Bit-Level Analysis of an SRT Divider Circuit[*]

Randal E. Bryant
Carnegie Mellon University
Pittsburgh, PA 15213
Randy.Bryant@cs.cmu.edu
http://www.cs.cmu/~bryant

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 lookup table entries in the Pentium(TM) floating point divider. Going beyond verification, we show that bit-level analysis can be used to generate a correct version of the table.

1. Introduction

Arithmetic circuits have received relatively little attention from the verification community, except by those using methods based on theorem proving, e.g., [14]. 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 [13] 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 especially challenging for methods based on ordered Binary Decision Diagrams (BDDs), the most popular alternative to theorem proving [5]. The BDDs representing the outputs of a multiplier grow exponentially with the word size [4], making them impractical for word sizes much beyond 16 bits. A similar result has been shown for representing the outputs of a divider [16]. On the other hand, the outputs of simpler units such as adders, subtractors, and comparators are represented very efficiently with BDDs.

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 entries in the "PD" table, used to generate a quotient digit on each division step, 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 [12] to handle logic blocks with larger numbers of inputs and outputs.

The intention of this paper is not to advance the state-of-the-art in formal verification, but rather to illustrate how existing technology could be applied to real life designs. Current tools fall well short of the ultimate goal of verifying complete floating point hardware designs against a high level, mathematical specification, e.g., the IEEE Floating Point Standard. Nonetheless, they can be applied to key components of a system, enhancing overall design quality. The decision of which components to verify requires a weighing of three factors: 1) how cleanly the functionality of the component can be specified, 2) whether the subsystem is tractable for existing formal verification tools, and 3) the chances that the subsystem may contain subtle design errors that cannot be detected by less formal measures, such as simulation. Although this "opportunistic verification" does not guarantee complete system correctness, any measures that can reduce the chances of design errors are worthy of consideration. The decision of whether and where to apply such verification should be based largely on economic grounds. That is, the cost of formally specifying and verifying a component should be compared to the cost of other verification methods, such as simulation, as well as to the cost of any design errors that could be missed by these less comprehensive approaches.

The Pentium FDIV problem provides a clear illustration for the potential value of opportunistic verification. Compared to the $475 million charge that Intel took against its 1994 revenues to replace defective Pentium chips, one can easily

[*] 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.

33rd Design Automation Conference ®

justify applying verification tools to a number of subsystems, including many parts of the floating point unit.

2. Bit-Level Analysis

A 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.

For the purpose of this presentation, the details of the BDD data structure and algorithms are not important. Instead, we will simply present some of the important characteristics. The survey article in [5] provides more information for the interested reader. A BDD represents a Boolean function as a directed, acyclic graph with vertices corresponding to the function variables. In an *Ordered* BDD, these variables are assigned an ordering, and all vertices must follow this ordering in the graph. By a simple set of transformation rules, such a BDD may be reduced to a canonical form, i.e., such that the representation of a given function is unique. Thus, if two circuits compute the same logic function, their BDD representations will be identical, even if they differ greatly in their structure. For example, BDD-based analysis can readily determine that a carry-lookahead adder is functionally equivalent to a carry-ripple one.

The BDD representation of a combinational logic circuit can be generated by evaluating its gate-level representation. Starting with single vertex BDDs representing variables for the primary inputs, each gate is evaluated to generate the BDD representation of its output based on the BDD representations of its inputs. For modeling sequential circuits, a form of "symbolic simulation" can be used, augmenting conventional 3-valued event-driven simulation with a BDD-based symbolic manipulator to generate representations of the node states as functions of Boolean variables assigned to the circuit inputs and state [7]. Although there are no commercial products providing this capability, prototype tools developed in universities have been available for over 5 years[2]. In this study, we used such a tool in addition to program consisting of a command line interface to a BDD package.

To perform functional verification at the bit level, we must also generate Boolean function representations of the desired behavior. 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. Furthermore, it would be quite reasonable to create a library of standard arithmetic functions (addition, multiplication, relational operations, etc.), and automatically generate checker circuits from a more abstract specification.

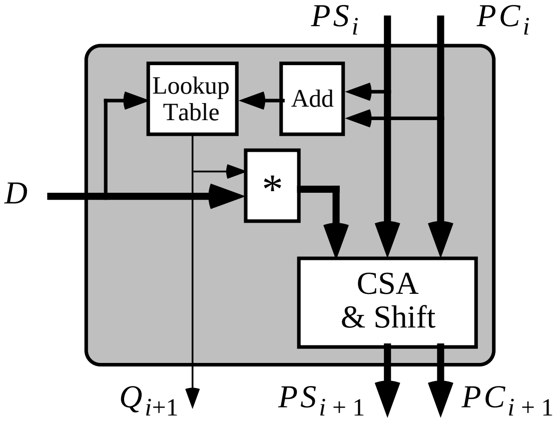


Figure 1: Block level representation of SRT divider stage

3. 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. We focus only on the part of the circuit computing the quotient of the two mantissas, as this is where the FDIV error occurrred. The exponent and rounding calculations are not considered. Although Intel has only divulged limited information about this circuit [13], Tim Coe has created a software model that matches Intel's description and that reproduces its erroneous behavior [9]. 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 the iterative "SRT" algorithm, named after the initials of the 3 people credited with its discovery. Each iteration of the algorithm generates one digit of the quotient. A key feature of this algorithm is that by using a redundant number representation, each quotient digit can be computed using only estimates of the partial remainder and divisor. This allows fast iteration even for large word sizes. The particular version of the SRT algorithm implemented by the Pentium operates with radix 4 quotient digits, so that two bits of the quotient are computed in each iteration.

The divider has 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 bit positions. The logic implementing one iteration is shown in Fig. 1. 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 subtraction 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. A 7-bit adder is used to add the high order bits of the partial remainder words as an estimate of the true partial remainder. The high 5 bits (one of which is always 1) of the divisor are used as an estimate of the true divisor value. These estimates are used to index into a lookup table, known as the "PD" table to generate the quotient digit. In our implementation, the table 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 $Range$:

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

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

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

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

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

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 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 design style and it allows a comparison between two independent, and very different, logic designs. Furthermore, the checker circuit can be designed in a simpler and more conservative style. 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 [13] 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.

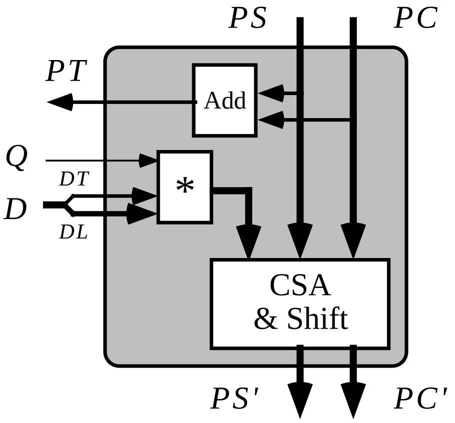


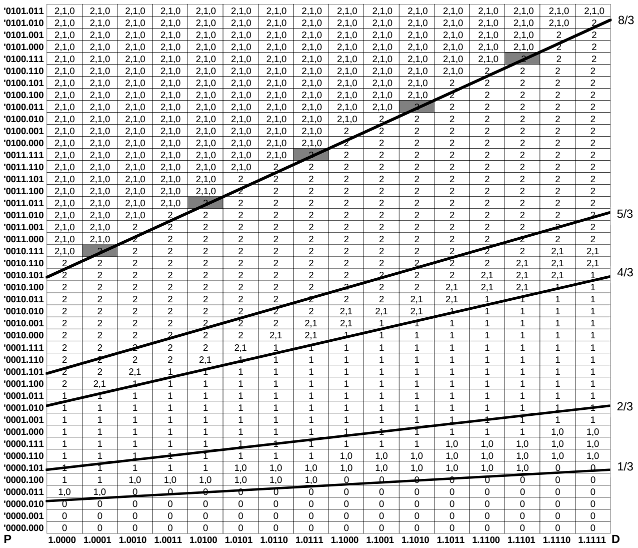
Figure 2: Divider stage modified for generation of PD table

4. 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 analysis technique generalizes on the method implemented by Madre and Coudert[12]. 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 [15].

To generate the relation for the PD table, we create a modified stage circuit, shown in Fig. 2. This circuit has the lookup table portion removed. The table inputs are turned into stage outputs PT (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

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


8/3
5/3
4/3
2/3
1/3
D

P	1.0000	1.0001	1.0010	1.0011	1.0100	1.0101	1.0110	1.0111	1.1000	1.1001	1.1010	1.1011	1.1100	1.1101	1.1110	1.1111
'0101.011	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0
'0101.010	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2
'0101.001	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2
'0101.000	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2
'0100.111	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2
'0100.110	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2
'0100.101	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2
'0100.100	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2
'0100.011	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2
'0100.010	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2
'0100.001	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2
'0100.000	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2
'0011.111	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2
'0011.110	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2
'0011.101	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2
'0011.100	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2
'0011.011	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2
'0011.010	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2
'0011.001	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
'0011.000	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
'0010.111	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1
'0010.110	2	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1
'0010.101	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1
'0010.100	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1	1
'0010.011	2	2	2	2	2	2	2	2	2	2	2,1	2,1	1	1	1	1
'0010.010	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1	1	1	1	1
'0010.001	2	2	2	2	2	2	2	2,1	2,1	1	1	1	1	1	1	1
'0010.000	2	2	2	2	2	2	2,1	2,1	1	1	1	1	1	1	1	1
'0001.111	2	2	2	2	2	2,1	1	1	1	1	1	1	1	1	1	1
'0001.110	2	2	2	2	2,1	1	1	1	1	1	1	1	1	1	1	1
'0001.101	2	2	2,1	1	1	1	1	1	1	1	1	1	1	1	1	1
'0001.100	2	2,1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'0001.011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'0001.010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'0001.001	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'0001.000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1,0	1,0
'0000.111	1	1	1	1	1	1	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0
'0000.110	1	1	1	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
'0000.101	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0	0
'0000.100	1	1	1,0	1,0	1,0	1,0	1,0	1,0	0	0	0	0	0	0	0	0
'0000.011	1,0	1,0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'0000.010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'0000.001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'0000.000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3: Positive PD table for radix-4 SRT divider generated by bit-level analysis

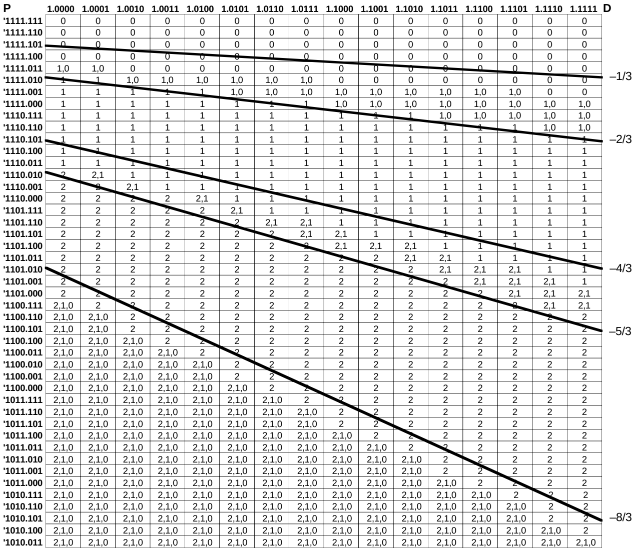
order bits DL. The set of variables encoding D is then written $\langle DT, DL \rangle$. The modified stage has outputs PT, PS', and PC', which depend on inputs DT, DL, Q, PS, and PC.

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" $\tau(DT, PT, Q)$, 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. Function τ yields 1 for every allowable combination of table input and output. For a given value of DT and PT, τ may yield 1 for no values of Q, indicating the problem is overconstrained, and the table cannot be constructed. Alternatively, it may yield 1 for either a single value of Q, indicating a unique choice of quotient digit, or for multiple values of Q, indicating more than one allowable digit. The τ function can be expressed as:

$$\begin{aligned} \tau(DT, PT, Q) \;&\equiv \\ &\forall DL, PS, PC[(P \Leftrightarrow PT) \Rightarrow \\ &\quad Stage(\langle DT, DL \rangle, Q, PS, PC, PS', PC')] \end{aligned} \tag{4}$$

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. Such quantification is implemented by our BDD package.

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. 3 for positive values of PT. 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 PT 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 PT, as is shown in Figure 4. For this table, the actual quotient digit values should be the negative of those shown.


D
−1/3
−2/3
−4/3
−5/3
−8/3

P	1.0000	1.0001	1.0010	1.0011	1.0100	1.0101	1.0110	1.0111	1.1000	1.1001	1.1010	1.1011	1.1100	1.1101	1.1110	1.1111
'1111.111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'1111.110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'1111.101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'1111.100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'1111.011	1,0	1,0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'1111.010	1	1	1,0	1,0	1,0	1,0	1,0	1,0	0	0	0	0	0	0	0	0
'1111.001	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0	0
'1111.000	1	1	1	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
'1110.111	1	1	1	1	1	1	1	1	1	1	1	1,0	1,0	1,0	1,0	1,0
'1110.110	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1,0	1,0
'1110.101	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'1110.100	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'1110.011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'1110.010	2	2,1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
'1110.001	2	2	2,1	1	1	1	1	1	1	1	1	1	1	1	1	1
'1110.000	2	2	2	2	2,1	1	1	1	1	1	1	1	1	1	1	1
'1101.111	2	2	2	2	2	2,1	1	1	1	1	1	1	1	1	1	1
'1101.110	2	2	2	2	2	2	2,1	2,1	1	1	1	1	1	1	1	1
'1101.101	2	2	2	2	2	2	2	2,1	2,1	1	1	1	1	1	1	1
'1101.100	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1	1	1	1	1
'1101.011	2	2	2	2	2	2	2	2	2	2	2,1	2,1	1	1	1	1
'1101.010	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1	1
'1101.001	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1	1
'1101.000	2	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1	2,1
'1100.111	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2,1	2,1
'1100.110	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
'1100.101	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
'1100.100	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2	2
'1100.011	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2	2
'1100.010	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2
'1100.001	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2	2
'1100.000	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2	2
'1011.111	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2	2
'1011.110	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2
'1011.101	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2	2
'1011.100	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2	2
'1011.011	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2	2
'1011.010	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2
'1011.001	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2	2
'1011.000	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2	2
'1010.111	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2	2
'1010.110	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2
'1010.101	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2	2
'1010.100	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2
'1010.011	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0	2,1,0

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

Although there are really no surprises in these tables, several features are worth highlighting. First, we have annotated the tables with lines of different slope, as is traditionally shown with PD plots [1]. The only slope value that appears in the specification are the values $\pm 8/3$, 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 synthesize a PLA implementation of the PD table. Watanabe and Brayton [15] 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.

5. 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 in his derivation of a PD table for this algorithm, Goldberg makes the same basic mistake as Intel, failing to recognize that some table entries are reachable even when the estimated values PT

and DT appear out of range.[10, 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 at exactly what design stage Intel's design error occurred, working from a low-level representation ensures that we are verifying the circuit as it will actually 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 [3].

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 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 [11]. 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 [11], or as Binary Moment Diagrams [6]. Recent work has shown that word-level specifications can be added to symbolic model checking [8]. 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] D. Beatty, K. Brace, R. E. Bryant, K. Cho, and L. Huang, "User's guide to COSMOS, a compiled simulator for MOS circuits," Version 3.0, Carnegie Mellon University, August, 1990.

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

[4] 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.

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

[6] R. E. Bryant, and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," *32nd Design Automation Conference*, 1995, pp. 535–541.

[7] K. Cho, and R. E. Bryant, "Test pattern generation for sequential MOS circuits by symbolic fault simulation," *26th Design Automation Conference*, June, 1989, pp. 418–423.

[8] E. M. Clarke, and X. Zhao, "Word level symbolic model checking: a new approach for verifying arithmetic circuits," *33rd Design Automation Conference*, 1996.

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

[10] D. Goldberg, "Computer arithmetic," Appendix A of D. A. Patterson, J. L. Hennessy, *Computer architecture: a quantitative approach*, First Edition, Morgan Kaufmann Publishers, 1990.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] I. Wegener, "Optimal lower bounds on the depth of polynomial-size threshold circuits for some arithmetic functions," *Information Processing Letters*, Vol. 46, pp. 85–87.

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