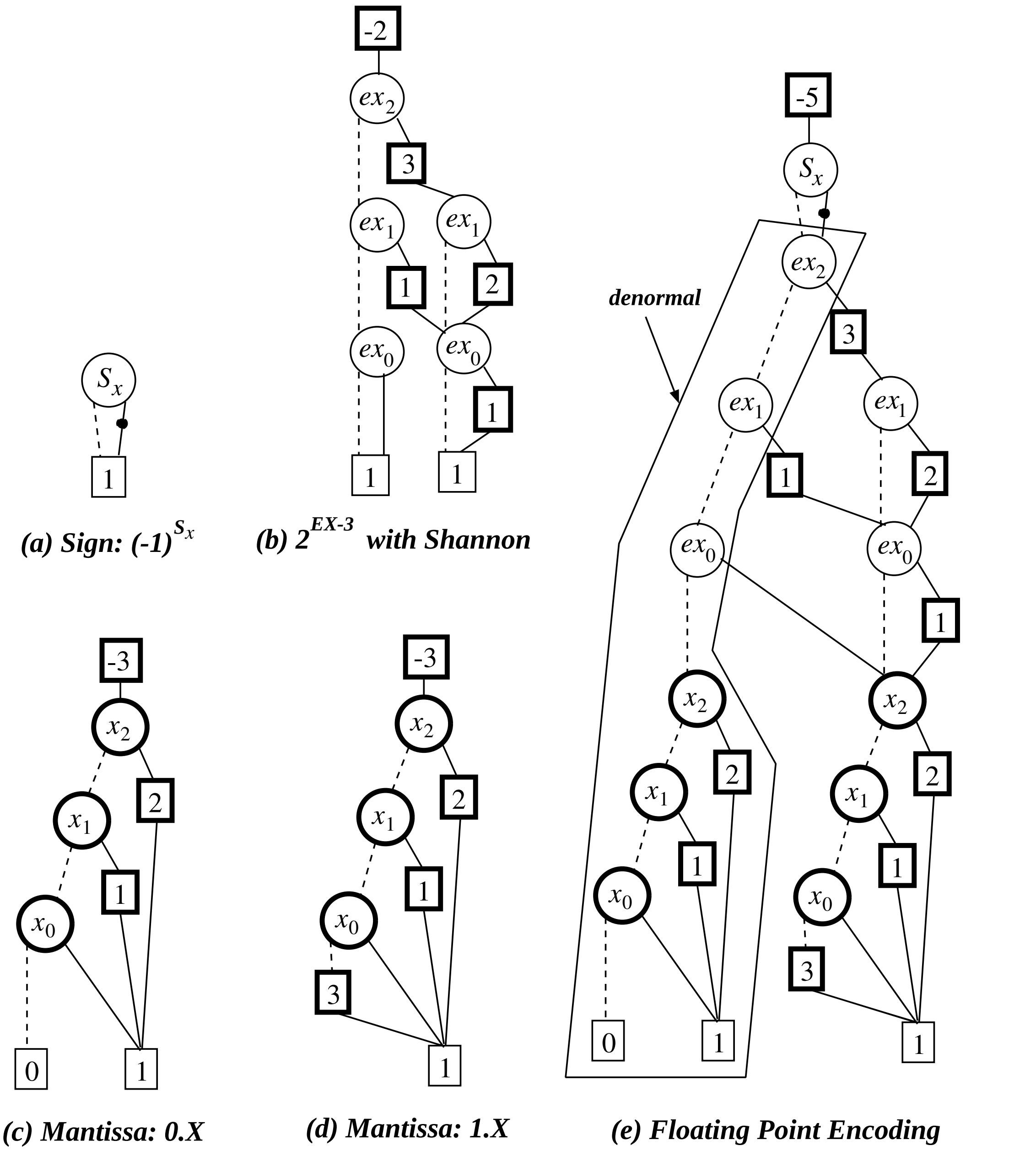
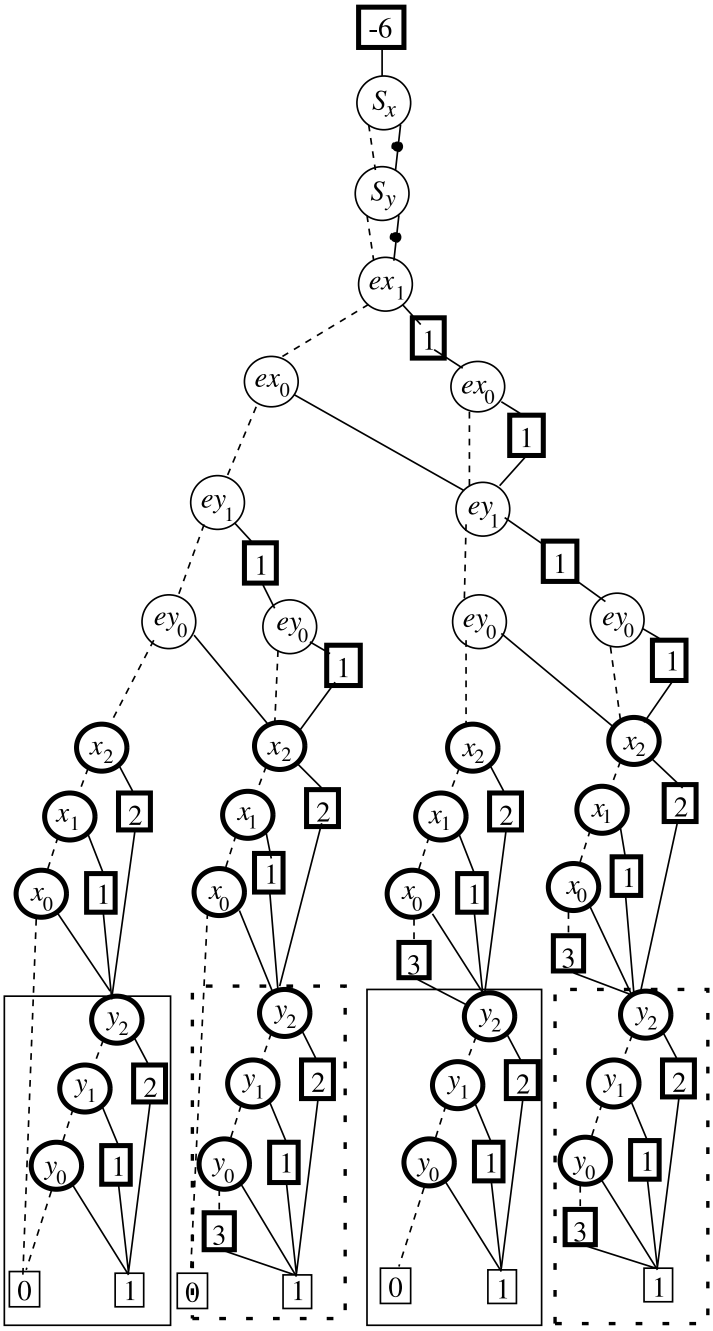


Figure 4.6: **Representations of floating-point encodings.**



$F_X * F_Y$

Figure 4.7: **Representation of floating-point multiplication.**

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

Figure 4.7 illustrates the *PHDD representation for floating-point multiplication. Observe that two negation edges reduce the graph size to one half of the original size. When $EX = 0$, the subgraph represents the function $0.X \times v_y.Y \times 2^{EY}$. When $EX \neq 0$, the subgraph represents the function $1.X \times v_y.Y \times 2^{EY}$. The size of exponent nodes grows linearly with the word size of the exponent part. The lower part of the resulting graph shows four mantissa products(from left to right): $X \times Y$, $X \times (2^3 + Y)$, $(2^3 + X) \times Y$, $(2^3 + X) \times (2^3 + Y)$. The first and third mantissa products share the common sub-function Y shown by the solid rectangles in Figure 4.7. The second and fourth products share the common sub-function $2^3 + Y$ shown by the dashed rectangles in Figure 4.7. In [27], 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 4.7, where n and m are the number of bits in the exponent and mantissa parts.

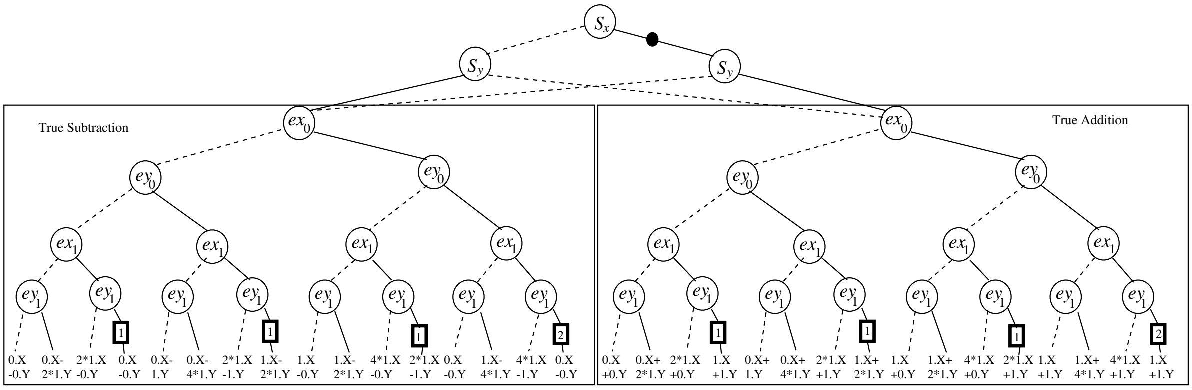


Figure 4.8: **Representation of floating-point addition.** For simplicity, the graph only shows sign bits, exponent bits and the possible combinations of mantissa sums.

For floating-point addition, the size of the resulting graph grows exponentially with the size of the exponent part. In Appendix 8, 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 4.8 illustrates the *PHDD representation of floating-point addition with two exponent bits for each floating-point operand. Observe that the negation edge reduces the graph size by half. 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 [27] 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.

4.4 Experimental Results

We have implemented *PHDD with basic BDD functions and applied it to verify arithmetic circuits. The circuit structure for four different types of multipliers are manually encoded in a C program which calls the BDD operations as well as *BMD or *PHDD operations. Our measurements are obtained on Sun Sparc 10 with 256 MB memory.

4.4.1 Integer Multipliers

Table 4.1 shows the performance comparison between *BMD and *PHDD for different integer multipliers with different word sizes. 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, while *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 require 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.

4.4.2 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

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 4.1: **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.

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

program. Table 4.2 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 exponent adder are negligible. The memory requirement is also similar to that of the integer multiplier.

4.4.3 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 4.3: **Performance for floating-point additions.** Results are shown for three different exponent word size with fixed mantissa size 23 and 52 bits.

Table 4.3 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.

4.5 Related Work

The major difference between *PHDD and the other three diagrams is in their ability to represent functions that map Boolean variables into floating-point values and their use of negation edges. Table 4.4 summarizes the differences between them.

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

Table 4.4: **Differences among four different diagrams.**

Compared to *BMDs [21], *PHDDs have three different decomposition types and a different method to represent and extract edge weights. These features enable *PHDDs to represent floating-point functions effectively. *BMD's edge weights are extracted as the greatest common divisor (GCD) of two children. In order to verify the multiplier with a size larger than 32 bits, *BMDs have to use multiple precision representation for integers to avoid the machine's 32-bit limit. This multiple precision representation and the GCD computation are expensive for *BMDs in terms of CPU time. Our powers of 2 method not only allows us to represent the floating-point functions but also improves the performance compared with *BMD's GCD method.

Compared with HDDs having six decompositions [30], *PHDDs have only three of them. In our experience, these three decompositions are sufficient to represent floating-point functions and verify floating-point arithmetic circuits. The other three decomposition types in HDDs may be useful for other application domains. Another difference is that *PHDDs have negation edges and edge weights, but HDDs do not. These features not only allow us to represent the floating-point functions but also reduce the graph size.

*PHDDs have only multiplicative edge weights, while K*BMDs [41] allow additive and multiplicative weights at the same time. The method of extracting the multiplicative weights is also different in these two representations. *PHDDs extract the powers-of-2 and choose the minimum of two children, but K*BMDs extract the greatest common divisor of two children like *BMDs. The additive weight in K*BMDs can be distributed down to the leaf nodes in *PHDD by recursively distributing to one or two branches depending on the decomposition type of the node. In our experience, additive weights do not significantly improve the sharing in the circuits we verified. The sharing of the additive weight may occur in other application domains.

Chapter 5

Extensions to Word-Level SMV

In Chapter 3, we presented a *BMD-based hierarchical verification approach for verification of integer circuits. *PHDDs were presented in Chapter 4 to provide a compact representation for integer and floating-point functions. Can we use *PHDDs in a hierarchical verification approach, similar to the one described in Chapter 3, to verify floating-point circuits such as adders? In this chapter, we will first discuss why *PHDD-based hierarchical verification is not suitable for floating-point circuits. We must either verify them in a flattened manner using pure *PHDDs or in a hierarchical manner using *PHDDs and a theorem prover. We prefer the former approach, because it can be fully automatic. Word-level SMV, introduced by Clarke *et al* [33], verifies integer circuits in a flattened manner and uses HDDs to represent integer functions. Since *PHDDs improve on HDDs, we integrate *PHDDs into word-level SMV for verification of floating-point circuits and develop several methodologies and additional *PHDD algorithms to enable the verification of floating-point circuits in a flattened manner.

The remainder of this chapter is organized as follows. Section 5.1 illustrates the drawbacks of *BMD-based hierarchical verification. Section 5.2 discusses word-level SMV with *PHDDs and two additional techniques for verification of floating-point circuits. Two additional *PHDDs algorithms are presented in Section 5.3.

5.1 Drawbacks of *BMD-Based Hierarchical Verification

In Chapter 3, we described our hierarchical verification method based on *BMDs to verify integer circuits such as multipliers and dividers. One of the main drawbacks of this approach is that the design must have a hierarchical form. From our experience, industrial designs usually do not have the module boundaries needed by our hierarchical verification approach. Since our approach has to partition the circuits into hierarchical modules, two circuit designs with the same functionality (e.g., integer multipliers based on adder-step and booth-encoding) can yield to two different hierarchical forms. Thus, the verification method for one circuit design cannot be reused directly on another design.

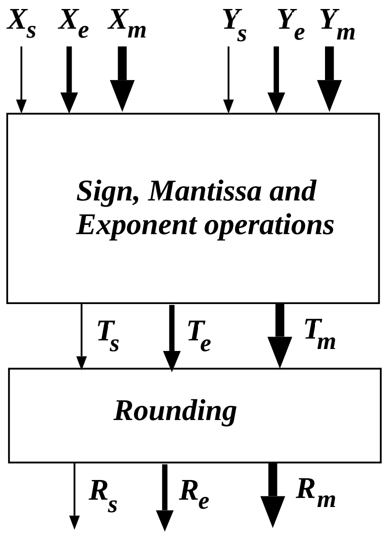


Figure 5.1: **Block Diagrams of floating-point circuits**

Another drawback is that our hierarchical verification approach cannot be easily extended to verify floating-point circuits, even using *PHDDs. The problem is caused by the rounding module in the floating-point circuits. Usually, floating-point circuits can be partitioned into two parts: the rounding module and the circuits before the rounding module. Figure 5.1 shows this partition of floating-point circuits, where X_s, X_e, X_m and Y_s, Y_e, Y_m are the sign, exponent and mantissa bits of inputs X and Y, respectively. T_s, T_e, T_m and R_s, R_m, R_m are the sign, exponent and mantissa bits of the intermediate result T, generated from the circuit before rounding, and the final result R after rounding, respectively. Usually, the number of bits in T_m is larger than X_m. R_m has the same size as X_m. The lower bits of T_m are used in rounding module to perform rounding operation. The decision diagrams explode in size during the composition of specifications in the rounding module, when the output of the rounding module is computed from the word-level functions obtained from the previous module of the circuit.

We use a floating-point multiplier with the round-to-nearest mode as an example to illustrate this problem in more detail. Assume the size of T_m is $2m$, where m is the size of X_m. Figure 5.2 shows the bit vectors T_m and R_m for the mantissa, where T_m is composed of $(t_{2m-1}, t_{2m-2}, ..., t_1, t_0)$ and R_m is composed of $(r_{m-1}, r_{m-2}, ..., r_1, r_0)$. When t_{2m-1}=0, the

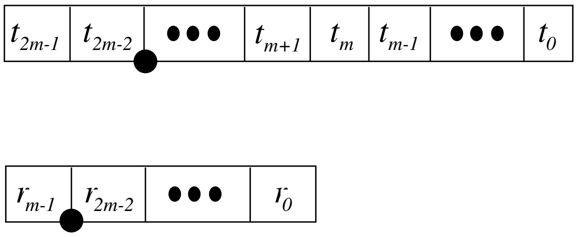


Figure 5.2: **Bit vectors of T_m and R_m**

rounding circuits use bit t_{m-1} (denoted L_0) and vector T_{l0}=($t_{m-2}, ..., t_0$) to decide whether 1 should be added into vector T_{h0} = ($t_{2m-2}, ..., t_m, t_{m-1}$). When t_{2m-1}=1, the rounding circuits use bit t_m (denoted L_1) and vector T_{l1}=($t_{m-1}, ..., t_0$) to decide whether 1 should be added into vector T_{h1} = ($t_{2m-1}, ..., t_{m+1}, t_m$). T_m, R_m, T_{h0}, T_{l0}, T_{h1} and T_{l1} are encoded as unsigned integers. The specification for the mantissa output R_m can be written as Equation 5.1:

$$R_m \;=\; \begin{cases} T_{h0} & t_{2m-1} = 0 \;\&\; !round_0 \\ T_{h0}+1 & t_{2m-1} = 0 \;\&\; round_0 \\ T_{h1} & t_{2m-1} = 1 \;\&\; !round_1 \\ T_{h1}+1 & t_{2m-1} = 1 \;\&\; round_1 \end{cases} \tag{5.1}$$

where ! represent Boolean complement, $round_0$ is $(T_{l0} > 2^{m-2}) \vee (L_0 == 0 \wedge T_{l0} == 2^{m-2})$ and $round_1$ is $(T_{h0} > 2^{m-1}) \vee (L_1 == 0 \wedge T_{l1} == 2^{m-1})$.

Since T_m is represented as a word-level function generated from the module before rounding in our hierarchical approach, $L_0, L_1, T_{h0}, T_{l0}, T_{h1}$ and T_{l1} must be computed from T_m. For example, T_{h1} is obtained from $T_m/2^m$ and L_1 is obtained from $T_{h1}\%2$. All these division and modular operations many grow exponentially. The BDDs for L_1=0 and L_0=0 grows exponentially with the value of m, because these are the middle bits of mantissa multiplier [14]. In our experience, we could not generate them when $m \geq 16$.

Because of these drawbacks, we decided to verify floating-point circuits in a flattened manner. Word-level SMV [33] is designed to verify circuits in the flattened manner. Thus, we improved word-level SMV by integrating *PHDDs and incorporating several techniques described in the rest of this chapter. The main advantage of this approach is that we can provide reusable specifications for the floating-point circuits such as adders and converters and we can make the verification process automatic.

5.2 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 [75], 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 [33] 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 [33] for the details of word-level SMV.

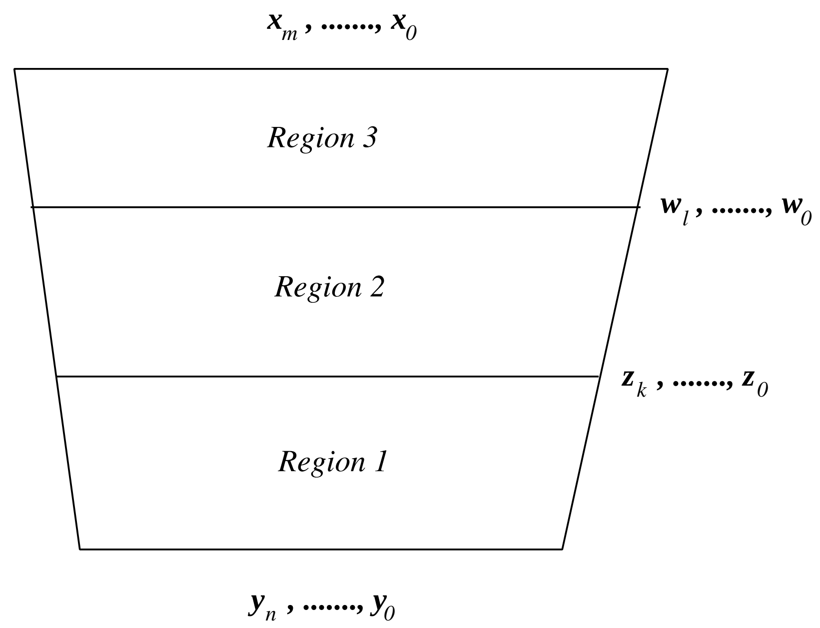


Figure 5.3: **Horizontal division of a combinational circuit,**

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 [99] describes the layering backward substitution, a variant of Hamaguchi's backward substitution approach [52], 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 symbolically evaluating bit level circuits. Figure 5.3 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 drawback of the backward substitution 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 a function more general than the original circuit of region 1. This generalization may cause the *PHDD for F to blowup.

5.2.1 Conditional Symbolic Simulation

To partially solve this problem, we introduced conditional symbolic simulation into word-level SMV. Symbolic simulation [19] 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 [63], 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. However, this approach is difficult to integrate into word-level SMV. Thus, we propose another approach to integrate conditional symbolic simulation into word-level SMV. Our approach is to apply the conditions directly during the symbolic simulation process. Right after building the BDDs for the output of each logic gate, the conditions are used to simplify the BDDs using the *restrict* [39] algorithm. Then, the simplified BDDs are 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.

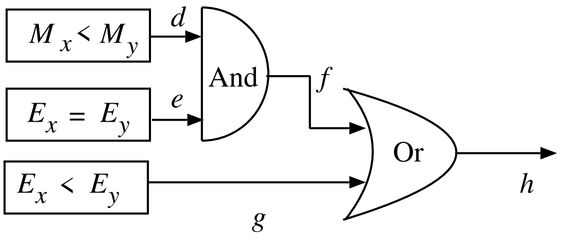


Figure 5.4: **The compare unit in floating-point adders.**

We use the following example to illustrate our conditional symbolic simulation process. Figure 5.4 shows the circuit for the compare unit in floating-point adders. Assume that k is the condition for a simulation run represented by a BDD, and the BDDs for signals d, e, and g are evaluated under our conditional symbolic simulation. In the simulation process, the BDD for signal f is evaluated by applying the And operations to the BDDs for signals d and e. Then, this BDD of signal f is simplified by the $restrict$ operation with the condition k. After that, the simplified BDD of signal f is used as one of the input to the Or gate. With proper conditions, this conditional symbolic simulation can reduce the BDDs of some internal signals to constant 0 or 1. For example, when the condition k is $E_x = E_y - 10$, signals e and g become 0 and 1, respectively. On the other hand, conditional symbolic simulation sometimes cannot reduce the BDDs of some internal signals at all. For example, condition k can not take any effect to reduce the BDDs of signal d, because the function of d is independent of the condition k.

5.2.2 Short-Circuiting Technique

Using conditional symbolic simulation, it is possible that the BDDs for some internal signals can be very large or cannot be build, but the BDDs for the final outputs are very small. For example, the BDDs of signal d in Figure 5.4 can be very difficult to build under the condition $E_x = E_y - 10$. If we try to build the BDDs for signal d and then signal e, then we cannot finish the job. However, if we build the BDD for signal e first, which will be 0, then we can directly return 0 for signal f without building the BDD for signal d.

Based on this observation, 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 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 *Xor* and *Xnor*, 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 5.4 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.

5.3 Additional *PHDD algorithms

5.3.1 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 [30]. 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 a branch-and-bound approach. The complexity of subtracting two HDDs is O($|F| \times |G|$). This algorithm can be shown to work 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 grow 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. In fact, Bryant and Chen have shown that $H = 0$ is NP-Hard for BMDs.

To solve this problem, we introduce relational operations with conditions to compute $cond \Rightarrow (F \sim G)$, 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 5.5. This algorithm produces the BDDs satisfying $H = 0$ under the condition $cond$, and is similar to the algorithm in [30], 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. As a convention, 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. We will demonstrate the usage of this algorithm to reduce computation time dramatically in Section 6.2.2.

5.3.2 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 [28]. Thus, Clarke's algorithm is not suitable for these operators with two *PHDDs having disjunctive sets of supporting variables.

We have developed algorithms for these relational operators with two *PHDDs having disjunctive sets of supporting variables. Figure 5.6 shows the new algorithm for computing BDDs

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

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

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 [30], 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.

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 5.7. 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.

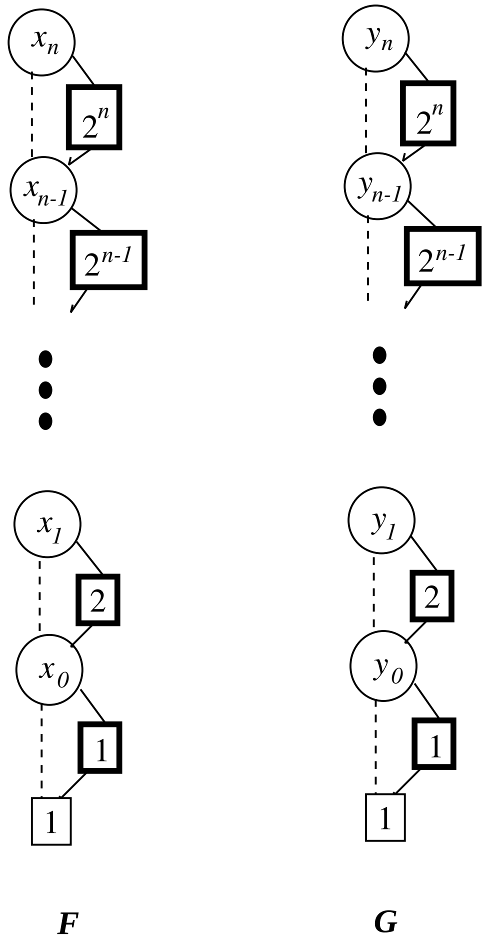


Figure 5.7: ***PHDDs for F and G.**

Chapter 6

Verification of Floating-Point Adders

In this chapter, we present the verification of a floating-point (FP) adder, obtained from the University of Michigan, as an example to validate our *PHDD representation and extended word-level SMV system. First, we briefly describe the floating-point adder design. Our methodology to verify floating-point adders is based on partitioning the input space to divide the specifications of floating-point adders into several hundred sub-specifications. Since we partition the input space, we introduce a BDD-based methodology to analyze the coverage of the input space by our specifications. Each sub-specifications can be verified within 5 minutes including counterexample generation if there is a design error in the input partition. Our system found five types of bugs in the design. For the corrected design, the verification of all specifications can be finished in 2 CPU hours on a Sun UltraSparc II machine.

The remainder of this chapter is organized as follows. Section 6.1 illustrates the design of the floating-point adder. The specifications of FP adders is presented in Section 6.2. Section 6.3 discusses the verification of the FP adder obtained from the University of Michigan including the design errors. Section 6.4 describes the verification of conversion circuits which convert the input from one format to another.

6.1 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 can not 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 $\pm\infty$.

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

Table 6.1: **Summary of the FP addition of two numbers of X and Y.** F represents the normal and denormal numbers. * indicates FP invalid arithmetic operands.

Table 6.1 summarizes the 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 6.1 shows the block diagram of the SNAP FP adder designed at Stanford University [83]. 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 shown in Figure 6.1, and using the final multiplexor to chose the correct output.

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

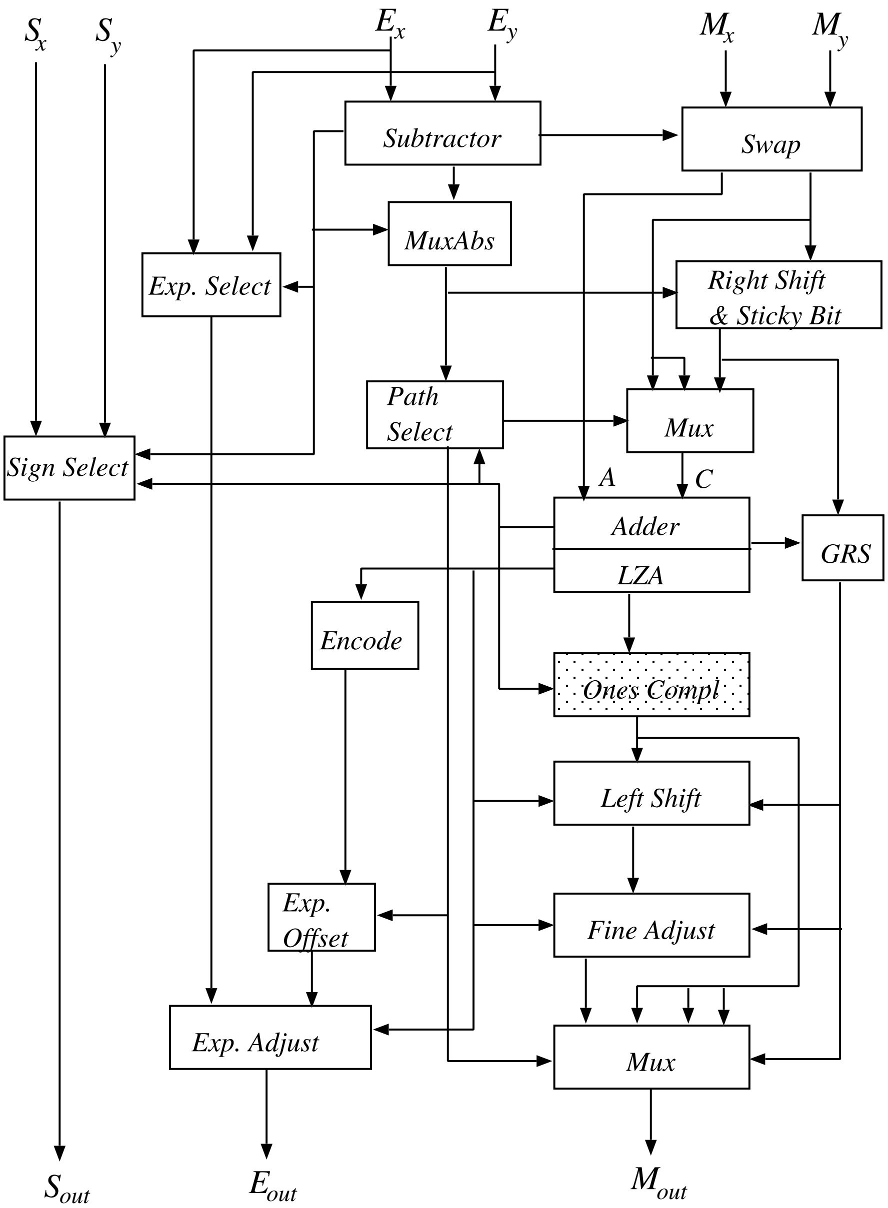


Figure 6.1: **The Stanford FP adder.**

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.

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 [95], are needed to swap the mantissas correctly. Figure 6.2 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 [95] and makes the verification harder described in Section 5.2.2. Figure 6.3 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).

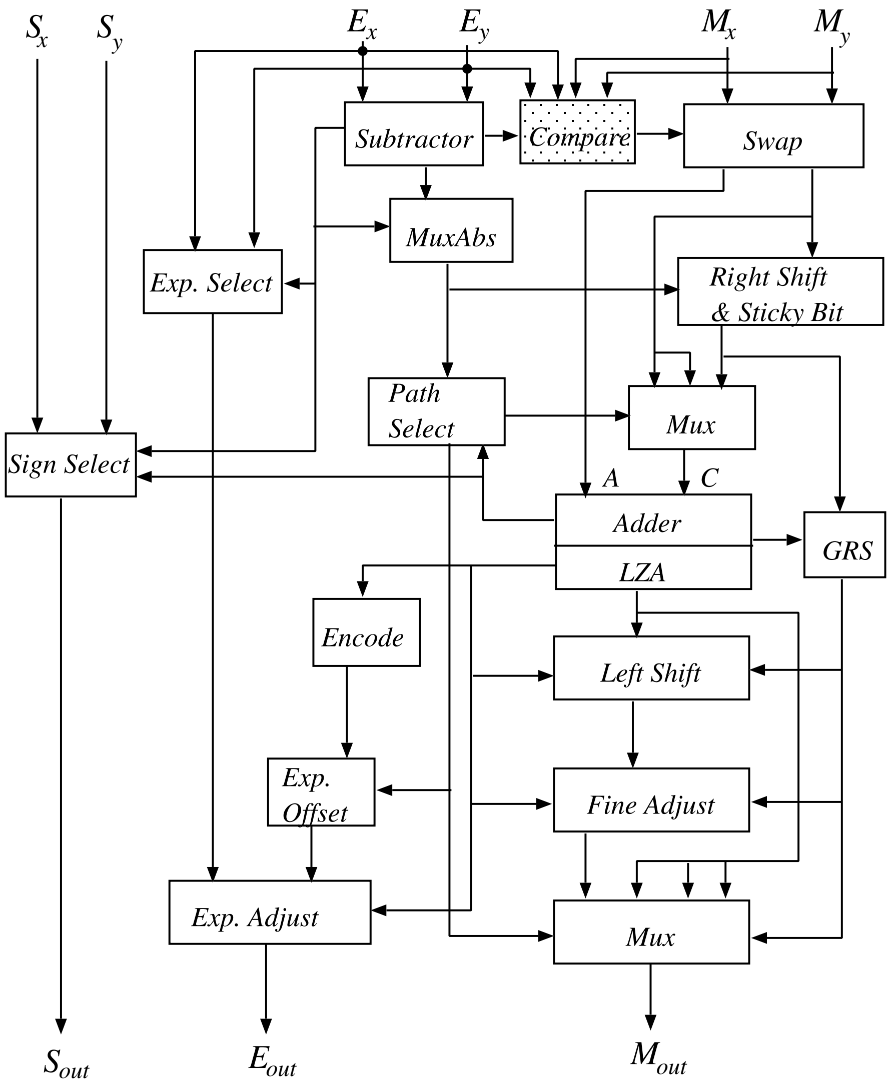


Figure 6.2: **A variant of Stanford FP adder.**

Otherwise, A is M_x.

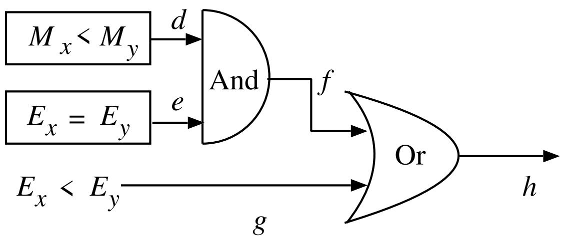


Figure 6.3: **Detailed circuit of the $compare$ unit.**

To verify this type of floating-point adders, the BDDs for the output bits cannot be built using conditional symbolic simulation, when the exponent difference is large. 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 6.3). 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 6.2 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. Thus, the BBDs for the outputs of floating-point adders cannot be built using conditional symbolic simulation.

Observe that signals e and g cannot be 1 simultaneously and signal d is only useful when e is 1. Thus, the BDDs of signal d must be built only when $E_x = E_y$. In this case, it has no problem building signal d, because the best ordering for both mantissa adder and $compare$ unit are the same. The short-circuiting technique described in Section 5.2.2 is used to overcome this ordering conflict problem, when $E_x \neq E_y$.

6.2 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

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

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 6.1.

$$X + Y = (-1)^{S_x} \times \begin{cases} (2^{E_x - B} \times M_x + M_y \times 2^{E_y - B}) & S_x = S_y \ (true\ addition) \\ (2^{E_x - B} \times M_x - M_y \times 2^{E_y - B}) & S_x \neq S_y \ (true\ subtraction) \end{cases} \quad (6.1)$$

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.

6.2.1 True Addition

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 6.2.

$$X + Y = (-1)^{S_x} \times \begin{cases} 2^{E_x - B} \times (M_x + (M_y >> i)) & E_y \leq E_x \\ 2^{E_y - B} \times (M_y + (M_x >> i)) & E_y > E_x \end{cases} \quad (6.2)$$

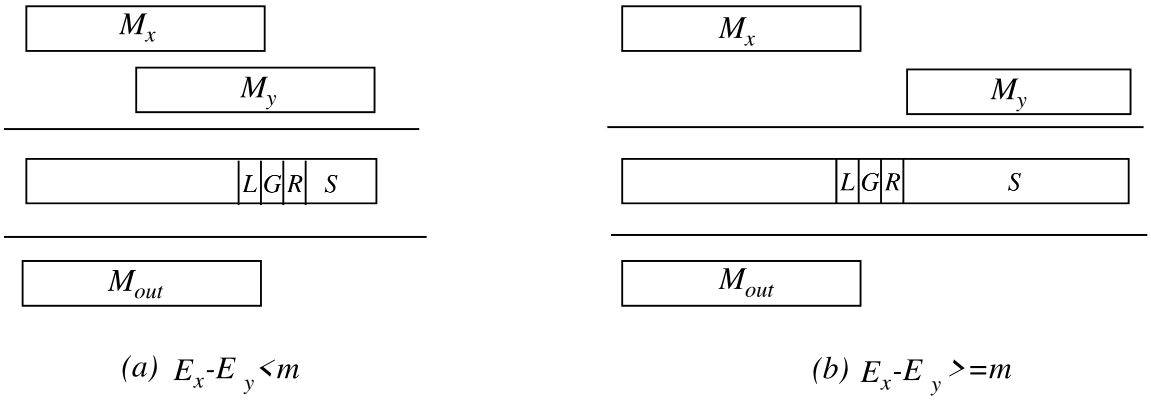


Figure 6.4: **Cases of true addition for the mantissa part.**

where $i = |E_x - E_y|$. When $E_y \leq E_x$, 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 6.4 illustrates the possible cases of true addition for $E_y \leq E_x$ based on the values of $E_x - E_y$. In Figure 6.4.a, for $0 \leq E_x - E_y < m$, 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 $E_x - E_y \geq m$ as shown in Figure 6.4.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:

$$\begin{cases} C_{a1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x + (M_y >> i))) & 0 \leq i < m \\ C_{a2} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x + (M_y >> m))) & i \geq m \\ C_{a3}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_y - B} \times (M_y + (M_x >> i))) & 0 < i < m \\ C_{a4} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_y - B} \times (M_y + (M_x >> m))) & i \geq m \end{cases} \quad (6.3)$$

where $C_{a1}[i]$, C_{a2}, $C_{a3}[i]$ and C_{a4} are $Cond\_add \& E_x = E_y + i$, $Cond\_add \&\ E_x > E_y + m$, $Cond\_add \& E_y = E_x + i$, and $Cond\_add \& E_y > E_x + m$, 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 $E_x - E_y \geq m$ are represented by $2^{E_x} = 2^{E_y + i}$ and $2^{E_x} \geq 2^{E_y + m}$. Both sets of variables must use Shannon decomposition to represent the FP function efficiently in [28]. 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. We observed that the best ordering for the specification represented by *PHDDs is the same ordering as the best ordering for the mantissa adder.

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.

6.2.2 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 true addition, the specifications for true subtraction can be written as Equation 6.4.

$$\begin{cases} C_{s1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> i))) & 2 \le i \le m \\ C_{s2} \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> m))) & i > m \\ C_{s3}[i] \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> i))) & 2 \le i \le m \\ C_{s4} \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> m))) & i > m \end{cases} \tag{6.4}$$

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 of 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 6.5.

$$\begin{cases} C_{c1}[i] \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - (M_y >> 1))) & 0 \le i < m \\ C_{c2}[i] \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - (M_x >> 1))) & 0 \le i < m \\ C_{c3}[i]) \Rightarrow OUT = Round((-1)^{S_x} \times 2^{E_x - B} \times (M_x - M_y)) & 1 \le i < m \\ C_{c4}[i]) \Rightarrow OUT = Round((-1)^{S_y} \times 2^{E_y - B} \times (M_y - M_x)) & 1 \le i < m \end{cases} \tag{6.5}$$

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: $(Cond\_sub\ \&E_x = E_y\ \&M_x = M_y) \Rightarrow OUT = 0$.

6.2.3 Specification Coverage

Since the specifications of floating-point adders are split into several hundred sub-specifications, do these sub-specifications cover the entire input space? To answer this question, one might use a theorem prover to check the case splitting. In contrast, we propose a BDD approach to compute the coverage of our specifications.

Our approach is based on the 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 $cond$s of our specifications, which can be computed 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.

6.3 Verification of FP Adders

In this section, we used the FP adder in the Aurora III Chip [56], 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 [83] at Stanford University. Dr. Huff found several errors with the approach described in [83]. 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.

6.3.1 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 6.5.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 removed. One approach is to directly connect the input of the latch to the output of the latch. This approach will eliminate some logic circuits related to the latch enable signals as shown on the right side of the latches in Figure 6.5.a. With this approach, the correctness of these circuits can not be checked. For example, an design error in the circuit, that always generated 0s for the enable signals of latches, can not be found, if we use this approach to remove the latches.

Our approach for latch removal is based on this observation: the data are written into the latches when the enable signals are 1. To ensure the correctness of the circuits for the enable signals, the latches can be replaced by And gates, as shown in Figure 6.5.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. With this approach, we can also check the correctness of circuits for the enable signals of the latches.

6.3.2 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 more than 10 million simulation runs performed by Dr. Huff in 4 days. Huff partitioned the simulation runs into three main operating regimes: alignment equal to 0, equal to 1, and greater than 1. For each regime, random floating-point numbers were fed to the design for simulation.

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

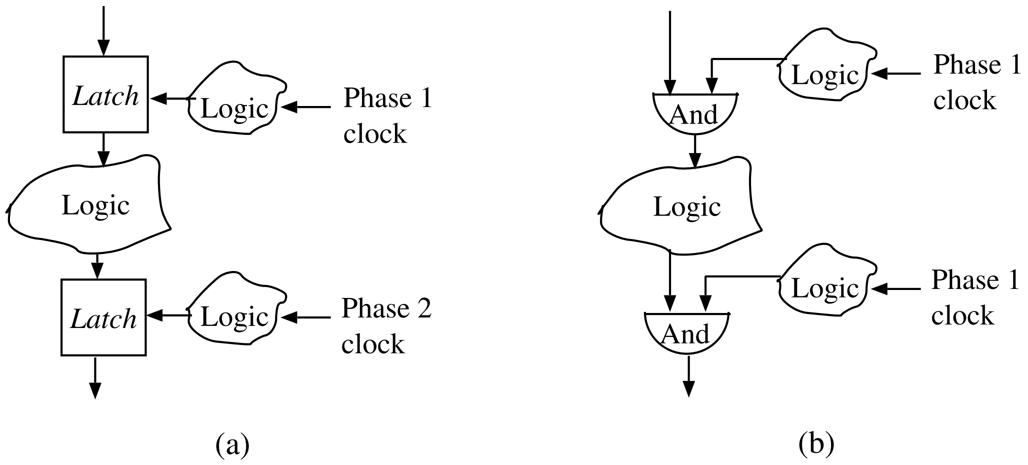


Figure 6.5: **Latch Removal.** (a) The pipelined, two phase design. (b) The design after latch removal.

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.

The second design error we found is in the sticky bit generation. The sticky bit generation is based on the table given in page 10 of Quach's paper describing the SNAP FP adder [83]. 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 6.6 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 [83].

The third design error occurs in the exponent path. The number of bits (6 bits) generated by the *Encode* unit, shown in Figure 6.1, is insufficient for the *exponent offset* unit, which may complement the vector and performs sign extension to 11 bits. 6 bits can represent values from 0 to 63. However, when the value is greater than 31 (i.e. 1xxxxx), the vector generated from the exponent offset unit looks like (111111xxxxx) after sign extension. This 11-bit vector is incorrect and the correct vector should be (000001xxxxx). Thus, a 7-bit vector must be generated by the *Encode* unit to achieve correct sign extension.

The fourth error is that Dr Huff missed one signal (denoted SHrev52 which indicates the

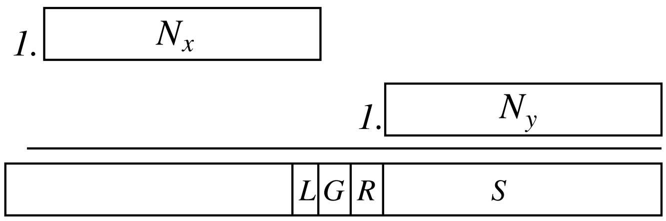


Figure 6.6: **Sticky bit generation, when $E_x - E_y$ = 54.**

intermediate result must be left shifted by 52 bits) in the OR gates to generate the 3rd least significant bit of the exponent offset in the *Encode* unit.

The fifth error occurs in the LZA unit which predicts the number of leading zeroes to be left shifted for normalization. When the exponent difference is 1 and A = 1.00...000 and C = 0.11...1111 (right shifted by 1 bit), the intermediate result is 0.00...0001, which must be left shifted by 53 bits. However, the LZA unit generated the incorrect value: 0 bits. I believe that this error is the most difficult one to be detected by simulation, since this is the only case in which intermediate result needs to be left shifted by 53 bits.

In summary, our system found bugs in the mantissa and exponent paths within several minutes. From our experience, the design errors in the mantissa path do not 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 6.3, 6.4 or 6.5.

6.3.3 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 6.1.b into Huff's FP adder. This new adder is equivalent to the FP adder in Figure 6.1.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 6.3 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

Case	CPU Time (seconds)		Max. Memory (MB)	
	FP adder I	FP adder II	FP adder I	FP adder II
True addition	3283	3329	49.07	54.91
True subtraction(*far*)	2654	2668	35.20	35.33
True subtraction (*close*)	994	1002	53.07	47.78

Table 6.3: **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 6.1.b. For true subtraction, *far* represent cases $|E_x - E_y| > 1$, and *close* represent cases $|E_x - E_y| \leq 1$.

simulation without the short-circuiting technique described in Section 5.2.2. 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.

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 of OUT for each specification 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 [29], 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.

6.4 Conversion Circuits

The overflow flag erratum of the FIST instruction (FP to integer conversion) [44] in Intel's Pentium Pro and Pentium II processors has illustrated the importance of verification of conversion circuits [56] which convert the data from one format to another. 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 FP adders, since these circuits are much simple than FP adders and only have one operand(i.e. less variables than FP adders). For example, the specification of the double-to-single operation, which converts the data from double precision to single precision, can be written as *"(overflow_flag = expected_overflow)* $\wedge$ *(not overflow_flag* $\Rightarrow$ *(output = expected_output))"*, where *overflow_flag* and *output* are directly from the circuit, and *expected_overflow* and *expected_output* are computed in terms of the inputs. This specification covers double precision which 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 operation 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).

Chapter 7

Conclusion

7.1 Summary

This thesis set out to provide techniques for verifying arithmetic circuits. In the previous chapters, we have described word-level decision diagrams and methodologies for the formal verification of these circuits. First, we introduced Multiplicative Binary Moment Diagrams (*BMDs) which provide a compact representation for integers functions. *BMD can serve as the basis for a hierarchical methodology for verifying integer circuits such as multipliers and dividers. Based on *BMDs and the hierarchical verification methodology, we have verified several integer circuits such as multipliers, dividers, and square roots.

Our *BMD-based approach cannot be directly applied to verify floating-point circuits. Two major challenges were that no existing word-level diagrams provide compact representations for floating-point functions and the decision diagrams explode during the composition of specifications in the rounding module. To overcome the first problem, we introduced Multiplicative Power Hybrid Decision Diagrams (*PHDDs) to represent integer and floating-point functions efficiently The performance comparison between *BMDs and *PHDDs for verification of integer circuits was discussed. To overcome the second problem, we changed our methodology to verify flattened design of floating-point circuits and described several improvements to word-level SMV to enable the verification of floating-point circuits such as adders.

We have illustrated the power of these techniques by verifying integer multipliers, dividers and square roots as well as floating-point adders. Our system found several design errors while verifying a floating-point adder obtained from the University of Michigan. We also demonstrated that our specifications are reusable for different implementations of floating-point adders. The important advantage of our system is that the verification process is fully automatic.

7.2 Future Work

7.2.1 Floating-Point Multipliers & Dividers

Floating-point adders, multipliers and dividers are the most common components of floating-point units. We have fully automated the verification of floating-point adders. However, we still have problems to automate the verification of floating-point multipliers and dividers. The main obstacle, which prevents us from verifying floating-point multipliers automatically, is that *PHDDs explode in size for the rounded result of floating-point multiplication and no good partitioning scheme of the input space was found to resolve this problem. Currently, floating-point multipliers must be divided into two sub-circuits: rounding module and the circuits before rounding. Each sub-circuit can be verified individually using word-level SMV with *PHDDs. However, the composition of the specifications of these sub-circuits can not be done using *PHDDs. Theorem provers are good candidate to handle this task.

Verification of floating-point dividers is even more challenging than verification of floating-point multipliers. Most of the circuit designs for floating-point dividers are iterative. The iterative design must be unrolled to verify the overall correctness of the result before rounding. Without unrolling, only the correctness of each iteration can be verified separately. Similar to floating-point multipliers, the rounding module must be verified separately.

It would be interesting to further investigate verification of these two types of floating-point circuits. One possible direction is to find a good partitioning scheme to partition the input space into several hundred or thousand partitions such that the verification task for each partition can be finished in a short time. Another possible direction is to develop another word-level diagram to provide a compact representation for the rounded result.

7.2.2 Arithmetic Circuits for MMX and DSP

In recent years, many processors have added MMX instructions to speed up the performance of the multimedia applications. For example, Intel introduced Pentium processors with MMX [50] in January, 1996 and Pentium II processors (i.e. Pentium Pro with MMX) in May, 1997 [51]. Other vendors such as Digital, HP, Sun and SGI added similar instructions into their processors [49]. Many of these MMX instructions perform arithmetic operations. Thus, the circuits for them are arithmetic circuits. Digital Signal Processors (DSP) also contain a lot of arithmetic circuits.

Most of arithmetic circuits for MMX or DSP are based on integer adders, multipliers and multiply-accumulate units. Recently, floating-point circuits are being used in MMX and DSP.

For example, Mpact2 introduced by Chromatic at the end of 1996 [98] contains floating-point adders and multipliers.

In addition to regular arithmetic, circuits for saturated operations are common in both MMX and DSP. A saturated addition operation on two 32-bit operands will yield the maximum value representable by 32 bits as the final result when the sum of two operands is greater the maximum value. We believe that our techniques can be directly applied to verify the circuits for saturated adders (both integer and floating-point). The verification of circuits for saturated floating-point multiplication has the same difficulty as normal floating-point multiplication. In addition, our techniques cannot successfully verify the saturated integer multiplication circuits, because the *PHDD (*BMD) explodes in size. To be specific, for two 32-bit inputs X and Y, the expected result will be $(X * Y \geq 2^{32})?2^{32} - 1 : X * Y$. We could not compute the BDD for $X * Y \geq 2^{32}$ and we believe that the *PHDDs for the expected result will grow exponentially with the word size. Further investigation of this problem is needed to verify this type of integer multiplier.

Chapter 8

*PHDD Complexity of Floating-Point Operations

In this Chapter, we prove the *PHDD complexity of floating-point multiplication and addition. The complexity of floating-point multiplication is shown to be linear in Section A. Section B shows that the complexity of floating-point addition grows linearly with the mantissa size, but grows exponentially with the exponent size.

A Floating-Point Multiplication

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 value of EX, Expanding and rearranging the terms of the multiplication $F_X \times F_Y$ yields:

$$\begin{aligned} F_X \times F_Y &= (-1)^{S_x \oplus S_y} \times (v_x.X \times 2^{EX-B}) \times (v_y.Y \times 2^{EY-B}) \\ &= (-1)^{S_x \oplus S_y} \times 2^{-2B} \times \begin{cases} 2^1 \times (0.X \times v_y.Y) \times 2^{EY} & Case\ 0 : EX = 0 \\ 2^{EX} \times (1.X \times (v_y.Y) \times 2^{EY} & Case\ 1 : EX \neq 0 \end{cases} \end{aligned} \tag{8.1}$$

The following theorem shows that the size of the resulting graph grows linearly with the word size for the floating-point multiplication.

Theorem 1 *The size of the resulting graph of floating-point multiplication is* $6(n+m)+3$*, where* n *and* m *are the number of bits in the exponent and mantissa parts.*

Proof: From Equation 8.1 and Figure 4.7, we know that there are no sharing in the sub-graphs for $EX = 0$ and $EX \neq 0$. For $EX = 0$, the size of the sub-graph except leaf nodes is the sum of the nodes for the exponent of F_Y ($2n - 1$ nodes), the nodes for the mantissa of F_X ($2m$ nodes), and the nodes for the mantissa of F_Y (m nodes). Similarly, for $EX \neq 0$, the size of the sub-graph except leaf nodes is also $2n + 3m - 1$. The size for the exponent part of F_X is $2n - 1$. The number of nodes for the sign bits and top level edge weight is 3, and the number of leaf nodes is 2. Therefore, the size of the resulting graph for floating-point multiplication is $6(n + m) + 3$. □

B Floating-Point Addition

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

For floating-point addition, the size of the resulting graph grows exponentially with the size of the exponent part. The following theorem proves that the number of distinct mantissa sums in *PHDD representation grows exponentially with the size of the exponent part.

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

Proof: We first show that the floating-point addition can be divided into two cases according to their sign bits. When $S_x \oplus S_y$ is equal to 0, the floating-point addition must be performed as "true addition" shown as Equation 6.1.

$$F_X + F_Y \;=\; (-1)^{S_x} \times (2^{EX-B} \times v_x.X + v_y.Y \times 2^{EY-B}) \tag{8.2}$$

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

$$F_X + F_Y \;=\; (-1)^{S_x} \times (2^{EX-B} \times v_x.X - v_y.Y \times 2^{EY-B}) \tag{8.3}$$

There is common distinct mantissa sum among true addition and true subtraction, since one performs addition and another performs subtraction.

Let us consider the true addition operation first. Based on the relation of EX and EY, Equation 8.2 can be rewritten as the following equation:

$$F_X + F_Y = (-1)^{S_x} \times \tag{8.4}$$

$$\begin{cases} (-1)^{S_x} \times 2^{EY-B} \times \{2^{EX-EY} \times 1.X + 1.Y\} & Case\ 0 : EX > 0 \,\& EY > 0 \,\& EX > EY \\ (-1)^{S_x} \times 2^{EX-B} \times \{1.X + 1.Y \times 2^{EY-EX}\} & Case\ 1 : EX > 0 \,\& EY > 0 \,\& EX < EY \\ (-1)^{S_x} \times 2^{EX-B} \times \{1.X + 1.Y\} & Case\ 2 : EX > 0 \,\& EY > 0 \,\& EX = EY \\ (-1)^{S_x} \times 2^{1-B} \times \{2^{EX-1} \times 1.X + 0.Y\} & Case\ 3 : EX > 0 \,\& EY = 0 \\ (-1)^{S_x} \times 2^{1-B} \times \{0.X + 1.Y \times 2^{EY-1}\} & Case\ 4 : EX = 0 \,\& EY > 0 \\ (-1)^{S_x} \times 2^{1-B} \times \{0.X + 0.Y\} & Case\ 5 : EX = EY = 0 \end{cases} \tag{8.5}$$

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

Similarly, Equation 8.3 can be rewritten as the following equation:

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

$$\begin{cases} (-1)^{S_x} \times 2^{EY-B} \times \{2^{EX-EY} \times 1.X - 1.Y\} & Case\ 0 : EX > 0 \,\& EY > 0 \,\& EX > EY \\ (-1)^{S_x} \times 2^{EX-B} \times \{1.X - 1.Y \times 2^{EY-EX}\} & Case\ 1 : EX > 0 \,\& EY > 0 \,\& EX < EY \\ (-1)^{S_x} \times 2^{EX-B} \times \{1.X - 1.Y\} & Case\ 2 : EX > 0 \,\& EY > 0 \,\& EX = EY \\ (-1)^{S_x} \times 2^{1-B} \times \{2^{EX-1} \times 1.X - 0.Y\} & Case\ 3 : EX > 0 \,\& EY = 0 \\ (-1)^{S_x} \times 2^{1-B} \times \{0.X - 1.Y \times 2^{EY-1}\} & Case\ 4 : EX = 0 \,\& EY > 0 \\ (-1)^{S_x} \times 2^{1-B} \times \{0.X - 0.Y\} & Case\ 5 : EX = EY = 0 \end{cases} \tag{8.6}$$

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

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

Proof: Theorem 2 showed that the number of distinct mantissa sums is $2^{n+3} - 10$. Except the leaf nodes, each mantissa sum can be represented by $2m$ nodes, but there is some sharing among the mantissa graphs. First, let us look at the sharing among the mantissa sums of true addition. For case 4 in Equation 8.5, the graphs to represent function $0.X + 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$, which is also in the graph representing function $1.X + 0.Y$. Thus, there are $2^n - 1$ distinct mantissa sums to share the same graph($1.Y$). Again, the graphs to represent $1.X + 1.Y$ in case 2 and $2 \times 1.X + 0.Y$ in case 3, share the sub-graph $2 + 0.Y$, since $1.X + 1.Y = 0.X + (2 + 0.Y)$ and $2 \times 1.X + 0.Y = 2 \times 0.X + (2 + 0.Y)$. Therefore, we have to subtract $(2^n - 1)m$ nodes from the total nodes.

Then, let us look at the true subtraction. First, the graph to represent $0.X - 0.Y$ shares the sub-graph $0.Y$ with $0.X + 0.Y$ in true addition, because of the negation edge. For Case 4 in Equation 8.6, the graphs to represent function $0.X - 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$ in true addition. The graphs to represent $1.X - 0.Y$ in case 3 and $2 \times 1.X - 1.Y$ in case 0, share the sub-graph $1 - 0.Y$, since $1.X + 1.Y = 0.X + (1 - 0.Y)$ and $2 \times 1.X - 1.Y = 2 \times 0.X + (1 - 0.Y)$. Therefore, we have to subtract $(2^n + 1)m$ nodes from the total nodes. Thus, the number of non-leaf nodes to represent these distinct mantissa sums is $(7 \times 2^n - 10) \times (2m)$.

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

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

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

Base Case: If $n = 2$, the number of exponent nodes for true addition is $5 \times 2^{2+1} - 8 \times 2 - 9 = 15$ as shown in Figure 4.8.

Induction Step: Assume the claim holds for $n = k$. To prove that the claim holds for $n = k + 1$, let EX_k and EY_k represent the low k bits of EX and EY. Thus, EX is represented as $2^k \times ex_k + EX_k$. Based on the values of EX_k and EY_k, Equation 6.1 can be rewritten as the following:

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

$$\begin{cases} 2^{1-B} \times \{2^{((2^k-1)\times ex_k)} \times G + 2^{((2^k-1)\times ey_k)} \times H\} & Case\ 0: EX_k = EY_k = 0 \\ 2^{1-B} \times \{2^{(EX_k-1+(2^k-1)\times ex_k)} \times 1.X + H \times 2^{((2^k-1)\times ey_k)}\} & Case\ 1: EX_k > 0\ \& EY_k = 0 \\ 2^{1-B} \times \{G \times 2^{((2^k-1)\times ex_k)} + 1.Y \times 2^{(EY_k-1+(2^k-1)\times ey_k)}\} & Case\ 2: EX_k = 0\ \& EY_k > 0 \\ 2^{EY_k-B} \times \{(2^{EX_k-EY_k+2^k\times ex_k)} \times 1.X + 1.Y \times 2^{(2^k\times ey_k)}\} & Case\ 3: EX_k > 0\ \& EY_k > 0\ \& EX_k > EY_k \\ 2^{EX_k-B} \times \{2^{(2^k\times ex_k)} \times 1.X \times +1.Y \times 2^{(EY_k-EX_k+2^k\times ey_k)}\} & Case\ 4: EX_k > 0\ \& EY_k > 0\ \& EX_k \le EY_k \end{cases} \tag{8.7}$$

where G (H) is $0.X$ $(0.Y)$ if ex_k (ey_k) is 0; otherwise, G (H) is $1.X$ $(1.Y)$. Figure 8.1.a illustrates the distinct sub-graphs after expanding variable ey_{k-1}. These sub-graphs are divided into five types, according to the cases in Equation 8.7. For Case 0, there is only one distinct sub-graph. For Case 1, there are $2^k - 1$ distinct sub-graphs, since the number of possible value of EX_k is $2^k - 1$ and each value of EX_k will generate a unique function. Similarly, there are $2^k - 1$, $2^k - 2$, and $2^k - 1$ distinct sub-graphs for Case 2, 3, and 4, respectively. Thus, the total number of distinct sub-graphs is $2^{k+2} - 4$.

Figures 8.1.b shows the sub-graph for Case 0. In the graphs, each tuple (i, P, j, Q) represents $2^i \times P + 2^j \times Q$. For example, tuple $(0, 0.X, 0, 0.Y)$ represents $2^0 \times 0.X + 2^0 \times 0.Y$. Figures 8.1.c to Figures 8.1.f show the graphs with a parameter i for Cases 1, 2, 3 and 4, which serve as the template of the graphs in the cases. For instance, the graph in Case 1 with $i = 1$ represents the function $2^{(EX_k-1+(2^k-1)\times ex_k)} \times 1.X + H \times 2^{((2^k-1)\times ey_k)}$ with $EX_k = 1$ in Case 2 of Equation 8.7.

Since each sub-graph is distinct, the nodes with variable ex_k are unique (i.e. no sharing). Observing from these five types of sub-graphs, the possible sharing among the nodes with variable ey_k is these cases: the ey_k nodes in case 2 share with that in cases 3 and 4, and the nodes in case 3 share with that in case 4. For the first case, the possible sharing is the right ey_k nodes in Figure 8.1.e and Figure 8.1.g. Observe that these two ey_k node will be that same in the graph with $i = j$ in case 2 and the graph with $i = j + 1$ in case 4. Since the possible values of i are ranged from 0 to $2^k - 3$, there are $2^k - 2$ ey_k nodes shared. When $i = 2^k - 2$, the right ey_k node in the graph of case 2 will be shared with the left ey_k node in the graph with i=1 in Figure 8.1.e. Therefore, all of the right ey_k nodes in Case 2 are shared nodes and are $2^k - 1$ nodes. For the second case, the possible sharing is the left ey_k node in Figure 8.1.e and the right ey_k node in Figure 8.1.f. Observe that when $i_1 + i_2 = 2^k$, the left ey_k node in the graph with $i = i_1$ in case 3 is the same as the right ey_k node in the graph with $i = i_2$ in case 4. Since $2 \le i_1 \le 2^k - 2$ and $0 \le i_2 \le 2^k - 2$, there are $2^k - 3$ nodes shared. Therefore, the total number of exponent nodes are $5 \times 2^{k+1} - 8 \times k - 9 + 3 \times (2^{k+2} - 4) - (2^k - 1) - (2^k - 3) = 5 \times 2^{(k+1)+1} - 8 \times (k + 1) - 9 = 5 \times 2^{n+1} - 8 \times n - 9$.

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

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

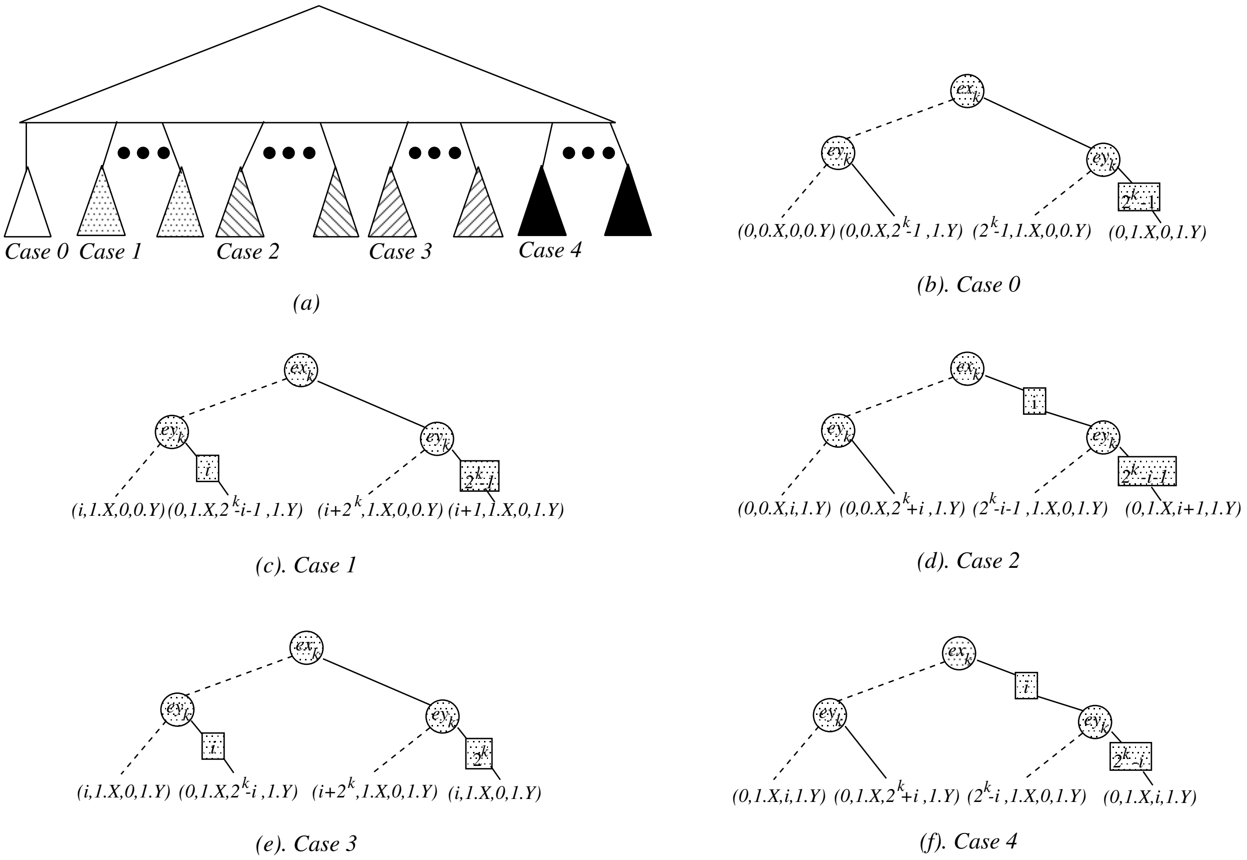


Figure 8.1: **Distinct sub-graphs after variable ey_{k-1}.** (a) Distinct sub-graphs after variable ey_{k-1} are divided into 5 types shown in graphs (b) to (f) which serve as template with a parameter i. (b) Case 0 only has one distinct graph. (c) $0 \leq i = EX_k - 1 \leq 2^k - 2$. (d) $0 \leq i = EY_k - 1 \leq 2^k - 2$. (e) $1 \leq i = EX_k - EY_k \leq 2^k - 2$. (f) $1 \leq i = EY_k - EX_k \leq 2^k - 2$.

Proof: The size of the resulting graph is the sum of the nodes for the sign, exponent and mantissa parts. The nodes for the sign part are 3 as shown in Figure 4.8. Lemma 1 and 2 have shown the sizes of the mantissa and exponent parts respectively. Therefore, their sum is $2^{n+1} \times (7m + 9) - 20m - 16n - 19$. $\square$

Bibliography

[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] AKERS, S. B. Binary decision diagrams. In *IEEE Transactions on Computers* (June 1978), pp. 6:509–516.

[3] ARDITI, L. *BMDs can delay the use of theorem proving for verifying arithmetic assembly instructions. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 34–48.

[4] BAHAR, R. I., FROHM, E. A., GAONA, C. M., HACHTEL, G. D., MACII, E., PARDO, A., AND .SOMENZI, F. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer-Aided Design* (November 1993), pp. 188–191.

[5] BARTELINK, D. Processes of the future. In *Solid State Technology* (Feburary 1995), pp. 42–53.

[6] BEATTY, D. L., AND BRYANT, R. E. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st ACM/IEEE Design Automation Conference* (June 1994).

[7] BOSE, S., AND FISHER, A. L. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of 1989 IEEE Internaational Conference on Computer Design: VLSI in Computer and Processors* (October 1989), pp. 217–221.

[8] BOYER, R. S., AND MOORE, J. S. *A computational Logic Handbook*. Academic Press, 1988.

[9] 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.

[10] BRAYTON, R. K., HACHTEL, G. D., SANGIOVANNI-VINCENTELLI, A., SOMENZI, F., AZIZ, A., CHENG, S.-T., EDWARDS, S., KHATRI, S., ANS ABELARDO PARDO, Y. K., QADEER, S., RANJAN, R. K., SARWARY, S., SHIPLE, T. R., SWAMY, G., AND VILLA, T. VIS: A system for verification and synthesis. 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. 428–436.

[11] BROCK, B., KAUFMANN, M., AND MOORE, J. S. ACL2 theorems about commerical microprocessors. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 275–293.

[12] BROWN, F. M. Reduced solutions of Boolean equations. In *IEEE Transactions on Computers* (October 1970), pp. 10:1230–1245.

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

[14] BRYANT, R. E. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. In *IEEE Transactions on Computers* (Feb 1991), pp. 2:205–213.

[15] BRYANT, R. E. Symbolic boolean manipulation with ordered binary decision diagrams. In *ACM Computing Surveys* (September 1992), pp. 3:293–318.

[16] BRYANT, R. E. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 236–243.

[17] BRYANT, R. E. Bit-level analysis of an SRT divider circuit. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996).

[18] BRYANT, R. E., BEATTY, D. L., BRACE, K., AND CHO, K. COSMOS: a compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (June 1987), pp. 9–16.

[19] 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.

[20] 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.

[21] 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.

[22] BURCH, J. R. Using BDDs to verify multipliers. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), pp. 408–412.

[23] BURCH, J. R., CLARKE, E. M., AND LONG, D. E. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), pp. 403–407.

[24] CABODI, G., CAMURATI, P., CORNO, F., PRINETTO, P., AND REORDA, M. S. Sequential circuit diagnosis based on formal verification techniques. In *Proceedings of the International Test Conference* (1992).

[25] CAMILLERI, A. J. Simulation as an aid to verifiaction using the HOL theorem prover. In *Proceedings of IFIP TC10 Working Conference: Design Methodologies for VLSI and Computer Architecture* (September 1988), pp. 148–168.

[26] 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).

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] CLARKE, E. M., GERMAN, S. M., AND ZHAO, X. Verifying the SRT division using theorem proving techniques. 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. 111–122.

[32] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. Model checking and abstraction. In *ACM Symposium on Principles of programming Languages* (1992).

[33] 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.

[34] 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.

[35] CLARKE, E. M., AND ZHAO, X. Analytica: A theorem prover for Mathematica. In *The Journal of Mathematica* (1993).

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

[37] COUDERT, O., BERTHET, C., AND MADRE, J. C. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for finite state systems* (June 1989), pp. 365–373.

[38] COUDERT, O., BERTHET, C., AND MADRE, J. C. Verification of sequential machines using Boolean function vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI designs* (1990), pp. 111–128.

[39] 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.

[40] DILL, D. L. The Murphi Verification System. 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. 390–393.

[41] 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.

[42] 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.

[43] ENDERS, R. Note on the complexity of binary moment diagram representations. In *IFIP WG 10.5 Workshop on Applications of Reed-Muller Expansion in circuit Design* (1995).

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

[45] GEIST, D., AND BEER, I. Efficient model checking by automated ordering of transition relation partitions. In *Computer-Aided Verification, CAV '94* (Stanford CA, USA, July 1994), R. Alur and T. A. Henzinger, Eds., no. 818 in Lecture Notes in Computer Science, Springer-Verlag, pp. 299–310.

[46] GORDON, M. J. C. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis* (1987), Birtwistle and Subramanyam, Eds., Kluwer Academic Publishers.

[47] GWENNAP, L. Sun tweaks SuperSparc to boost performance. In *Microprocessor Report* (November 1994).

[48] GWENNAP, L. Pentium Pro debuts with few bugs. In *Microprocessor Report* (December 1995).

[49] GWENNAP, L. Digital, mips add multimedia extensions. In *Microprocessor Report* (November 1996), pp. 24–28.

[50] GWENNAP, L. Intel's MMX speeds multimedia. In *Microprocessor Report* (March 1996), pp. 1–6.

[51] GWENNAP, L. Pentium II debuts at 300MHz. In *Microprocessor Report* (May 1997), pp. 1–8.

[52] 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.

[53] HARDIN, R. H., HAR'EL, Z., AND KURSHAN, R. P. COSPAN. 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. 423–427.

[54] HOLZMANN, G. J., AND PELED, D. Partial order reductions with on-the-fly model checking. In *Computer-Aided Verification, CAV '94* (Stanford CA, USA, July 1994), R. Alur and T. A. Henzinger, Eds., no. 818 in Lecture Notes in Computer Science, Springer-Verlag, pp. 377–390.

[55] HOLZMANN, G. J., AND PELED, D. The state of SPIN. 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. 385–389.

[56] 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).

[57] HUNT, W. A. FM8501: A verified microprocessor. In *Lecture Notes in Artifical Intelligence* (1994), Springer Verlag.

[58] I. MINATO, S. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 272–277.

[59] IP, C. N., AND DILL, D. L. Efficient verification of symmetric concurrent systems. In *Proceedings of 1993 IEEE Internaational Conference on Computer Design: VLSI in Computer and Processors* (October 1993), pp. 230–234.

[60] IP, C. N., AND DILL, D. L. State reduction using reversible rules. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996).

[61] JAIN, A., NELSON, K., AND BRYANT, R. E. Verifying nondeterministic implementations of deterministic systems. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 109–125.

[62] JAIN, J., BITNER, J., ABADIR, M. S., ABRAHAM, J. A., AND FUSSELL, D. S. Indexed BDDs: Algorithmic advances in techniques to represent and verify boolean functions. In *IEEE Transactions on Computers* (November 1997), pp. 11:1230–1245.

[63] 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.

[64] JOYCE, J. Formal verification and implementation of a microprocessor. In *VLSI Specification, Verification and Synthesis* (1987), Birtwistle and Subramanyam, Eds., Kluwer Academic Publishers, pp. 129–157.

[65] JOYCE, J. J., AND SEGER, C.-J. H. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 469–474.

[66] KAPUR, D., AND SUBRAMANIAM, M. Mechanically verifying a family of multiplier circuits. 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. 135–146.

[67] KAUFMANN, M., AND MOORE, J. S. ACL2: An industrial strength version of Nqthm. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS-96)* (June 1996), pp. 23–34.

[68] KEBSCHULL, U., SCHUBERT, E., AND ROSENTIEL, W. Multilevel logic based on functional decision diagrams. In *Proceedings of the European Design Automation Conference* (1992), pp. 43–47.

[69] KIMURA, S. Residue BDD and its application to the verification of arithmetic circuits. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (June 1995), pp. 542–548.

[70] KUEHLMANN, A., SRINIVASAN, A., AND LAPOTIN, D. P. Verity–a formal verification program for custom CMOS circuits. *IBM Journal of Research Development* (January 1995).

[71] KURSHAN, R. P., AND LAMPORT, L. Verification of a multiplier: 64 bits and beyond. In *Computer-Aided Verification, CAV '93* (Elounda,Greece, July/August 1993), C. Courcoubeties, Ed., no. 1697 in Lecture Notes in Computer Science, Springer-Verlag, pp. 166–179.

[72] LAI, Y.-T., PEDRAM, M., AND VRUDHULA, S. B. K. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (August 1994), pp. 959–975.

[73] LAI, Y.-T., AND SASTRY, S. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design Automation Conference* (June 1992), pp. 608–613.

[74] 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.

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

[76] MCSORLEY, O. L. High-speed arithmetic in binary computers. In *Proceedings of IRE* (1961), pp. 67–91.

[77] MELHAM, T. F. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis* (1987), Birtwistle and Subramanyam, Eds., Kluwer Academic Publishers, pp. 269–291.

[78] MINATO, S.-I. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1995.

[79] MINER, P. S., AND LEATHRUM, J. F. Verification of IEEE compliant subtractive division algorithms. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 64–78.

[80] OWRE, S., RUSHBY, J., SHANKAR, N., AND SRIVAS, M. K. A tutorial on using PVS for hardware verification. In *Proceedings of Second International Conference of Theorem provers in Circuit Design: Theory, Pracctice and Experience* (September 1994), pp. 258–279.

[81] PANDEY, M., RAIMI, R., BEATTY, D. L., AND BRYANT, R. E. Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996).

[82] PANDEY, M., RAIMI, R., BRYANT, R. E., AND ABADIR, M. S. Formal verification of content addressable memories using symolic trajectory evaluation. In *Proceedings of the 34th ACM/IEEE Design Automation Conference* (June 1997).

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

[84] RAJAN, S., SHANKAR, N., AND SRIVAS, M. K. Automatic datapath abstraction in hardware systems. In *Computer-Aided Verification, CAV '95* (Liege, Belgium, June 1995), P. Wolper, Ed., no. 939 in Lecture Notes in Computer Science, Springer-Verlag.

[85] RAJAN, S., SHANKAR, N., AND SRIVAS, M. K. An integeration of model-checking with automated proof checking. In *Computer-Aided Verification, CAV '95* (Liege, Belgium, June 1995), P. Wolper, Ed., no. 939 in Lecture Notes in Computer Science, Springer-Verlag, pp. 84–97.

[86] RAVI, K., PARDO, A., HACHTEL, G. D., AND SOMENZI, F. Modular verification of multipliers. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 49–63.

[87] ROBERTSON, J. E. A new class of digital division methods. In *IRE Transactions on Electronic Computers* (1958), pp. 218–222.

[88] ROTTER, D., HAMAGUCHI, K., ICHI MINATO, S., AND YAJIMA, S. Manipulation of large scale polynomial using BMDs. In *IEICE Transition on Fundamentals of Electronics, Communications and Computer Sciences* (October 1997), pp. 1774–1781.

[89] 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.

[90] SCHROER, O., AND WEGENER, I. The theory of zero-suppressed BDDs and the number of knights tours. Tech. Rep. 552/1994, University of Dortmund, Fachbereich Informatik, 1994.

[91] SEGER, C.-J. H. Voss– a formal hardware verification system: User's guide. Tech. Rep. 93-45, Dept. of Computer Science, University of British Columbia, 1993.

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

[93] SRIVAS, M. K., AND MILLER, S. P. Applying formal verification to a commercial microprocessor. In *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications* (Chiba, Japan, Aug. 1995), S. D. Johnson, Ed., Proceedings published in a single volume jointly with ASP-DAC '95, CHDL '95, and VLSI '95, IEEE Catalog no. 95TH8102, pp. 493–502.

[94] STANKOVIC, R. S., AND SASAO, T. Decision diagrams for discrete functions: Classification and unified interpretation. In *Proceedings of ASP-DAC '98* (Yokohoma,Japan, Feb. 1998), pp. 439–445.

[95] 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.

[96] TOCHTER, K. D. Techniques of muliplication and division for automatic binary computers. In *Quart. J. Mech. Appl. Match* (1958), pp. 364–384.

[97] YANG, B., CHEN, Y.-A., BRYANT, R. E., AND O'HALLARON, D. R. Space- and time-efficient bdd construction via working set control. In *Proceedings of ASP-DAC '98* (Yokohoma,Japan, Feb. 1998), pp. 423–432.

[98] YAO, Y. Chromatic's Mpact2 boosts 3D. In *Microprocessor Report* (November 1996), pp. 1–10.

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