Restricted Symbolic Evaluation Is Fast and Useful

J. Lawrence Carter and Barry K. Rosen
IBM Research Division
P.O. Box 218
Yorktown Heights NY 10598

Gordon L. Smith
IBM Data Systems Division
P.O. Box 950
Poughkeepsie NY 12602

Vijay Pitchumani
Syracuse University
ECE Department
Syracuse NY 13244

Full symbolic evaluation of large circuits with many inputs is combinatorially impractical, but restricted forms of symbolic evaluation are enough for some important uses. This paper introduces a new method of simulation with ***two zillion and three values***. The values that are propagated by the simulation include the familiar 0, 1, and X, and also a collection of ***named unknowns*** and their formal negations. Each value fits in a single computer word. Applications of this restricted symbolic evaluation include design rules checking for circuits with embedded arrays and timing verification. This paper explores these two applications briefly. By carefully choosing rules for combining the two zillion and three values, and the represenions of the values, it is possible to make simulation surprisingly efficient. This paper presents two variants and an implementation of each. Both are fast; one is faster but sometimes yields less information.

1. Introduction

Many steps in the design of modern computers involve gate-level simulation [2], where the output lines of logic blocks (such as NAND gates and LATCH primitives) are assigned logic values (0, 1, and perhaps other values) based on the values held by the input lines to the blocks. We assume that cyclic paths in the design are accommodated in the standard ways, such as using event-driven simulation or breaking the cycles at latches and ensuring that no latch that has a clock active is fed by another such latch [4]. Many uses of gate-level simulation involve the simulation of lines whose values are unspecified or not known at the time of simulation. In this paper, we introduce a method for handling unspecified logic values called "simulation with two zillion and three values" (denoted (2Z+3)-valued simulation). Two variants of (2Z+3)-valued simulation are described: one is faster; the other gives more information. We describe two important applications. Finally, we describe implementations that show the techniques are practical.

There are a number of known techniques for handling unspecified logic values in gate-level simulation. At one extreme is 3-valued simulation, where all unknown lines are given the value X. In 3-valued simulation, one knows nothing about the possible relationships between one X and any other. At the other extreme is full symbolic execution [3], where unknown lines are initialized to distinct values, and accurate information is maintained during the simulation about how different computed values relate to each other. The advantage of 3-valued simulation is speed - it requires only a small multiple over the time required for 2-valued simulation. (For example, in HSS [1] the multiple is 2.9.) The advantage of full symbolic simulation is accuracy; however, determining the equivalence of lines in a combinational circuit is an NP-complete problem, and hence one cannot reasonably hope to find an efficient full symbolic evaluation algorithm that can handle circuits with thousands of gates.

Simulation with two zillion and three values lies between these extremes, but falls distinctly within the practical realm. As with full symbolic simulation, lines that are initially unknown are assigned distinct ***named unknowns***, a(1), a(2), and so on. In (2Z+3)-valued simulation, the only symbolic values are the named unknowns and their formal negations, $\neg$a(1), $\neg$a(2), and so on. If the output of a simulated logic block is not determinate (0 or 1) and cannot be expressed as a symbolic value, then it is assigned the value X. Thus, a NAND gate whose input lines have values 1 and a(17) will be given the output value $\neg$a(17), but the NAND of a(17) and a(23) will be X. The name "two zillion and three" reflects the fact that there are a large number of distinct named unknowns, plus their negatives, plus the 0, 1, and X of 3-valued simulation.

Section 2 has the detailed rules for operating with two zillion and three values. We will introduce two variants that differ in how they evaluate certain expressions, such as a(17) AND $\neg$a(17). Section 3 briefly explores applications to array rules checking and to timing verification. For both applications, the number n of named unknowns is large enough that enumerating the 2^n possible assignments represented by the named unknowns is out of the question. Full symbolic evaluation, with complex expressions that treat the symbolic values as variables, is also out of the question.

The remaining sections give implementations of (2Z+3)-valued simulation. The simple variant (Section 4) is faster; the refined variant (Section 5) sometimes yields more information by computing fewer X values. Both avoid the high costs of full symbolic evaluation. The variants share the same representation of values, so the techniques can be intermixed. In this representation, any value is represented by a single computer word, and operations are implemented with fast arithmetic and bitwise operations. In order to represent each value conveniently within a single computer word, the number of named unknowns (called a ***zillion*** here) must be less than a large constant determined by the word size and by the details of the representation. When our representation is used on a computer with 32-bit words, a zillion is $2^{27} - 2$. Applications are likely to require tens, hundreds, or perhaps thousands of named unknowns. The zillion provided here will allow for many millions.

2. Logic Values

Rules for propagating the two zillion and three values through AND and NOT blocks are tabulated in Figure 1. The rules for other Boolean operations (e.g., NAND, NOR, and OR) can be derived from these. The tables also indicate how latches are handled; latches and memory arrays will be discussed later in this section. In the tables, a(i) and a(j) represent any of the named unknowns and $\neg$a(i) and $\neg$a(j) represent their negatives. Intuitively, we propagate a symbolic value through a gate when all the other inputs to the gate take on the noncontrolling value.

In one variant of (2Z+3)-valued simulation, any combination of symbolic values yields X. We will call this variant the "simple rules". With the available values, the simple rules give the only reasonable way to fill in the table when i $\neq$ j, as in $\neg$a(5) AND a(9) = X. However, when i = j, rules like $\neg$a(9) AND a(9) = 0 would sometimes provide more information, since fewer X values would be produced. This insight motivates the "refined rules" explained in Section 5. Figure 2 illustrates the advantages of the refined rules for propagating symbolic values through logic that implements the XOR function with NAND gates.

AND	0	1	X	a(j)	¬a(j)
0	0	0	0	0	0
1	0	1	X	a(j)	¬a(j)
X	0	X	X	X	X
a(i)	0	a(i)	X	X	X
¬a(i)	0	¬a(i)	X	X	X

	NOT	LATCH
0	1	old value
1	0	new data
X	X	X
a(i)	¬a(i)	X
¬a(i)	a(i)	X

Figure 1. Operations on simulated values. The behavior of a latch is described as a function of its governing clock, assuming that 0 means an inactive clock while 1 means an active clock. Dotted boxes indicate where the simple rules (shown here) sometimes yield less information than the refined rules (explained in Section 5).

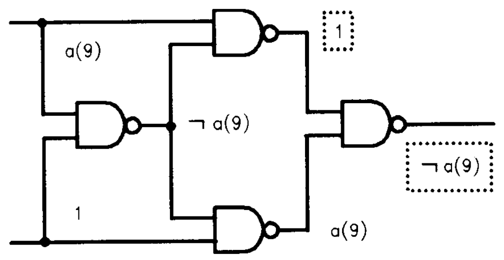

Figure 2. The refined rules for (2Z+3)-valued simulation can propagate a symbolic value through logic implementing the XOR function. The simple rules would compute X for the values shown in boxes.

Latches are handled in a straightforward manner under the simple rules: if the governing clock is inactive, the latch retains its old value; if the clock is active, the latch takes on the value of the data input; and otherwise (when the clock is not known to be 0 or 1), the latch takes the value X. Under the refined rules, a test is made to see if the old value of the latch is the same as the data input. If so, that value is retained no matter what value the clock takes on.

A READ operation of an array can be simulated by first examining the values on the address lines. If each address line value is determinate (either 0 or 1), then the contents of the corresponding array cell are placed on the data output lines. Otherwise, X's are placed on the output lines. (It would be consistent with the spirit of the refined rules to make a further check to see if all the array cells that might match an indeterminate address contained the same value, and, if so, to output that value. However, this test would be very expensive to implement and unlikely to yield improved information.)

To simulate a WRITE operation, again the address lines need to be examined. If each address line value is determinate, then the values on the data lines are written into the appropriate memory cell. Otherwise, we set to X each memory cell whose address matches the determinate address line values. For example, if there are four address lines and their values are 0, X, 1, and a(17), then the array locations 0010, 0011, 0110, and 0111 are set to X. Under the refined rules, an additional check is made to see if the data lines match the array cell contents; if so, then no change is made to the cell.

Section 4 implements the simple rules, which are adequate for verifying many array initialization protocols. Section 5 implements the refined rules. For brevity, both sections consider combinational logic only. Latches and arrays may be implemented by straightforward coding of the foregoing definitions, without using any special properties of the representation.

3. Applications

In many VLSI design environments, there is a design rule to the effect that it must be possible to initialize each array to contain any desired data. It may not be at all obvious whether a particular circuit satisfies the rule, particularly when an array is embedded in the midst of the circuit. Thus, the designer may need to provide an ***array initialization protocol*** that tells how to initialize the array. The rules checking system would then need to verify the correctness of the protocol. The protocol specifies what values are to be applied to primary inputs and latches at each time step of the initialization, as well as the method of pulsing the clocks. The values 0 and 1 are actual logical values that will be applied. Named unknowns represent the arbitrary data (e.g., output of a pseudo-random pattern generator, or actual test patterns) that one hopes will end up in the array. The X's are unspecified by the protocol, and can be used either in situations where the designer does not care what value is applied, or where the value may be changing from a 0 to a 1 or vice versa.

Similarly, if there is a design rule to the effect that it must be possible to output the contents of each array, then one needs to specify an ***array logout protocol***. Again, it may not be obvious that the protocol is correct. It will not be pursued in this paper, but logout protocols can be verified by techniques similar to those for initialization protocols.

These same two requirements - that it must be possible to initialize and to read out memory elements - commonly arise in a different context. If we think of the internal latches of a design as being an array, then the requirements say that it must be possible to independently set and read the latches.

Verifying an Array Initialization Protocol

Given an array initialization protocol, (2Z+3)-valued simulation can be used to verify that the protocol succeeds in initializing the array as desired. This is done as follows:

1. The array is initialized to X's.
2. The protocol is simulated.
3. The array contents are examined. If any cell contains a 0 or a 1, it means that it was not possible to read arbitrary data into that cell, so the protocol fails the rules check. If any cell contains X, the protocol also fails. (This failure mode has a special significance discussed later in this section.) Otherwise, all the values in the array are symbolic values – but a further test should be made that each named unknown appears at most once (in either its positive or negative form). This test ensures that all the bits can be set independently.

Assuming the protocol passes the above tests, one now knows the exact mapping from the test data to the array contents. For example, one might know that array cell 13 holds the data put on the 5-th data line on the 22-nd clock cycle. This information can be used to bypass future simulations of the initialization protocol.

Suppose, on the other hand, that the protocol fails. If the only failures are caused by cells containing X, then it is possible that the protocol is really correct but depends on logical equivalences not captured by our simple propagation rules. One possible course of action would be simply to reject the protocol. Another possible course would be to use a more detailed analysis when the only failures are caused by cells containing X. In this latter case, the use of (2Z+3)-valued simulation first has still accomplished something: the more detailed analysis can be restricted to examining the computations that write an X into a cell of the array.

An Application to Timing Verification

Complex operations sometimes require that a latch hold its value, whatever that value is, throughout several clock cycles. A polarity hold latch with an arbitrary initial value a(1) will continue to hold a(1) if it is clocked with the same value a(1) on the data input. Thus, the requirement could be met by gating the latch's value back to its data input line during the relevant cycles. (This implementation avoids the necessity of inhibiting the clock.) For verifying that timing requirements are met in such a situation, the data in the latch may need to be named but do not need to be known. The refined rules of (2Z+3)-valued simulation can verify that latches containing symbolic values do indeed hold these values for the required number of clock cycles.

If one attempted to verify persistence over several clock cycles using 3-valued simulation, then even if one found that a particular value (0 or 1) remained in the latch for the required duration, one still would not be sure whether this was just a coincidence that relied on the particular setting of other lines in the circuit. Thus, one might need to try all 2^n assignments to the n input lines to achieve the effect of one round of (2Z+3)-valued simulation.

4. Implementation of the Simple Rules

We assume that the target computer has two's-complement arithmetic with 32-bit words. (Once the techniques are understood, it is not hard to adapt them to other architectures.) We represent each value by a word, which can be considered to be a signed integer. Figure 3 gives representations for the values 0, 1, X, and a(i) and ¬a(i) for i ranging from 1 to $2^{27}-2$. There are two alternate representations for the value X.

Value	Representation	
	(as integer)	(as Hex string)
0	0	00000000
1	-1	FFFFFFFF
a(1)	$2^{28}+2$	10000002
¬a(1)	$2^{28}+3$	10000003
...	... etc ...	...
$a(2^{27}-2)$	$2^{29}-4$	1FFFFFFC
$\neg a(2^{27}-2)$	$2^{29}-3$	1FFFFFFD
X	$2^{29}-2$	1FFFFFFE
X (alternate)	$2^{29}-1$	1FFFFFFF

Figure 3. Representations of two zillion and three values.

When viewed as integers in the computer's arithmetic, the representations of all indeterminate values (i.e., X and the symbolic values) lie between $2^{28}+2$ and $2^{29}-1$, inclusive. This range of numbers was carefully chosen so that there is an efficient way to determine whether there is more than one indeterminate value among a set of up to 4 values. Specifically, if the representations of 4 or fewer values are added together, and at most one of the values was indeterminate, then the result will be at most $2^{29}-1$. On the other hand, if 2 or more of the values are indeterminate, then the sum of their representations will be at least $2 \times (2^{28}+2) - 2 = 2^{29}+2$. Thus, a set of 4 or fewer values contains at most one indeterminate value if and only if the sum of the representations is less than 2^{29}.

Implementations of the AND gate with up to four inputs, and of the NOT gate, are shown in Figure 4 and Figure 5 .

```
R := BitwiseAND(U,V,...)
if ( R > 0 ) then /* At least one argument is indeterminate. */
   do S := U+V+...
      if ( S ≥ 2^29 ) then R := Xrep /* Xrep = 2^29 - 2 */
   end
Return ( R )
```

Figure 4. Implementation of simple AND with at most four inputs U,V,... .

```
if ( U > 0 )
then R := BitwiseXOR(U,1) /* exclusive-or */
else R := BitwiseXOR(U, -1)
Return ( R )
```

Figure 5. Implementation of NOT with input U.

5. Implementation of the Refined Rules

The refined rules should be as informative as possible, given that the output value of each gate is to be one of the two zillion and three values. To achieve this goal, a p-input AND gate cannot be represented as a tree of $(p - 1)$ two-input gates. No such tree will make both $\text{AND}(a(5),X,\neg a(5)) = 0$ and $\text{AND}(a(5),\neg a(5),X) = 0$.

This section defines and implements the refined rules for a p-input AND gate. Other logic operations such as OR, NOR, and even XOR are treated similiarly. The truth table for NOT is the same here as in Figure 1, so the implementation is the same also.

Consider any AND gate G. Let L be the list of G's input values. Remove duplicate values from L to produce a set S. If there is a value of i such that S contains both a(i) and ¬a(i), then G's output is 0. Otherwise, G's output is computed by applying the simple rules to S.

A direct implementation of the above definition of the refined AND operation would be slow, but Figure 6 implements this operation efficiently. For clarity, we treat the number p of inputs as a variable and use variables i and j that range up to p in loops. Because the number of values of p used in each technology is small, a separate version with unrolled loops should be compiled for each value of p. (The common special case of two inputs could then get additional improvements.) The procedure of Figure 6 computes AND(...) using at most $p(p + 1)/2$ bitwise operations and some arithmetic. At most $3p(p + 1)/2 + 1$ tests are performed. We count only explicit tests, not tests against p that terminate loops, since the latter will not appear after loops have been unrolled. The general process exploits the representation of values so as to handle the rule for combining matched positive and negative values without the cost of determining which of the two values is positive and which is negative. The general process also organizes the search through pairs of values carefully.

```
R := 0
do i := 1 to p
   U := (i-th input value)
   if ( U ≠ −1 ) then
      do  if ( U = 0 ) then Return ( 0 )
          if ( U ≥ Xrep )
          then R := Xrep
          else  do  R := BitwiseOR(R,U)
                    do  j := i+1 to p
                        V := (j-th input value)
                        if ( V ≠ −1 ) then
                           do  N := BitwiseXOR(U,V)
                               if ( N = 1 ) then Return ( 0 )
                               if ( N ≠ 0 ) then R := Xrep
                           end
                    end
                end
      end
end
if ( R ≠ 0 ) then Return ( R ) else Return ( −1 )
```

Figure 6. Implementation of refined AND with p inputs.

Thanks to the fact that the simple and refined calculations share the same representation of the two zillion and three values, it is possible to get the same results as the refined calculations without actually doing them at each gate. At any gate with just two inputs, the refined calculation tailored for this number of inputs can be done as a matter of course: it takes only about twice as long as the simple calculation. At any gate with more than two inputs, on the other hand, we perform the simple calculation first and then test whether the output is X. If it is not X, then there would be nothing to gain by a refined calculation. If it is X, then we perform the refined calculation before continuing to the next gate. For gates with at most four inputs, the simple calculation followed by the test for X entails a total of at most three tests. This is much less than the 19 (or 31) tests that the general refined calculation in Figure 6 may require for a gate with three (or four) inputs.

6. Some Final Words

We have described simulation techniques that, to a limited extent, keep track of distinct unknown values. We have given applications and implementation methods to show that simulation with two zillion and three values is useful and practical. To put the speed of (2Z+3)-valued simulation into perspective, it is helpful to recall that event-driven logic simulators often run at about 1000 or 2000 events per second on machines that run at about a million instructions per second [5]. Processing an event entails performing 500 to 1000 machine instructions, some of which are devoted to evaluation of logic gates. Even in the most expensive common case (that of a 4-way AND under the refined rules), our pseudocode would require fewer than 200 instructions in place of whatever instructions would normally be devoted to gate evaluation. Processing an event under (2Z+3)-valued simulation would entail performing at most 700 to 1200 machine instructions, so the rate of processing could decrease, but only moderately. At worst, the rate might change from 1000 (or 2000) events per second to 800 (or 1400) events per second. In the applications discussed here, the moderate decrease in rate buys a major increase in the amount of information.

The overhead of event-driven simulation is high enough to justify using the refined rules as a matter of course. On the other hand, under a low-overhead simulation strategy (for example, see [1]), a significantly higher gate-evaluation rate might be achieved with the simple rules. The two variants of (2Z+3)-valued simulation represent different choices in the tradeoff between detail of information provided and speed of simulation. Other choices are also possible, and might be advantageous for further applications.

References

1. Z. Barzilai, J.L. Carter, B.K. Rosen, and J.D. Rutledge, "HSS - A High-Speed Simulator", *IEEE Transactions on Computer-Aided Design*, Vol. 6, Num. 4 (July 1987), 601-617.
2. M.A. Breuer and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.
3. J.A. Darringer, "The Application of Program Verification Techniques to Hardware Verification", *Proc. 16th ACM-IEEE Design Automation Conf.* (June 1979), 375-381.
4. E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability", *Proc. 14th ACM-IEEE Design Automation Conf.* (June 1977), 462-468.
5. J. Werner and R. Beresford, "A System Engineer's Guide to Simulators", *VLSI Design*, Vol. 5, Num. 2 (Feb. 1984), 27-31.