Formal Verification Using Parametric Representations of Boolean Constraints

Mark D. Aagaard, Robert B. Jones, Carl-Johan H. Seger
Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA

Abstract *We describe the use of* parametric representations *of Boolean predicates to encode data-space constraints and significantly extend the capacity of formal verification. The constraints are used to decompose verifications by sets of case splits and to restrict verifications by validity conditions. Our technique is applicable to any symbolic simulator. We illustrate our technique on state-of-the-art Intel(R) designs, without removing latches or modifying the circuits in any way.*

1 Introduction

When automated formal-verification techniques, such as model checking, reach their capacity limits, verification tasks must be simplified or decomposed into smaller pieces. In this paper we use *parametric representations* of Boolean predicates to increase the effective capacity of symbolic-simulation based verification. The parametric representation allows us to impose constraints such that we simulate circuits only with data that satisfies the constraints. We use this to mitigate capacity limitations by only simulating legal values and case splitting over the legal values to decompose a verification task into a set of smaller tasks. Different BDD variable orders can be used for different case splits, which also helps address capacity problems.

Our technique is independent of the symbolic simulation algorithm, does not require any modifications to the circuit, can be used to constrain any signal mentioned in the property, and is applicable to a wide variety of circuits. We describe the application of our approach to a synthetic hidden-weighted bit circuit, an Intel IA-32 instruction decoder, and an Intel floating-point adder/subtracter.

Symbolic simulation plays several roles in formal verification. In reachability based model-checkers (such as SMV [4]), the transition relation for the circuit is typically computed using symbolic simulation. Symbolic simulation can also form the basis for the verification algorithm itself, as is done in symbolic trajectory evaluation [13].

The parametric representation is an alternative to the more common representation of sets as characteristic functions. Rather than a single expression, a parametric representation is an array of expressions. This array has two defining features. First, each expression in the array corresponds to a variable, usually one that is in the support of the characteristic function. Second, all instantiations of the variables in the parametric representation result in a vector that is an element of the set.

DAC 99, New Orleans, Louisiana

As an example, consider the set:

$$S = \{\langle 1001\rangle, \langle 1000\rangle, \langle 0101\rangle\}$$

If we represent the 4-bit array with the variables a_3, a_2, a_1, and a_0 then a non-minimized characteristic function is:

$$a_3\overline{a_2}\,\overline{a_1}\,a_0 + a_3\overline{a_2}\,a_1\,\overline{a_0} + \overline{a_3}\,a_2\overline{a_1}\,a_0$$

We can represent the same set using a *parametric representation* with *parametric* variables p_1 and p_0 as:

$$\langle p_1, \overline{p_1}, 0, \overline{p_1} + p_0\rangle$$

Below, we list the resulting vector for all possible assignments to the variables p_1 and p_0. Note that while $\langle 0101\rangle$ appears twice, the range of the parametric vector is exactly the set S.

Parametric variables		Resulting vector
p_1	p_0	$\langle p_1, \overline{p_1}, 0, \overline{p_1} + p_0\rangle$
0	0	$\langle 0101\rangle$
0	1	$\langle 0101\rangle$
1	0	$\langle 1000\rangle$
1	1	$\langle 1001\rangle$

Many parametric representations are possible; we could also represent S with:

$$\langle \overline{p_1} + \overline{p_0}, p_1 p_0, 0, p_1\rangle$$

There are many ways to decompose a verification task. Perhaps the most common technique for datapath verification is structural decomposition—verifying individual pieces of the circuit and then composing these verification results to prove the top-level correctness statement. Although structural decomposition has the theoretical advantage of unlimited capacity, it requires significant manual effort and detailed knowledge of the circuit implementation.

With data-space partitioning (case-splitting), we decompose the data space into a number of sets and separately verify the circuit for each set. We have found that case splitting generally requires much less effort than structural decomposition. This is because it is easier to compose the verification results and it requires only minimal knowledge of the circuit implementation. As our examples illustrate (Section 5), finding a set of cases that significantly reduces the size of the BDDs requires some understanding of the *algorithm* that the circuit implements, but usually does not require any knowledge of the internal structure (e.g., signal names or timing). In our verification system, once a set of case splits has been chosen, the verification and justification of the correctness of the decomposition is automated. Case splitting is also more robust in the face of internal design changes, as we are usually able to avoid mentioning internal signals in our properties and verification scripts.

Combining the parametric representation with case splitting reduces verification complexity by eliminating the simulation of data values that do not satisfy the given constraint. We write a generic correctness statement as:

$$P(\vec{x}) \Longrightarrow Q(\vec{x})$$

where P is a constraint on the data space, $\vec{x}$ is a vector of BDD variables, and Q is a function that carries out the symbolic-simulation based verification. A conventional approach would perform this verification in three steps:

1. evaluate $P(\vec{x})$, resulting in p
2. evaluate $Q(\vec{x})$, resulting in q
3. evaluate $p \Longrightarrow q$

This approach has the disadvantage that it evaluates Q for *all* values of $\vec{x}$, not just those that satisfy P.

We improve on this by using the parametric representation to evaluate Q only for values of $\vec{x}$ that satisfy P. We use a function `param` to compute a parameterized functional vector representation of P and rewrite the correctness statement as:

$$Q(\texttt{param}(P(\vec{x})))$$

Now, the constraints imposed by P are encoded inside the symbolic simulator, and it does not have to build expressions for combinations of values that are not of interest. In general, restricting the data space is not guaranteed to decrease simulation complexity. For certain circuit structures, the wrong choice of restrictions can actually increase the complexity of the calculation.

To be effective, computing a parametric representation and symbolically simulating for a subset of data values must be more efficient than symbolically simulating with unconstrained data. For a decomposition to be sound, the data partitions must cover all of the relevant values. Our verification system integrates the parametric representation with symbolic trajectory evaluation [13] and a lightweight theorem prover [1]. Using the theorem prover helps organize our verification process and achieve sound verification results.

1.1 Related Work

The two classes of work that are most similar to ours are alternative representations of Boolean predicates and the use of case splitting in model checking.

Jain and Gopalakrishnan [12] reviewed a variety of representations for Boolean predicates, including Boole's method, Lowenheim's method, and the generalized-cofactor method. Our method is based on a recursive Shannon decomposition of the predicate and is most similar to Boole's method [2]. One difference between our method and Boole's is that we provide unconstrained parametric variables for nodes that are not restricted by the predicate. This results in parametric vectors that are more convenient for direct use in hardware verification.

Coudert *et al* presented an approach for deriving a parametric representation, which they called a *functional representation* [6]. In a later paper [7], Coudert and Madre described their algorithm using a constraint function called *cnst*. Applying *cnst* to a vector of a predicate's support variables yields the same result as our parametric algorithm applied to the same vector, assuming that the variables appear in the vector according to their position in the BDD variable ordering. However, our parametric function requires fewer recursive BDD calls, since we compute the complete vector in one recursion over the constraint BDD. We have seen examples where our parametric algorithm is almost an order of magnitude faster than applying *cnst* on an input vector. We have not yet encountered an example where Coudert *et al*'s approach is faster than our parametric algorithm.

Jain and Gopalakrishnan [12] used a variety of customized parameterization schemes that were defined recursively and tailored to the hardware being verified. The industrial circuits we commonly encounter, including even the floating-point adders in Section 5, are not sufficiently regular to describe recursively. Their technique also has the limitation that parameterization predicates must be expressed in disjunctive normal form. This is computationally infeasible for the complex predicates typical of large circuits.

Chen and Bryant [5] used constraints and case splitting to verify two floating-point adder/subtracters with SMV. In our approach, we parameterize our data vectors prior to simulation and then use conventional symbolic simulation. By comparison, Chen and Bryant modified the way that SMV builds transition relations. After the symbolic simulation of each gate in the circuit, they used Coudert and Madre's *restrict* function [7] to simplify the partially-built transition function. Chen and Bryant also modified the circuit. They removed all latches (essentially converting the datapath to a combinational circuit). Latch removal is not possible for pipelines with loops, significant control circuitry, or non-trivial clocking schemes.

Chen and Bryant's two adders were relatively similar to each each other, were verified for a single floating-point format and rounding mode, and the verifications included neither special cases (NaNs, infinities, etc.) nor flags (overflow, underflow, etc.). The two floating-point adder/subtracters we verified (Section 5.2) have very different implementations and the verifications included three different formats, all IEEE rounding modes, the output flags, and proprietary special cases. In addition to being more complete, our work provides several other advantages over that of Chen and Bryant. First, because parameterization is done prior to simulation, it can be used with any symbolic simulator—no modifications to existing verification engines are necessary. Second, symbolic trajectory evaluation can verify the datapaths of complex pipelined circuits without modifying the circuit. Finally, we demonstrate that our approach is applicable to a variety of arithmetic and non-arithmetic circuits.

2 Computation of a Parametric Representation

In choosing a parametric representation we try to minimize the computational resources needed both to compute the parametric representation and to symbolically simulate the circuit. After experimenting with several heuristics, we chose one that targets simpler parametric expressions at the cost of additional Boolean variables. We found that minimizing variables usually leads to complex parametric functions, which in turn makes symbolic simulation more expensive.

Our algorithm for computing the parametric representation is fairly straightforward. Pseudo-code for the core function is given in Figure 1. We illustrate the function with four arguments: `var_vec` is an array of BDD variables in the support of `cond`, `pvar_vec` is an array of fresh BDD variables to be used in the parametric representation, `index` is initially the length of `var_vec`, and `cond` is the predicate (a BDD) to be parameterized. Strictly speaking the function $\texttt{param}(P(\vec{x}))$ in Section 1 should be written $\texttt{param}(\vec{x}, \vec{y}, \text{length}(\vec{x}), P(\vec{x}))$, where $\vec{y}$ is a fresh set of parametric variables. We omit the extra arguments in our notation for clarity in presentation.

If the predicate is not satisfiable, then it does not have a parametric representation, and so the algorithm exits with a fatal error (ln 10–11). The algorithm performs a recursive Shannon decomposition of the predicate by iterating through `var_vec` (ln 15,20,23,26–27), and taking the positive and negative cofactors of the refined predicate (ln 16–17). If the positive (negative) cofactor is empty, then the variable at this position cannot be `T` (`F`) under any interpretation, so a scalar `F` (`T`) is returned (ln 18–23).

If both cofactors are non-scalar (neither `T` nor `F`), then the appropriate parametric variable (from `pvar_vec`) is inserted in this position, and `param` is called recursively for both the positive and negative cofactors (ln 26–27). In each case, the final result vector `res` is constructed by combining the recursively generated results (ln 20,23,26–30).

As is typical of BDD applications, caching intermediate results (ln 12–13,33) is critical to the performance of the algorithm: it often means the difference between exponential and linear runtimes.

```
 1: param(bdd var_vec[], bdd pvar_vec[], int index, bdd cond)
 2: // var_vec is an array of BDD variables
 3: // pvar_vec an array of the parametric variables
 4: // index is length of varlist on initial call
 5: // cond is Boolean predicate (BDD) to parameterize
 6: {
 7:     bdd bit, low, high;
 8:     bdd lres[index-1], hres[index-1], res[index];
 9:
10:     if (cond = F)
11:         fatal (''predicate not satisfiable'');
12:     if (exists_cached(var_vec, pvar_vec, index, cond))
13:         return (cached(var_vec, pvar_vec, index, cond));
14:     if (index = 0)
15:         return T;
16:     low  := negative_cofactor(cond, var_vec[index]);
17:     high := positive_cofactor(cond, var_vec[index]);
18:     if (low = F) {
19:         bit  := T;
20:         res := param(var_vec, pvar_vec, index-1, high);
21:     } elseif (high = F) {
22:         bit  := F;
23:         res := param(var_vec, pvar_vec, index-1, low);
24:     } else {
25:         bit  := pvar_vec[index];
26:         lres := param(var_vec, pvar_vec, index-1, low);
27:         hres := param(var_vec, pvar_vec, index-1, high);
28:         for i := 1 to index-1 {
29:            res[i] := BDD_ite(bit, hres[i], lres[i]);
30:         }
31:     }
32:     res[index] := bit;
33:     insert_cache(var_vec, pvar_vec, index, cond, res);
34:     return res;
35: }
```

Figure 1: Pseudo-code for parametric vector computation

The main correctness characteristics of the algorithm, which are similar to those presented by Coudert *et al* [6], are summarized in Theorem 1.

Theorem 1 *Parametric algorithm*
Assume $P(\vec{x})$ is a satisfiable predicate over a vector of Boolean variables $\vec{x}$. Let $P'(\vec{y})$ denote the result of evaluating `param`$(P(\vec{x}))$*, as defined in Figure 1.*

1. *All parametric vectors satisfy P:*
$$\forall \vec{y}.\, P(P'(\vec{y}))$$
2. *The range of the parametric vector covers all satisfying vectors of P:*
$$\forall \vec{x}.\, \left(P(\vec{x}) \Longrightarrow \exists \vec{y}.\, \vec{x} \equiv P'(\vec{y})\right)$$

An important aspect of the algorithm is that, if the variables are applied according to the BDD variable order, then the sizes of the parametric functions are bounded by the size of the characteristic function ANDed with a free variable. Thus, if we can compute the characteristic function for the predicate, we can also compute the parametric representation.

There are tradeoffs in choosing to use a parametric representation rather than the conventional characteristic function. The main motivation for using the characteristic representation in symbolic model checking is that it supports all set operations efficiently and is a canonical representation [4]. In contrast, the parametric representation is not canonical and supports only set union efficiently. The distinct benefit of parameterization is that symbolic simulation, and thus image computation, is done only for the particular data of interest.

3 Applying Parameterization in Input-Space Partitioning

We use the synthetic "hidden-weighted-bit" circuit (Figure 2) to demonstrate the power of data-space partitioning with parametric representations. The circuit counts the number of 1s in the vector `input`, and outputs the input bit corresponding to this count.

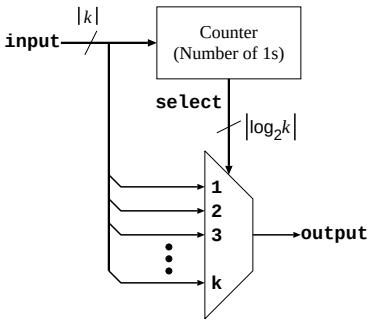

Figure 2: Hidden-weighted-bit circuit

We make the environmental assumption that the input vector is never all 0s. The output of the circuit requires a BDD of exponential size in terms of k (independent of variable ordering) [3]. Intuitively, the BDD size is exponential because as a given variable is read in any path down the BDD, its value must be remembered until all variables have finally been read. Not until then can it be determined whether or not the original variable will affect the value on the output. A similar phenomenon leads to the exponential-size BDDs of multipliers and floating-point adders.

We can use input-space partitioning to verify a hidden-weighted bit circuit in k symbolic simulation runs. In each run i, we parameterize the input data such that exactly i bits are 1. For run i, `select` will be i and `output` will be `input`$[i]$. Thus the size of the output BDD will simply be the size of the parameterized input bit i.

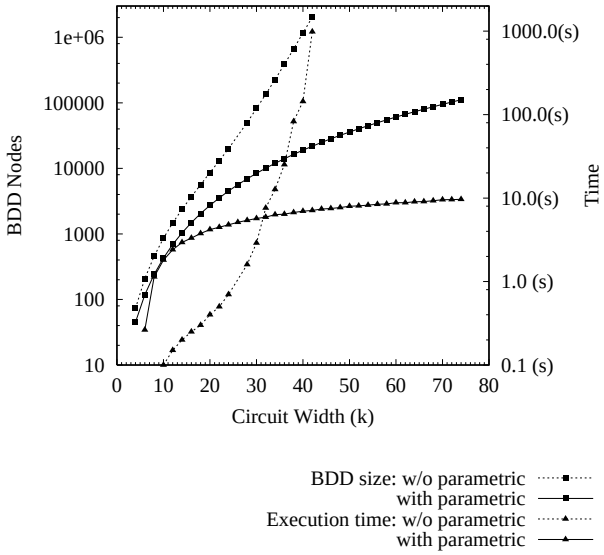

Figure 3: Verification of hidden-weighted-bit circuit with and without input-space partitioning

To illustrate the efficacy of parametric representations, in Figure 3 we contrast direct verification of the circuit with verification using parameterized input-space partitioning. The figure illustrates run times and maximum BDD sizes encountered during verification for different circuit widths. Clearly, using parametric representations greatly increases capacity.

Theorem 2 formalizes the intuition behind input space partitioning:

Theorem 2 *Input space partitioning*
Given a satisfiable validity predicate $P(\vec{x})$ over a vector of Boolean variables $\vec{x}$, a predicate to be proved $Q(\vec{x})$, and a set of case-splitting predicates $P_i(\vec{x})$, $(1 \leq i \leq n)$:
If

1. *all of the case-split predicates are satisfiable:*
$$\text{for } 1 \leq i \leq n : \exists \vec{x}.\ P_i(\vec{x})$$
2. *the union of all the subcases covers the global validity condition:*
$$\forall \vec{x}.\ P(\vec{x}) \Longrightarrow \left(\bigvee_{i=1}^{n} P_i(\vec{x}) \right)$$
and
3. *the verification of each subcase (using the parametric representation) is successful:*
$$\text{for } 1 \leq i \leq n : \forall \vec{x}.\ Q(\texttt{param}(P_i(\vec{x})))$$

then we can conclude:
$$\forall \vec{x}.\ P(\vec{x}) \Longrightarrow\ Q(\vec{x}).$$

Condition 3 of Theorem 2 is where the actual verification takes place. Condition 1 can almost always be solved by BDD-based tautology checkers. Solving Condition 2 for large BDDs that require different variable orderings can be problematic with simple tautology checking. This leads to the need for more sophisticated decision procedures and motivates the integration of theorem proving capabilities.

4 Our Implementation

We have created a verification environment that supports symbolic simulation, theorem proving and tautology checking. For our symbolic simulator, we use the Voss [9] implementation of symbolic trajectory evaluation [13]. Voss provides a functional, strongly-typed programming language with built-in BDDs. This provides an able platform for developing theorem proving and tautology checking capabilities.

Tight integration between the model checker and tautology checker makes large verifications feasible, because both tools use the same BDD variable orders and associated data structures. Similarly, the close connection between the model checker and the theorem prover provides an added degree of confidence that has proved beneficial. For example, we did not discover the need for Condition 1 in Theorem 2 until we formalized the use of the parametric representation in the theorem prover. The theorem prover ensures that the case splits that we symbolically simulate are *exactly* those that we use to check the side conditions. In each major verification we have performed, the theorem prover has helped us find mistakes in our verification scripts. For example, we found a missing boundary condition with respect to exponent biases in one of the floating-point adder/subtracter verifications.

5 Examples

We have applied the parametric representation to verify several industrial circuits, as summarized in Table 1. Note that the table does not count all of the latches in the circuits, but only those actually used in the verification. All of the runs reported were performed on a 120MHz HP 9000[1] with 786MB of physical memory.

The instruction marker (IM) is part of the decode pipeline of an Intel microprocessor; it marks the boundaries between Intel architecture (IA-32) instructions. The two Add/Sub circuits are very different implementations of floating-point addition and subtraction. The control circuit handles retirement duties of an IA-32 processor. Two of the examples, the IM and Add/Sub1 are described in Sections 5.1 and 5.2.

[1] All trademarks are property of their respective owners.

Table 1: Verification Statistics.

Circuit	Latches	BDD nodes	Cases	Time
IM	1100	4.2M	28	8 hr
Add/Sub1	3600	2.0M	333	15 hr
Add/Sub2	6900	1.5M	478	24 hr
Control	7000	0.8M	126	16 hr

All verifications were performed on unmodified, gate-level circuits—we did not remove any latches or simplify the hardware in any manner. Each example used the parametric representation to restrict the data signals to legal values and to perform case decomposition. Even with decomposition, most of the individual verification runs pushed the capacity limits of our state-of-the-art symbolic simulation engine.

For clarity, we illustrate our case splits as tree diagrams. In fact, our tools allow arbitrary collections of cases to be checked against the criteria in Section 2. This allows great flexibility in structuring verifications, which in turn allows more readable verification scripts. Discovering an effective and correct set of case splits is one of the biggest challenges in verifying a circuit. It requires an understanding of the algorithm the circuit implements, the ability to predict BDD behavior on a wide variety of functions, and a familiarity with mathematical logic.

5.1 Instruction Marker (IM)

Decoding Intel architecture (IA-32) instructions is complicated by the fact that different instructions have different lengths, the size of the instruction set (several hundred opcodes [11]), and the existence of prefix bytes that affect the *length* of the subsequent instruction.

In modern IA-32 microprocessors, one of the initial steps in instruction decoding is to find and mark the boundaries between instructions. We verified the datapath of a circuit (the IM) that computes the length of each instruction it sees and then marks the boundaries between instructions. The IM processes a fixed number of bytes simultaneously. We call these bytes a *packet*.

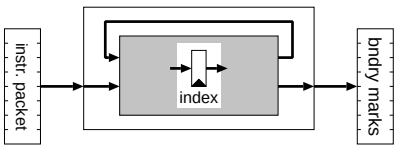

Figure 4: High-level structure of IM

The IM is implemented as a pipelined datapath with internal state (Figure 4). For the last instruction in each packet, the IM computes the number of bytes that overflow in the next packet. This number, along with some additional information, is stored as the internal state. We developed a validity predicate that characterizes all legal IA-32 instructions and a length specification for all IA-32 instructions. More details are available in an earlier paper [1].

To bring the complexity of the symbolic simulation within the range that could be handled by our tools, we restricted the input to legal IA-32 instructions and performed a set of case splits. We case split on whether the processor was in a 16 or 32-bit addressing mode and on the value of the internal IM state: $i_0 \ldots i_n$ (Figure 5). All of the verification runs used the same variable ordering.

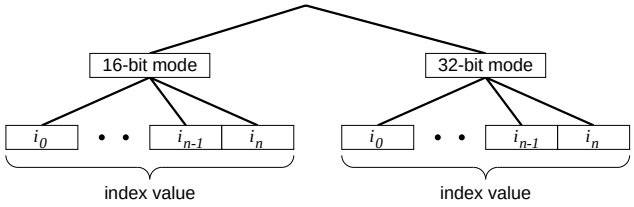

Figure 5: IM case splits

Finding the right decomposition strategy was quite a challenge for the IM. We contemplated a variety of approaches, such case-splitting on the size of instructions, but struggled to find an approach that was feasible for both symbolic simulation and justifying the decomposition. It was only after we fully understood the high-level algorithm for the IM that we arrived at a successful decomposition strategy. When experimenting with the strategy we studied how the circuit represented the internal state. However, this was the only part of the internal implementation that we were forced to understand in detail. Case splitting and the parametric representation allowed both us and our verification script to be oblivious to most of the implementation details.

The greatest benefit of the parametric representation for the IM is in the interaction between the case splits and the validity condition (input stream contains only legal IA-32 instructions). For example, if the index says the first byte in the next packet begins a new instruction, then the bytes beginning at that position must be a legal IA-32 instruction. Although our case splits constrain the internal state to be scalar (a vector of 0s and 1s without symbolic variables), the resulting predicate for a legal IA-32 instruction stream is a very complex symbolic expression.

In this example, as with the others, we established the side-conditions that (1) each case split was satisfiable and (2) that the case splits taken together covered the original validity predicate. A total of eight previously unknown IM bugs were discovered during our verification of this pre-silicon design. Furthermore, four of the eight bugs were considered by the chip design team to be "high-quality"—meaning that they would have been difficult to find and diagnose with traditional validation techniques.

5.2 Floating-Point Addition and Subtraction

In floating-point addition and subtraction, we distinguish between "true" and general operators. True addition occurs with addition when the operands have the same sign and with subtraction when the operands have opposite signs. Similarly, true subtraction occurs with subtraction when the operands have the same sign and with addition when the operands have opposite signs.

The arithmetic portion of our specification was based upon a textbook algorithm presented by Feldman and Retter [8]. Specifications for flags and special cases were based on IEEE Standard 754 [10] and proprietary architectural documentation.

Figure 6 shows the outline of the cases we used, which are very similar to those that were independently chosen by Chen and Bryant [5]. The chief BDD-capacity problem is caused by shifting the significands so that they are aligned. The amount to shift is determined by the difference between the two exponents.

In our preliminary experiments, we discovered that a single trajectory-evaluation run could handle a small range of shift amounts. The size of the range varied with the difference of the exponents, which would have made it quite difficult to find a minimum-size set of cases. For simplicity, we chose each case to be a particular shift amount.

The top set of case splits in Figure 6 cover the situations where the exponents differ by more than the width of the significand ($e_1 - e_2 > n$ and $e_2 - e_1 > n$), the exponents differ by more than one ($n \geq e_1 - e_2 > 1$ and $1 < e_2 - e_1 \geq n$), and the exponents differ by zero or one ($1 \geq e_1 - e_2 \geq -1$). When the exponents differ by more than the width of the significand, the significands do not overlap and no alignment shift is necessary. The result is, modulo rounding, the larger of the two operands. Post-rounding normalization requires shifting the significand by at most one bit, which can be easily handled by BDDs.

With floating-point true subtraction, when the exponents differ by zero or one, the result can be much smaller than the operands and require renormalization. For example:

$$1.0011010 - 1.0011001 = 0.0000001$$

which must be shifted 7 bits to the left to become normalized. Renormalization is not needed with true addition. This leads to a case split between true addition and subtraction when the exponents differ by zero or one. With true addition, we have three cases (1, 0, and -1). With true subtraction, we have one case for each possible renormalization amount.

6 Conclusion

Data-space partitioning is a valuable technique for decomposing verification tasks into smaller, more-manageable pieces. Its primary advantage over structural decomposition is that the verification can be accomplished without exposing internal implementation details of the circuit. This is especially important for large industrial circuits where the internal timing is complex and where it is difficult to write clean specifications for sub-components.

In addition to enabling case-splitting without modifying the circuits or our verification engines, the parametric representation enables us to place constraints on input and state spaces. This allows us to verify circuits under only the conditions of interest, which can dramatically increase the effective capacity of automated formal verification tools. Together, these two aspects of encoding data-space constraints have enabled us to verify circuits much larger than would otherwise have been feasible.

Acknowledgments

We thank Clark Barrett for his instrumental work in the original version of the IM proof, Tom Melham for writing the initial FADD specification, Donald Syme for helping with the decision procedures to justify the floating-point case splits, and Mike Jones for his comments on a draft of the paper.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *ACM/IEEE Design Automation Conference*, pages 538–541. ACM/IEEE, July 1998.

[2] G. Boole. *The Mathematical Analysis of Logic*. Macmillan 1847. Reprinted 1948, B. Blackwell, 1847.

[3] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with applications to integer multiplication. *IEEE Transactions on Computers*, C-40(2):205–213, Feb. 1991.

[4] J. Burch, E. Clarke, and K. McMillan. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference*, pages 46–51. ACM/IEEE, 1990.

[5] Y.-A. Chen and R. Bryant. Verification of floating-point adders. In A. J. Hu and M. Y. Vardi, editors, *Workshop on Computer-Aided Verification*, pages 488–499, July 1998.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 179–196, Nov. 1989.

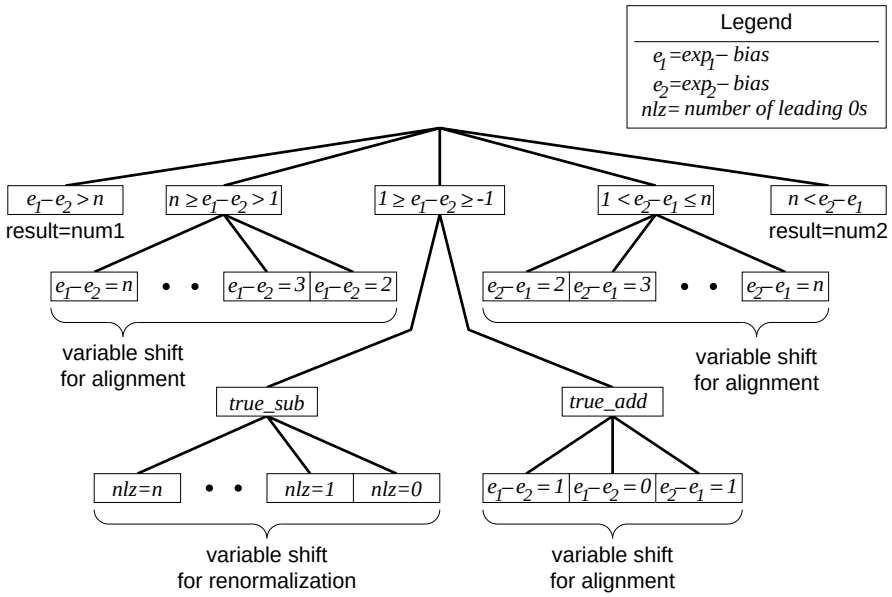

Figure 6: Case splits for combined addition/subtraction

[7] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design*, pages 78–82, Nov. 1990.

[8] J. M. Feldman and C. T. Retter. *Computer Architecture*. McGraw-Hill, 1994.

[9] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3–78. Springer Verlag; New York, 1997.

[10] IEEE. *IEEE Standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985, 1985.

[11] Intel. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation, 1993.

[12] P. Jain and G. Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *IEEE Transactions on Computer Aided Design*, 1994.

[13] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, Apr. 1994.