Formally Verifying Arithmetic Circuits — Avoiding the PENTIUM® FDIV bug

Sérgio Campos, Manpreet Khaira, Xudong Zhao —Intel Development Laboratories

Edmund Clarke — Carnegie Mellon University

Overview

This paper describes a major breakthrough in the verification of arithmetic circuits. Using a methodology developed by IDL, formal verification can now be applied to floating point and integer arithmetic circuits. This is the only known method that allows the full data path of complex arithmetic circuits to be formally verified. Using this technology we have reproduced the Pentium® FDIV *and* the P6 IDIV bugs. Moreover, we have also been able to verify the correctness of the modified circuits.

The Pentium® FDIV bug highlighted the importance of this technology. Several analyses of the error concluded that the only way to avoid it was to use formal verification because exhaustively covering all possible test cases using simulation is impossible. They also reported that none of the current formal verification approaches could have been used, due to the complexity of the circuit.

This new technology sets a new standard in this area by combining *symbolic model checking* [3], the engine behind Intel's Prover formal verifier, and a new data representation called *binary moment diagrams* [1]. These methods allow much larger circuits to be formally verified. Currently, we are verifying circuits with 600 variables, more than three times larger than practical with previous methods. Even though the circuits are very complex, most properties can be verified in minutes, using 10 to 20 megs of memory. We are currently verifying the P6 floating point multiplier and adder. We also plan to verify most of the execution cluster.

Introduction

A very efficient technique for verifying the correctness of sequential circuits is *symbolic model checking* [3]. It is based on binary decision diagrams (BDD) and has been very successful in verifying the control logic of industrial circuits at Intel [4] and elsewhere. However, the BDD representation is inherently exponential for some functions that occur frequently in the data path. This prevents their use in the verification of arithmetic circuits.

Recently, however, data structures that allow an efficient representation of such functions have been derived from BDDs. These are binary moment diagrams (BMD) [1] and multi-terminal BDDs (MTBDD). The first allows an efficient representation of the data path, while the second effectively models the control path. We have combined both methods introducing *hybrid decision diagrams*. By using this representation we are able to handle circuits with both control logic and wide data paths.

Extensions to Binary Decision Diagrams

A binary decision diagram (BDD) is a canonical representation for a boolean formula. It is a binary decision graph where there is a total order placed on the occurrence of variables as one traverses the graph from root to leaf.They are often substantially more compact than traditional representations such as conjunctive or disjunctive normal forms. They can also be manipulated very efficiently. Hence, BDDs are widely used for a variety of CAD applications, including symbolic simulation and verification of combinational logic and sequential circuits.

Multi-Terminal BDDs have a similar structure. However, while BDDs have boolean variables and leaves, MTBDDs have boolean variables, but integer leaves. They therefore can represent functions from booleans to integers. Functions from integers to integers can also be represented when the input is encoded in binary form. There are efficient algorithms that compute many arithmetic operations, such as addition and multiplication, when operands are given in this form.

Recently a new representation for functions that map boolean vectors to integers has been developed. This representation is called binary moment diagram. It is also traversed from top to bottom, and the node with variable x is computed recursively by $f = f_L + x\, f_R$, where f_L and f_R are the values of the left and right children of this node. This representation gives a compact representation for certain functions which have exponential size if represented directly by MTBDDs.

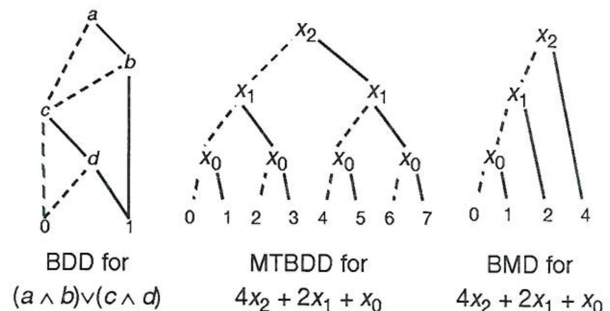


Figure 1 Examples of Decision Diagrams
(Follow full lines if variable is true; dotted lines if it is false)

Verifying the Data Path

Though symbolic model checking techniques based on Binary Decision Diagrams (BDDs) have successfully verified control logic, there are fundamental problems with applying BDDs or even MTBDDs to the verification of arithmetic circuits. The number of possible values for the data

variables is exponential in the number of bits, making their representation also exponential.

Bryant and Chen have shown that BMDs provide a compact representation for certain functions that have exponential MTBDDs [1]. Their method determines that a circuit is correct when the BMDs for the circuit and the specification are exactly the same. They have used this representation to verify the data path of arithmetic circuits. However, there can be cases in which the circuit is correct but the BMDs are not identical. Since certain combinations of control variables can never occur, the behavior of the circuit in those states is irrelevant, resulting in distinct but correct BMDs. Moreover, this technique has other limitations. For example, it cannot handle inequalities.

We have integrated both approaches by using a combination of MTBDDs and BMDs in the verification of arithmetic circuits. We call these hybrid decision diagrams (HDD). In particular, for state variables corresponding to data bits, this representation behaves like a BMD; while for state variables corresponding to control signals, it behaves like a MTBDD. By using this representation, we are able to handle circuits with both control logic and wide data paths.

We have extended the symbolic model checking system SMV [3] to allow the expression of properties involving relationships among data variables. Originally, properties could only reason about state variables. In the extended system, we allow properties that relate the value of collections of data variables. These collections, called words, are arrays of single bit state variables. An arithmetic expression is constructed from words in the circuit, constants and arithmetic operations on words. These expressions are represented by hybrid BDDs.

For example, the extended SMV system can handle properties such as *DataOut* = *DataIn1* + *DataIn2*, where each of these variables is an array of 68 bits. Such properties cannot be expressed in the original SMV system.

Arithmetic Operations and Relations

The following operations on integer valued functions are implemented in the extended SMV system. Addition and scalar multiplication have linear complexity. The worst case complexity for multiplication of two functions is exponential, but in practice, we are able to compute it efficiently. Finally, we have also implemented if-then-else operations.

Model checking of word level properties requires computing which variable values satisfy an arithmetic relation. We have developed an algorithm that can substantially reduce the cost for computing this information. Suppose that we want to compute the values that satisfy $f > 0$. Each branch in the hybrid decision diagram for f corresponds to a subset of variable values. If the maximum value of a branch is less than or equal to 0, then none of the values in this branch satisfy the inequality. If the minimum value of a branch is greater than 0, then all its values satisfy the inequality. In both cases, we avoid checking the signs of the individual assignments in the branch.

The improved algorithm works extremely well for verification of arithmetic circuits if the expressions in the specification are linear. Most of the formulas that occur in the verification of the SRT division algorithm are linear.

Symbolic Model Checking Word Properties

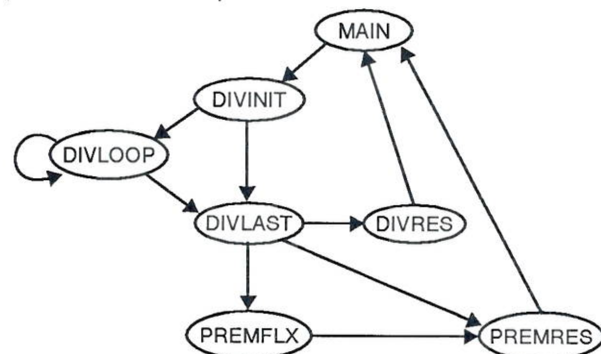
Model checking is a technique that, given a state-transition graph and a property expressed as a temporal logic formula [3], determines which states in the graph satisfy the property. There exist algorithms to perform this task in time linear to the size of the formula and graph. However, constructing the state graph is an exponential problem given its parallel components. Since most practical problems are given in such terms, this puts a limit on the size of problems that can be handled. In symbolic model checking systems, BDDs are used to represent the transition relation and sets of states. The model checking process is performed iteratively on these BDDs. Symbolic model checking techniques increased dramatically the size of the systems that can be verified.

However, in spite of being successful in verifying control logic, model checking algorithms cannot be directly used for verifying arithmetic circuits. Expressions that involve variables with integer values cannot be handled properly. Word level model checking overcomes this problem by extending the original algorithms to determine the value of arithmetic expressions using hybrid decision diagrams.

With our technique, the model is represented using BDDs as in the original algorithm. HDDs are used only to compute the value of expressions. The hybrid decision diagram representation of a word is computed by converting each bit of the word into an HDD and summing the resulting values. Although this process is exponential in the worst case, it is very efficient in practice. The algorithm described to obtain the BDD representing the set of variable assignments that make an algebraic relation true is used to convert the HDD representation of expressions to the BDD representation of relations. After the BDD representation for relations is generated, the BDDs for temporal formulas are computed in the same way as in standard model checking. In particular, the iterative computations are exactly the same in both cases.

Verification of the P5 Division Circuit

Using word level model checking, we have successfully verified the circuits, both control and datapath, that implement the SRT division and square root in the Pentium® processor. These circuits perform two different operations, division and remainder, and its phases can be seen in the figure below. In both cases it transitions to the DIVINIT state and then to the DIVLOOP state. It iterates for at most 34 cycles in this state. After this phase, the upper path through DIVRES is taken in case the operation is division. The lower path is taken if the operation is remainder.



We have verified the control logic and the full 70 bits data path of the circuit. Let *r* be partial remainder, *q* be quotient, *d* be the divisor and *D* the dividend. We have checked the properties:

- Loop invariant: $r + q\,d = D$
- The new partial remainder is within the bounds defined by the algorithm: $-8/3\ d \leq r \leq 8/3\ d$.

For example, we have proven that at the DIVINIT state, the remainder is the dividend, the quotient is zero and therefore, both properties hold. The formulas verified are:

```
AG(state=DIVINIT -> r = dividend & q = 0)

AG(state=DIVINIT -> -8*d <= 3*r <= 8*d)
```

We have also shown that the second property always holds in the DIVLOOP state, and that $r + q\ d$ is invariant with respect to left shifting (the results of iteration *i* are left shifted to be used in iteration *i + 1*):

```
AG(state=DIVLOOP -> (-8*d <= 3*r <= 8*d))

AG((state=DIVLOOP & (-8*d <= 3*r <= 8*d)) ->
    ((r + q*d) * 4 = next(r + q*d)))
```

These properties guarantee that the invariant above holds in the DIVLOOP state. Similar properties have been proven for the DIVLAST, DIVRES, PREMFLX and PREMRES states.

Error in the Old Quotient Prediction Table

This verification procedure was also applied to the old design of P5 division circuit. We have built an SMV model manually translated from the iHDL code. The exact same error that appears on the Pentium® processor was then reproduced.

However, the verification of the second property generates an exponential size BDD, and does not finish. Thus, an indirect approach was used. We broke the property into three subproperties:

- The inequality holds for the initial state.
- If the inequality holds in the current state, then it is also satisfied in the next state.
- All reachable states are defined in the table.

The error appears when the third property is checked. There can be a maximum of 4096 entries in the quotient table, even though many are unreachable. The corrected quotient table redundantly includes all 4096 entries, but the old one attempts to exclude redundant entries. Let `error` stand for a state not in quotient table. To show that all reachable states are defined we must prove:

```
AG (state=DIVLOOP -> !error)
```

However, this property also generates an exponential size BDD. We overcome this problem by restricting the search depth, since the circuit always completes the division in less than 34 cycles. We use the bounded operator:

```
ABG 0..34 (state=DIVLOOP -> !error)
```

We proved that for the old quotient table, only 5 terms are missing. These correspond to the terms in which the leading 5 bits of the divisor are 10001, 10100, 10111, 11010 and 11101. We have found counter examples for each of them. The counterexamples that generate erroneous division results for each of the 5 missing terms are:

10001	7865467 / 4718591
10100	7341819 / 5505023
10111	8391671 / 6291455
11010	9437175 / 7077887
11101	20979183 / 15728639

Moreover, in each of the 5 assignments of the leading bits of the divisor, the error can only possibly appear when those bits are followed by one of the following patterns:

111111000000,
111111110111,
111111110000000,
111111111000,
11111100100000,
111111111101,
11111111111001111,
11111111111.

The following properties provide some insight into the magnitude of the error due the missing terms in the quotient table. The error can never appear in the first 6 radices. It can only appear at the 12th bit (4th digit) or later. This is guaranteed by the property `ABG 0..5 !error`. An error can appear at the seventh radix only when the leading divisor bits are followed by 11111111111. For all the other bit patterns, property `ABG 0..6 !error` is true. These results can potentially be used to provide a formal proof for some of Intel's claims in [2].

Verification of the square root circuit

We have also proved the correctness of the circuit for computing square roots. We have manually built a SMV model for the Pentium® square root circuit from the iHDL code. We verified the following property that immediately guarantees the correctness of the square root.

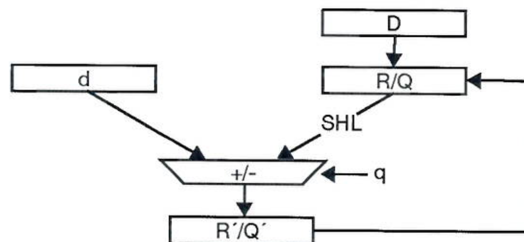
*partial_result = (sum + sqrt0 * sqrt0)<<68 + qpos) &*
*next(partial_result) = partial_result * 4*

Where, *qneg* is the square root being computed; *qpos* contains the value of the radicand; and *sum* contains the value of the remainder. The value of *sqrt0* is the value of *qneg[2:69]* followed by a '1' bit. The above formula states that *(sum + sqrt0 * sqrt0)<<68 + qpos)* stays constant with respect to left shifting. The initial value of the expression is the radicand, guaranteeing that when the algorithm ends, the square of the square root computed plus the remainder is equivalent to the radicand, the expected behavior.

Verification of the P6 IDIV circuit

The examples described above have been manually translated from iHDL to SMV. This makes the verification process more complex, because knowledge of the SMV system is required to perform it. The Prover system has been developed to overcome this constraint by automatically translating iHDL code into SMV. Unfortunately, the exponential behavior of BDDs prevents manual intervention from being completely eliminated. Prover, however, offers several ways to improve the final code, reducing manual intervention significantly, and simplifying the verification process. Using this approach we have verified the P6 integer division circuit.

Integer division on the P6 uses a non-restoring radix-2 algorithm. At each iteration one quotient bit is generated based on the signs of the divisor *d* and partial remainder *R*. *R* is then multiplied by 2 and the divisor added (or subtracted) to it to generate the next partial remainder *R´*. The quotient bit is shifted into the LSB of *R´*. After 32 iterations the final remainder occupies the upper half of the register *R*, while the quotient *Q* occupies the lower half. A last step is needed. In some cases the quotient and the remainder have to be fixed in the following way: The quotient is fixed by adding one, the remainder by adding (or subtracting) the divisor. More information can be found in the P6 MAS.



Automatic Generation of SMV Code

In order to verify this circuit, we have translated the iHDL code using Prover. A direct translation does not work, however, because the circuit is too complex. Prover's pruning options were used to avoid this problem. These options let the user identify the signals that are relevant to the properties being verified. Prover only generates the code for the signals that are in the fan-in cone of those identified. Moreover, prover also lets the user set specific signals to constant values, simplifying even further the SMV code.

The resulting code was considerably more efficient than the previous one, but not efficient enough to allow the complete verification. Manual changes were needed, such as retiming parts of the circuit and removing lines of source code that were not relevant to the verification. The main change, however, consisted of breaking the model into two separate steps, the loop and the remainder/quotient fixes.

Properties Verified

To verify that each iteration of the circuit is correct, we must check the following invariants:

- Loop invariant: $2R \pm d = R'$ (*d* is added or subtracted depending on the quotient bit)
- Safety condition: $-d \leq R \leq d$

We have proven that any values of *d* and *R* that satisfy the safety condition also satisfy the loop invariant. By showing that the initial values of *d* and *R* satisfy that inequality we conclude that each iteration is correct. We can restrict the search to only one iteration, since the results are valid for *any* values of *d* and *R*. We have checked the property:

```
AG((-d <= R <= d) -> (2*R +[-] d = next(R)))
```

To verify that the remainder and quotient fixes are correct we have checked that if no fixes are needed, the output is the value computed in the last iteration. Otherwise, if a quotient fix is needed, the output is $Q + 1$, and if a remainder fix is needed, the output is $R \pm d$:

```
AG (!quofix -> (Q = oldQ))
AG (quofix -> (Q = oldQ + 1))
AG (remfix -> (R = oldR +[-] d))
AG (!remfix -> (R = oldR))
```

We found an error in each case. Both values of *R* and *Q* are stored in the same register, *R* in the upper half, and *Q* in the lower half. However, when the last value of the quotient is FFFFFFFFh, adding one to it will cause a carry into the upper half, adding one to R. This error was known to the designers, and it was reproduced in our model.

Another problem occurs when the last value of the quotient is 7FFFFFFFh. In that case, fixing the quotient changes its sign, and the output is *not* the previous value plus one. However, this is the expected behavior, and the specification of what the quotient fix phase does is not accurate.

Prover has been extremely useful in the verification of the IDIV circuit. The use of tools such as Prover allows formal verification to be performed by non-experts, significantly increasing the range of applications of the method. However, some aspects of the verification, such as retiming and breaking the model into two steps in our example, cannot be easily automated. This indicates that while formal verification can be done systematically and efficiently, it is still not a completely automatic process. Current research aims at reducing manual intervention to a minimum.

Conclusions

The method described in this paper is the only one known to the authors that is able to formally verify arithmetic circuits with wide data paths. Using this method we have been able to reproduce the Pentium® FDIV as well as the P6 IDIV errors. In both cases we have also been able to verify the corrected circuit. We are currently verifying the P6 IMUL and FADD circuits and plan to verify most of the units in the P6 execution cluster.

Our method is a breakthrough in the verification of arithmetic circuits, opening several possibilities for future applications. It is now possible to formally verify circuits that were beyond the scope of formal methods. This can help creating more reliable circuits and makes the design process more efficient. We believe that this method can be of great use in the design of future generations of products.

Acknowledgments

The authors would like to thank Shtadler Ze'ev and Fix Limor for several invaluable discussions on how to use Prover in the most efficient way.

References

[1] Verification of Arithmetic Functions with Binary Moment Diagrams, R. Bryant and Y. Chen, tech. report CMU-CS-94-160, Carnegie Mellon University, May 1994.

[2] Statistical Analysis of Floating Point Flaw in the Pentium® Processor(1994), H.P. Sharangpani, M.L. Barton, Intel Corporation, November, 1994.

[3] Symbolic Model Checking: An Approach to the State Explosion Problem, K. McMillan, Ph.D. Thesis, Carnegie Mellon University, 1992.

[4] Formally Verifying Bus Designs. S. Campos, M. Khaira, X. Zhao. Intel Design Technology Conference, 1994.

[5] Prover EPS. Y. Kimchi, Z. Shtadler, S. Rotem and K. Pines. Intel Corporation, November, 1992.