Bit-Level Analysis of an SRT Divider Circuit

Randal E. Bryant

April 18, 1995
CMU-CS-95-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

It is impractical to verify multiplier or divider circuits entirely at the bit-level using ordered Binary Decision Diagrams (BDDs), because the BDD representations for these functions grow exponentially with the word size. It is possible, however, to analyze individual stages of these circuits using BDDs. Such analysis can be helpful when implementing complex arithmetic algorithms. As a demonstration, we show that Intel could have used BDDs to detect erroneous table entries in the Pentium floating point divider. Going beyond verification, we show that bit-level analysis can be used to generate a correct version of the table.

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: Formal verification, binary decision diagrams, SRT division, Intel Pentium, circuit redesign

1. Introduction

Arithmetic circuits have received relatively little attention from the verification community, except by those using methods based on theorem proving, e.g., [12]. This inattention is due to two main reasons. First, many perceive that arithmetic circuit design is fairly straightforward—the same implementation techniques have been used for years, and designers are confident of their ability to detect errors using conventional simulation. Intel's recent experience with its Pentium floating point divider [11] has exposed the error in this thinking. There are many places one can make mistakes in designing these circuits, some of which may be very hard to detect with the limited number of cases that can be tested by simulation. Second, these circuits are particularly troublesome for methods based on ordered Binary Decision Diagrams (BDDs), the most popular alternative to theorem proving [4]. The BDDs representing the outputs of a multiplier grow exponentially with the word size [3], making them impractical for word sizes much beyond 16 bits. Other arithmetic functions, such as division, also seem to be intractable using BDDs, although this has not been proved formally.

In this paper, we demonstrate that BDD-based verification can be usefully applied to complex arithmetic circuits. Even though it is not feasible to verify the overall circuit functionality, just verifying one iteration can uncover many possible design errors. We demonstrate this by showing the desired behavior for one iteration of radix-4 SRT division [1], as used in the Pentium divider, can be specified and verified using BDDs. This verification will detect incorrect table entries such as occurred in the Pentium, as well as other potentially subtle design errors. Going on beyond verification, we show that a correct PD table can be generated automatically. Our method extends the correction method described by Madre and Coudert [10] to handle logic blocks with larger numbers of inputs and outputs.

2. Binary Decision Diagrams

BDDs serve as a data structure for representing and manipulating Boolean functions. For reading this paper, it is not necessary to understand much about the actual data structure, rather just some general properties. A recent survey paper [4] covers the subject in greater detail. In this paper, we consider only the class of BDDs known as *Ordered* BDDs.

Binary Decision Diagrams represent Boolean functions as directed, acyclic graphs, with nonterminal vertices labeled by variables, and terminal vertices labeled as either 0 or 1. Each nonterminal vertex has two outgoing branches, corresponding to the possible values of the variable. Given a valuation of the variables, the function is evaluated by traversing the graph from the root to the leaf, following the appropriate outgoing edge from each vertex. In an *ordered* diagram, the variables along every path from the root to a leaf must occur in an order consistent with some total ordering of the variables. As an example, Fig. 1 shows a BDD representation of the function given by the Boolean expression $(x_1 + x_2) \cdot x_3$. In this figure, the arcs below each vertex correspond to the cases where the variable is 0 (dashed) or 1 (solid).

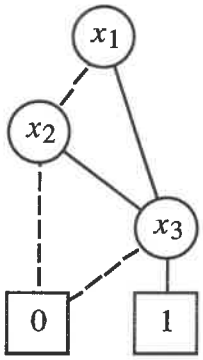

Figure 1: BDD representation of function $(x_1 + x_2) \cdot x_3$. A dashed (solid) branch denotes the case where the decision variable is 0 (1).

Ordered BDDs have the three principal advantages over other representations of Boolean functions. First, by applying a simple set of reduction rules we can reduce a graph to a canonical form. That is, for a given variable order any Boolean function has a unique (up to isomorphism) representation in reduced form. Second, the representations of many functions encountered in digital design are quite compact. All of the functions in the analysis reported here have BDD representations that grow linearly with the data word size.

One key operation for Boolean functions is implemented by the APPLY algorithm. This operation is given as arguments two Boolean functions (represented by two BDDs having consistent variable orderings), plus a binary Boolean operator (e.g., AND, EXCLUSIVE-OR). The APPLY algorithm then generates an OBDD representation of the function obtained by applying the operation to the argument functions. With this algorithm, BDDs representing the Boolean functions computed by a logic gate network can be generated by a form of symbolic evaluation. We start with BDDs representing the inputs as single variables. Proceeding through the network, we generate a BDD representation of each gate output by symbolically applying the gate operator to the BDD representations of the gate inputs.

The size of the BDD representation for a function can depend heavily on the variable ordered used. A number of techniques have been devised to find a suitable variable ordering, ranging from heuristics based on the topology of the gate-level network, to schemes that dynamically reorder the variables as the BDDs are being constructed and manipulated. In the experiments described here, we selected the orderings manually.

3. Bit-Level Analysis

Bit-level analysis of a logic circuit involves generating Boolean function representations of the signal values in terms of variables representing the primary inputs and possibly the state. BDDs have proved the most successful data structure for representing such functions due to their compact size and the ease with which they can be compared and manipulated. These functions can be generated directly from a low-level representation of the circuit, for example from a logic gate description.

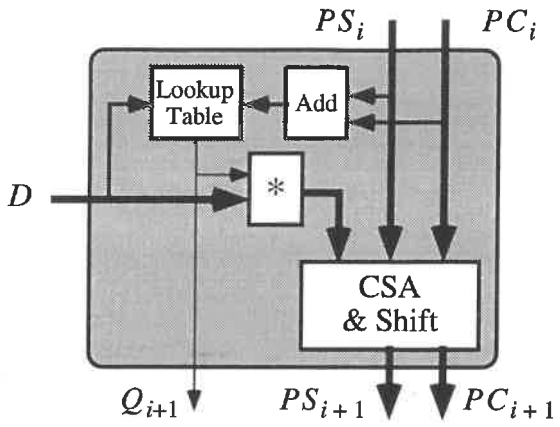

Figure 2: Block level representation of SRT divider stage

To perform functional verification at the bit level, we must also generate Boolean function representations of the desired behavior. For arithmetic circuits, this is not the ideal level of specification. One method is to construct a "known good" implementation of the desired behavior for comparison against the actual design. The utility of the verification then depends on how reliably such an implementation can be generated. For common functions such as addition or multiplication, this is not a difficult task, but for functions such as radix conversion, floating point arithmetic, or division, generating the specification becomes more problematic. An alternative is to generate "checker circuits" that will determine whether the actual circuit's inputs and outputs satisfy the desired arithmetic properties. Generating checker circuits is also somewhat tedious and prone to error, but we claim that the effort can still be worthwhile. It allows us to test the circuit operation over all possible input and state combinations, rather than the limited cases that can be tested with conventional simulation.

4. Verification of SRT Division

As an illustration of bit-level verification, consider the task of verifying a divider similar to that used in the Intel Pentium floating point unit. Although Intel has only divulged limited information about this circuit [11], Tim Coe has created a software model that matches Intel's description and that reproduces its erroneous behavior [6]. Our circuit design is a gate-level implementation derived from Coe's model. Although the actual Pentium divider undoubtably differs from ours in its details, the same verification methods should apply.

The Pentium divider uses an iterative method, having as state a partial remainder, initialized to the dividend, and a partial quotient, initialized to 0. Each iteration extracts two bits worth of quotient, subtracts the correspondingly weighted value of the divisor from the partial remainder, and shifts the partial remainder left by 2 positions. The logic implementing one iteration is shown in Fig. 2. This "stage" has as inputs the divisor D, and the partial remainder, encoded as a pair of words PS_i and PC_i. The actual partial remainder is given by the sum of these two words. It has as outputs the extracted quotient digit Q_{i+1} (ranging from -2 to $+2$), and the updated partial remainder words PS_{i+1} and PC_{i+1}. The subtrac-

tion of the weighted divisor is performed with a carry-save adder (CSA), avoiding the need for propagation through a carry chain. Our implementation of the stage uses a 70-bit word size, enough for extended precision floating point arithmetic, and contains around 1100 logic gates. The PD table, used to look up quotient digits based on truncated values of the divisor and partial remainder, was created from a PLA description generated by the ESPRESSO logic optimizer and then translated automatically into a gate-level equivalent.

A specification for one iteration of the divider can be expressed readily, using the formulation by Atkins [1], modified for the particular numeric format. In our circuit, divisor D is always positive, with a leading 1 to the left of the binary point, while partial remainder words PS_i and PC_i are in two's complement form, with 3 bits, plus the sign bit to the left of the binary point. D can therefore be a number in the range 1.0 to nearly 2.0, while PC_i and PS_i can range from -8.0 to (nearly) 8.0. For valid operation, we impose a "range" constraint on the relative values of the divisor and partial remainder at each step, expressed as a predicate *Range*:

$$\begin{aligned} &\mathit{Range}(D, PS, PC) \;\equiv \\ &\quad -8D \;\leq\; 3(PS + PC) \;\leq\; 8D \end{aligned} \tag{1}$$

We also require the updated partial remainder to be the result of subtracting the weighted divisor and shifting by two, expressed as a predicate *Value*:

$$\begin{aligned} &\mathit{Value}(D, Q, PS, PC, PS', PC') \;\equiv \\ &\quad PS' + PC' \;=\; 4(PS + PC - Q\dot{D}) \end{aligned} \tag{2}$$

Using these predicates, the specification for one iteration can be written as a predicate *Stage*:

$$\begin{aligned} &\mathit{Stage}(D, Q, PS, PC, PS', PC') \;\equiv \\ &\quad \mathit{Range}(D, PS_i, PC_i) \Rightarrow \\ &\qquad [\mathit{Range}(D, PS_{i+1}, PC_{i+1}) \wedge \mathit{Value}(D, Q_{i+1}, PS_i, PC_i, PS_{i+1}, PC_{i+1})] \end{aligned} \tag{3}$$

This specification states that for all legal stage inputs (i.e., satisfying the range constraint) the stage outputs also satisfy the range constraint, and the inputs and outputs are properly related. This specification is reasonably high level and captures the essence of the algorithm.

To check this specification at the bit-level, we must translate the arithmetic expressions, inequalities, and logical connectives into Boolean operations. We did this by constructing a gate-level "checker" circuit consisting of such components as adders, shifters, comparators, and complementers. The checker circuit is actually much larger than the circuit it is checking, requiring a total of around 4300 logic gates. On the other hand, it uses a more routine and conservative design style. In any case, we gain the benefit of comparing two independent, and very different, logic designs.

The verification succeeds for a correct PD table (using entries matching those used by Coe's program) but fails when the five entries described by Intel [11] are changed from 2 to 0. Running on a SUN Microsystems Sparcstation 10, the verification (good or bad) requires around 10 minutes of CPU time and 112 Megabytes of storage, generating BDDs totalling 4.2 million nodes. This performance could most likely be improved with a better variable ordering.

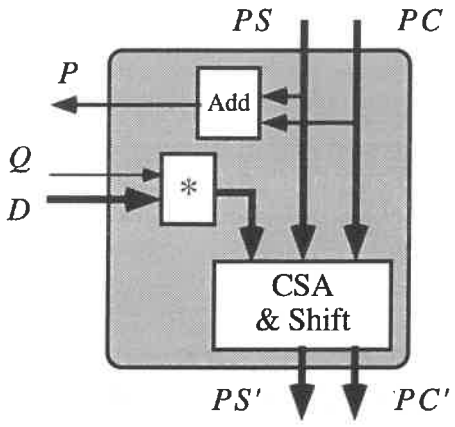

Figure 3: Divider stage modified for generation of PD table

5. Automatic Generation of PD Table

Bit-level circuit analysis can go beyond verifying a design to actually suggest corrections or even be used in the initial synthesis. We illustrate this for the divider circuit by showing how to generate the PD table from the specification given in (3). Our approach generalizes on the method implemented by Madre and Coudert[10]. In their program they replaced a small, k-input, single output portion of the circuit by a "universal logic block" having 2^k auxiliary Boolean variables to encode all possible Boolean functions that could be realized by the block. This technique would not be practical for the PD table, since the block has 11 inputs and 2 outputs.[1] With our method, we express the allowed behavior of a k-input, n-output block as a Boolean function over $k+n$ Boolean variables. This function yields 1 for the allowed combinations of input and output, and 0 otherwise. In effect, we are describing the table as a Boolean relation [13]. Fujita, *et al* have shown that Boolean unification can also be used for redesigning circuits [7]. It does not seem possible to use unification for PD table generation, since there is no way to express the constraint that the table circuit depend on only a subset of the circuit inputs.

To generate the relation for the PD table, we create a modified stage circuit, shown in Fig. 3. This circuit has the lookup table portion removed. The table inputs are turned into stage outputs P (the result of adding the high order 7 bits of PS_i and PC_i, and the quotient digit is made into a stage input Q. The original table also has the high order bits of the divisor as inputs, and therefore we partition D into its high order bits DT (4 bits, omitting the bit guaranteed to be 1), and its low order bits DL. The set of variables encoding D is then written $\langle DT, DL \rangle$. The modified stage has outputs P, PS', and PC', which depend on inputs DT, DL, Q, PS, and PC.

Our goal is to generate a specification of the table, indicating for each input, the allowable values of the quotient digit. This is expressed as a "characteristic function" $\tau(DT, PT, Q)$,

[1] We need only generate the magnitude of the quotient digit. The sign can be determined from that of the truncated partial remainder.

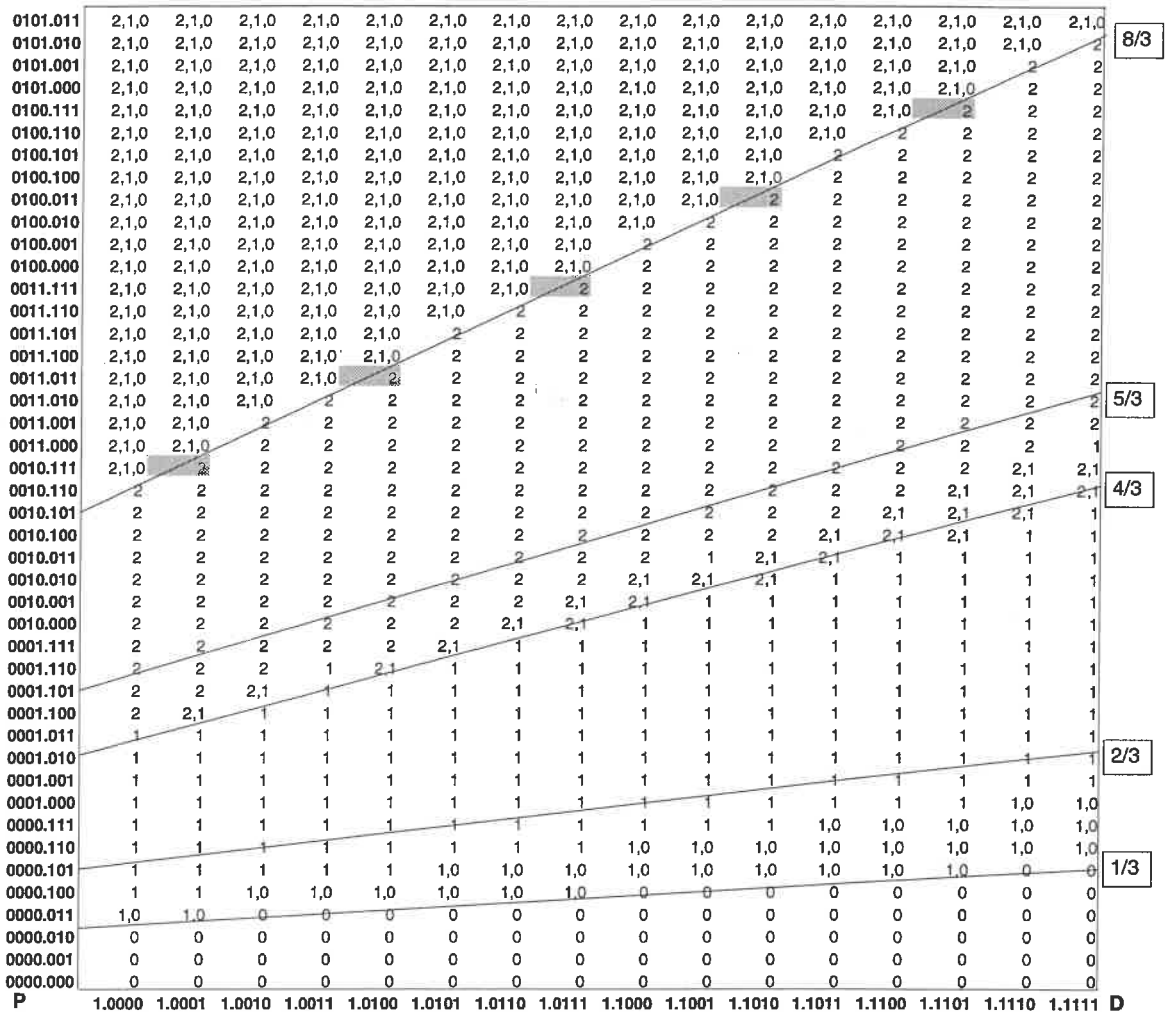

Figure 4: Partial PD table for radix-4 SRT divider generated by bit-level analysis

where DT and PT are sets of Boolean variables (4 and 7, respectively) encoding the table inputs, and Q is a set of Boolean variables encoding the table outputs. The function τ yields 1 for every allowable combination of table input and output. This function can be expressed as:

$$\tau(DT, PT, Q) \equiv \forall DL, PS, PC\,[(P = PT) \Rightarrow \mathit{Stage}(\langle DT, DL\rangle, Q, PS, PC, PS', PC')] \quad (4)$$

The universal quantification in (4) states that the table correctly handle all combinations of partial remainder and divisor, based only on the information encoded in the table inputs PT and DT. Our BDD package implements both universal and existential quantification.

Computation of the function τ required around 12.5 minutes of CPU time, generating a maximum of 4.5 million BDD nodes occupying 132 Megabytes of storage. The results are shown in Fig. 4 for positive values of P. For each value of the truncated divisor (labeled **D**) and each nonnegative value of the truncated partial remainder (labeled **P**), the allowable quotient digit values are shown. To save space, rows for values of P beyond 0101.011_2 are

omitted, since all of their entries are "don't cares," i.e., they are labeled "2,1,0." A similar, although not fully symmetric version of the table was generated for negative values of P.

Although there are really no surprises in this table, several features are worth highlighting. First, we have annotated the table with lines of different slope, as is traditionally shown with PD plots [1]. The only slope value that appears in the specification is the value 8/3, arising from the inequality of (1). The other values were, in effect, determined automatically through the generation of τ. Second, the shaded regions indicate the cases where Intel had erroneous values of 0. As our plot shows, the only allowable value for these entries is 2. Finally, note that every table entry has at least one value. If there were any entries indicating no allowable values, this would indicate that the table could not be generated. In particular, it would mean that the limited information given the table about the divisor and partial remainder did not suffice to guarantee selection of a valid quotient digit.

The information computed by this analysis could be used to directly synthesis a PLA implementation of the PD table. Watanabe and Brayton [13] have shown that two-level optimization can be performed using are relational specification of the desired functions. This form allows more a more general specification of the Don't Care conditions.

6. Conclusions

This analysis demonstrates the utility of BDD-based techniques even for circuits that cannot be verified in their entirety. For complex algorithms such as SRT, verifying a complete iteration of the algorithm is a significant step in ensuring the correctness of the complete computation. Even though the SRT algorithm has been studied and implemented many times, it is easy to make mistakes. Although Intel's incorrect implementation has received the most publicity, it is worth noting that Goldberg makes the same basic mistake as Intel in his exposition of a PD table for this algorithm [8, Table A.30].[2]

One advantage of BDD-based analysis over other approaches to verification is the ability to work with low-level circuit models. Thus, we could work directly from a PLA format file in constructing our model of the PD table. Although it is unclear exactly how Intel generated their erroneous PLA, working from a low-level representation ensures that we are verifying the circuit as it will be implemented. It would even be possible to work from a mask-level circuit description, using a combination of transistor circuit extraction and symbolic switch-level analysis to generate the circuit model [2].

Generating the checker circuits for this analysis involved a significant effort, and we have no reliable means of verifying these checker circuits. Viewed in economic terms, the need to generate checker circuits both increases the cost and reduces the potential benefit of performing formal verification. Nonetheless, as the cost of making mistakes grows, it becomes easier to justify an increased investment in formal verification. An investment of around 1 person-month of effort, costing no more than $20,000, would be more than sufficient

[2]This mistake was pointed out to me by Tim Coe.

for the analysis described here. Such an expense is trifling compared to the $475 million charge Intel has taken on account of the Pentium error. Of course, it is easy to identify an error once its location is known, but we claim that our analysis would cover many possible sources of design error in the divider circuit. Getting one iteration right is a key step in implementing this algorithm.

Over the long term, it would be preferable to have a system whereby users could express their specifications using a combination of arithmetic expressions, predicates such as inequalities, and Boolean connectives, in the manner of (1–4). This would indeed be possible using word-level expressions as part of the specification, as formulated by Lai and Vrudhula [9]. These expressions can be generated and manipulated as "pseudo-Boolean" functions mapping Boolean variables to numeric values. Such functions can be represented as edge-valued BDDs [9], or as Binary Moment Diagrams [5]. Having such a capability would both decrease the cost and increase the benefit of formal verification for arithmetic circuits.

References

[1] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainder," *IEEE Transactions on Computers*, Vol. C-17, No. 10 (October, 1968), pp. 925–934.

[2] R. E. Bryant. "Boolean Analysis of MOS Circuits," *IEEE Transactions on CAD/IC*, Vol. CAD-6, No. 4 (July, 1987), pp. 634–649.

[3] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.

[4] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293–318.

[5] R. E. Bryant, and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," *32nd Design Automation Conference*, 1995.

[6] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs Journal*, Vol. 20, No. 4 (April, 1995) pp. 129–135.

[7] M. Fujita, Y. Tamiya, Y. Kukimoto, and K.-C. Chen, "Application of Boolean Unification to Combinational Logic Synthesis," *International Conference on Computer-Aided Design*, 1991, pp. 510–513.

[8] D. Goldberg, "Computer Arithmetic," Appendix A of D. A. Patterson, J. L. Hennessy, *Computer architecture: a quantitative approach*, Morgan Kaufmann Publishers.

[9] Y.-T. Lai, and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," *29th Design Automation Conference*, June, 1992, pp. 608–613.

[10] J.-C. Madre, and O. Coudert, "Automating the diagnosis and rectification of design errors in PRIAM," *International Conference on Computer-Aided Design*, 1989, pp. 30–33.

[11] H. P. Sharangpani, M. L. Barton, "Statistical analysis of floating point flaw in the Pentium processor(1994)," Intel Technical Report, Nov. 30, 1994.

[12] D. Verkest, L. Claesen, and H. DeMan, "A proof of the nonrestoring division algorithm and its implementation on an ALU," *Formal Methods in System Design*, Vol. 4, No. 1 (January, 1994), pp. 5–32.

[13] Y. Watanabe, and R. K. Brayton, "Heuristic minimization of multiple-valued relations," *IEEE Transactions on CAD/IC*, Vol. 12, No. 10 (October, 1993), pp. 1458–1472.