A Scalable Symbolic Simulator for Verilog RTL

Sasidhar Sunkari*, Supratik Chakraborty†, Vivekananda Vedula‡ and Kailasnath Maneparambil§

* Intel, Bangalore
Email : sasidhar.sunkari@intel.com
phone: +91 80 26053655

† Indian Institute of Technology, Bombay
Email : supratik@cse.iitb.ac.in

‡ Intel, Bangalore
Email : vivekananda.vedula@intel.com

§ Intel, Arizona
Email : kailasnath.s.maneparambil@intel.com

Abstract

Symbolic simulation is an important technique used in formal property verification and test generation for digital circuits. Existing symbolic simulators predominantly operate at the gate level, reasoning about individual bits and signals. As a result, their performance does not scale well to large circuits like microprocessors. Word-level symbolic simulators address this problem to some extent, but present other challenges, such as fixpoint detection when simulating multiple modules that mutually trigger each other. In this paper, we present some exploratory ideas for performing word-level symbolic simulation over a Verilog RTL description of a circuit. We outline the basic technique of simulation and of handling fixpoints, discuss issues faced in our approach and present solution techniques to counter these issues. We also present initial experimental results obtained by applying our algorithms to a Verilog model of an x86 processor design.

I. Introduction

Simulation with specific test inputs is the predominant technique used in functional verification of digital circuits. Unfortunately, it is impractical to exhaustively simulate every possible execution of large and complex circuits like microprocessors. Symbolic simulation offers a promising alternative by combining the flexibility and scalability of conventional simulation with the analytical power of sophisticated symbolic methods. In symbolic simulation, we replace multiple simulation runs, each with different inputs, with a single run in which the inputs are assigned symbolic values. The output of a run of symbolic simulation is a set of symbolic expressions representing possible values of circuit nodes at different time instants. By reasoning about these symbolic expressions, it is often possible to infer properties of the circuit that would have otherwise required a large number of specific input based simulation runs to discover. Symbolic simulation has been used in earlier work for formal and semi-formal property verification, as well as for test case generation. Early symbolic simulators that were successfully applied on non-trivial circuit designs dealt with simulation at the switch or gate-level [1], [2], [3], [4], and reasoned about individual bits or signals. For circuits of the scale of microprocessors, however, reasoning at the level of switches and individual bits is prohibitively expensive. It is therefore necessary to simulate such large circuits at the word-level, starting from a Register Transfer Level (RTL) description of the design. While there has been earlier work on word-level symbolic simulation for RTL designs [5], these approaches have primarily restricted the simulation to a pre-determined (or

This work was supported by Intel Corporation, and was done while Sasidhar Sunkari was at Indian Institute of Technology, Bombay.

user-provided) number of simulation cycles. This works well if all interesting signal transitions at all circuit nodes happen within the pre-determined number of cycles. Several RTL designs, however, have a set of blocks that mutually trigger each other. In such cases, it may be difficult to statically determine how many simulation cycles must elapse before the values of all nodes stabilize. Hence, fixing the number of simulation cycles a priori may not be the right way to symbolically simulate such designs. In this paper, we describe our ongoing effort towards building a symbolic simulator for Verilog RTL designs that addresses the above issue through a fixed point formulation, and also scales well in performance with the size of the RTL description. This work is part of a larger project that aims to use symbolic simulation and approximate constraint solving techniques for test instruction generation for microprocessor designs. However, the focus of this paper is on the design of the symbolic simulator.

To illustrate the potential advantages of word-level symbolic simulation over bit-level simulation, consider the example shown in Fig. I. This example has an adder with inputs I_1 and I_2. When the circuit is started, the values of I_1 and I_2 are provided by the user through the multiplexers at the inputs of the adder. The arithmetic sum obtained at the adder output is loaded into register R at the next rising edge of the clock. The output of the register is then fed to a left shifter (a combinational circuit) T, that effectively multiplies its input by 2. The values of I_1 and I_2 in subsequent cycles of operation are obtained from the outputs of R and T respectively through the multiplexers. We assume that all operands are treated as unsigned integers, all datapaths are 64 bits wide, and when overflow occurs, the least significant 64 bits are retained.

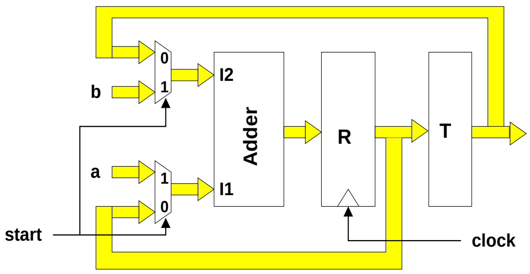


Fig. 1. Block diagram of a sequential circuit

Suppose the initial symbolic values of I_1 and I_2, as given by the user, are a and b respectively. *Assuming no overflows, i.e. all symbolic expressions evaluate to values between* $2^{64}-1$ *and* 0, the word-level simulation of the above circuit for the first four clock cycles is shown in Table I. For clarity of presentation, the expressions in the Table have been simplified arithmetically. These expressions can now be used for performing various analyses of the circuit in Fig. I. Specifically, a property that is inferred from these expressions holds true for all simulation runs starting from values of a and b that don't result in overflows in the first four cycles. Thus the result of symbolic simulation can be used for property checking. The expressions obtained above may also be used to generate test cases or to prove untestability of specific test goals. For example, suppose we wish to find values of inputs a and b such that the most significant bit (MSB) of register R is 1 in the fourth clock cycle without any overflows in any datapath operation. From Table I we find that the expression for R in the fourth clock cycle is $27(a+b)$. Since register R is 64-bits wide, we require $27(a+b) \geq 2^{63}$. To ensure that the output of the adder doesn't overflow in the fourth clock cycle, we also require $81(a+b) \leq 2^{64}-1$, so . Unfortunately, both these constraints cannot be simultaneously satisfied. Therefore it is impossible to obtain test inputs that will set the MSB of R to 1 in the fourth clock cycle without causing any overflows. In fact, since the output of the adder in any cycle (≥ 1) is three times the value of R in the same cycle, it is impossible to have the MSB of R set to 1 without the output of the adder overflowing, if all data buses have the same width.

The above example illustrates how word-level symbolic simulation scores over bit-level symbolic simulation. If we were to solve the same test generation problem by symbolically simulating the circuit at the level of individual bits, we would have generated a separate expression for each bit at the output of each block. These expressions would typically be represented as Binary Decision Diagrams (BDD) or propositional formulae in Conjunctive Normal Form (CNF). Next, a SAT solver or BDD-based engine would be used to check the propositional satisfiability of the expression generated for the most significant bit of register R. In general, this may involve far more computational effort than the above analysis based on word-level symbolic expressions, in which we reasoned using properties of bit-vectors or integers instead of using bit-level reasoning. Note also that the word-level expressions indicate the impossibilty of finding a test input that sets the MSB of R to 1 without causing any overflows, independent of the number of clock cycles and width of the datapath, as long as all data buses have the same width. It would be very difficult to arrive at the same conclusion using bit-level expressions. This demonstrates the power of symbolic reasoning at the word-level.

The focus of this paper is on the design and development of a word-level symbolic simulator for Verilog RTL designs. Our simulator takes as input a circuit design in Verilog and performs word-level simulation of the various behavioral constructs. This generates symbolic expressions encoding possible values of different circuit nodes. The

TABLE I. Symbolic simulation of circuit in Fig. I

Clock	I_1	I_2	Adder output	R	T
0	a	b	$a+b$	Unspecified	Unspecified
1	$a+b$	$2(a+b)$	$3(a+b)$	$(a+b)$	$2(a+b)$
2	$3(a+b)$	$6(a+b)$	$9(a+b)$	$3(a+b)$	$6(a+b)$
3	$9(a+b)$	$18(a+b)$	$27(a+b)$	$9(a+b)$	$18(a+b)$
4	$27(a+b)$	$54(a+b)$	$81(a+b)$	$27(a+b)$	$54(a+b)$

following sections describe the various issues faced in developing our prototype simulator, and discusses techniques used to overcome these issues. Section II outlines related work in symbolic simulation for property verification and test case generation for digital circuits. Our approach to performing word-level symbolic simulation is then presented in section III. An interesting sub-problem that needs to be addressed is the identification of sets of `always` blocks that mutually (or recursively) trigger each other. Section IV describes an efficient and approximate technique used in our simulator for identifying such `always` blocks. This allows our simulator to scale to large designs while conservatively identifying `always` blocks that are recursively triggered. Section V presents some initial experimental results on a Verilog x86 design. Finally, Section VI concludes the paper with a note on directions for future work.

II. Related Work

Formal verification of switch and gate-level digital circuits using symbolic simulation and a ternary valued lattice was extensively studied by Bryant and Seger in their seminal work on Symbolic Trajectory Evaluation(STE)[1], [2], [3]. This technique has subsequently been implemented in tools like COSMOS, and has been successfully applied to verify the correctness of several circuits [4]. In ternary valued symbolic simulation, expressions at circuit nodes are often encoded and represented using BDDs. Wilson et al [6] studied the use of approximations to reduce the sizes of BDDs in bit-level symbolic simulation. Their approach selectively assigns approximate values (combinations of 0, 1 and X) to internal nodes in a circuit, when such approximate assignments do not affect the functionality being tested or verified.

Su et al [7] used a form of symbolic simulation and special logic transformation techniques to automatically identify invariants in finite-state transition systems. Zeng et al used symbolic simulation and word-level satisfiability checking to generate functional test vectors [5]. They considered using both an integer linear program formulation and a constraint logic program formulation for word-level satisfiability checking. In general, solving a set of constraints on expressions obtained from symbolic simulation involves reasoning about multiple theories like those of bit-vectors, arrays, bounded integers, uninterpretted functions, etc. The development of powerful Satisfiability-Modulo-Theories (SMT) solvers in recent years has greatly aided the use of symbolic simulation and constraint solving based approaches for property verification and test generation of digital circuits..

The theory of abstract interpretation, first formulated by Cousot and Cousot [8], has been used with significant success to tame the practical complexity of verification in the context of sequential programs. In this approach, a concrete domain of values is mapped to a "simpler" abstract domain that permits easier reasoning, albeit at the cost of some precision loss. Once the analysis has been performed in the abstract domain, the results can be re-interpretted in the concrete domain, often helping in proving properties of complex systems. The theory of abstract interpretation guarantees the soundness of any such verification result, although there are no completeness guarantees. Since word-level symbolic simulation may also be looked upon as abstract execution of a program in a hardware description language (HDL), the theory of abstract interpretation offers promising ways to deal with the complexity of word-level symbolic simulation.

Despite some industrial interest, symbolic simulation is however yet to see widespread usage in commercial tools. Among the commercial offerings that use symbolic simulation technology, Blue Pearl Software has a tool for checking whether a chip meets its timing goals. This tool uses proprietary technologies for RTL analysis, high level symbolic simulation and state space exploration. Innologic (currently part of Synopsys) uses symbolic simulation for checking equivalence of designs at different levels of description. Nusym is yet another company that makes use of symbolic simulation and analysis techniques for automation of test extraction, coverage estimation and coverage convergence.

III. Word Level Symbolic Simulation

The symbolic simulator described in this paper simulates only the synthesizable subset of Verilog. Synthesizable Verilog has certain restrictions which make the design of the simulator easier. For example, the number of iterations for each loop statement is known at compile time and hence they can be unrolled before the expressions are generated . The behavioral code thus contains only the conditional statements and assignments.

The symbolic expressions are generated by parsing all the control flow paths in each behavioral block (always statement). The input data structure used for word-level simulation is explained in III-A.

A. Control Data Flowgraph

In word-level symbolic simulation, the circuit model is a description of the circuit at RTL. The given RTL design is translated into a collection of interconnected *Control Data Flow Graphs*(CDFGs). Each CDFG represents the control and data flow within an always block of a module. Thus we would have as many CDFGs as the number of always blocks in the design. A representative node for each module contains the rest of information like continuous assignments, links to CDFGs of all always blocks within the module, etc. The representative nodes for each module are connected by instance nodes according to the instantiations made in the design. The port connectivity information is contained in these instance nodes. The expressions are generated by traversing the CDFG and executing the statements in accordance with the semantics of Verilog language[9]. 'Execution of a statement' here means symbolic manipulation of words. The generated expressions have word level operators like arithmetic add, left shift, etc which are used in the RTL design. We define a *word* as "*a contiguous vector of bits in a variable that remains undivided through out the design*". Alternately, a *word* is "*a bit vector whose bits are read or written all at once*", that is, there does not exist a case where only a part of the word is read/written and the remaining are not. For example, considering a and b to be two 32-bit variables, the following lines of Verilog code
a[0:9] = 10b'1;
a[10:31] = b[10:31];
causes the division of words as a[0:9], a[10:31], b[0:9] and b[10:31].
Also, if we have an assignment
x[0:9] = y[0:9]
where x[0:9] is unfragmented but y[0:9] is fragmented into y[0:4] and y[5:9], then we also breakdown x[0:9] into x[0:4] and x[5:9] and make the assignments
x[0:4] = y[0:4] and x[5:9] = y[5:9]

The Language Reference Manual for Verilog [9] describes a simulation algorithm which is based on discrete event model and uses a stratified queue. The value updates to variables in the design causes events to be generated which are enqueued in a priority queue. Word-level symbolic simulation was tried using this algorithm to simulate an example design. Informally, the simulation algorithm is to traverse all the control paths of the top module and execute the respective statements(continuous assignments, port assignments and always blocks) which are triggered due to assignments made in the traversal. However it was observed that *the simulation never terminated!* This section describes the reason for the failure of the naive simulation algorithm. It presents the challenges posed by the $always$ statement in performing symbolic simulation and presents our approach of dealing with them.

The major issue that was encountered was to detect a change in a symbolic expression which is present in the sensitivity list of an always statement. In other words, given two word level symbolic expressions E_1 and E_2, we wish to say whether these two expressions are equivalent or not. In our implementation, it was assumed that an assignment to a variable, would have possibly changed the value of the variable and executed statements that are sensitive to the variable being assigned. However, there could exist synthesizable Verilog designs where an always block execution can possibly cause a series a excitations which propagate back to the sensitivity list of the always block itself and thus re-trigger itself. Such a case leads to an infinite sequence of executions and hence the simulation algorithm never terminated. The below example is a case where such an infinite triggering is possible.

module A(a, b, c)	**module** B(x, y)
input[0:2] b,c;	input[0:2] x;
output reg[0:2] a;	output reg[0:2] y
reg[0:2] x,y;	**always**@(x)
B b(x,y);	**if**(x==3b'100)
always@(b or c or y)	y=x<<1
if(b==3b'000)	**endmodule**
x = c << 1	
else	
a = y & b	
endmodule	

Let us assume that we are symbolically executing the always block in module A which causes an assignment to the variable x. This value change propagates to the instance of module B and causes its always block to execute. This execution assigns a value to the variable y which is propagated back to the parent module A. Now, since the always block in module A is sensitive on y,it is executed again. Thus, the chain of events can repeat for an infinite number of times.

It is possible to define an **Excitation Dependency Graph** to depict the chain of events that can possibly cause an always block to recursively trigger itself. The nodes of an excitation dependency graph are the words of each instance and the always blocks of each instance in the design. The excitation dependency graph shows how each entity could possibly trigger another entity directly.

There exists edges between various entities according to the following rules.

- Each word in the right hand side of a continuous assignment has an edge from it to all words in the left hand side of the same assignment.
- Each word in a port assignment has an edge to/from the respective ports of the child instance depending on whether the port assignment is input/output.
- An always block has an edge from itself to all the words that are defined with in the always block.
- A word in the sensitivity list has an edge from itself to the always block it is triggering.

The excitation dependency graph for the example Verilog code mentioned above is shown in figure III-A. The nodes are labeled as <module-name>_entity. *A possible recursive triggering of an always block is identified by a cycle in the excitation graph.*

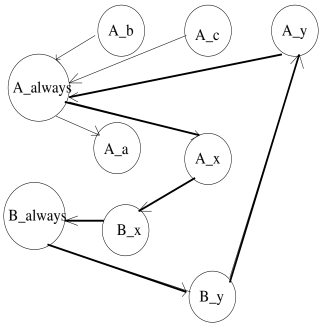
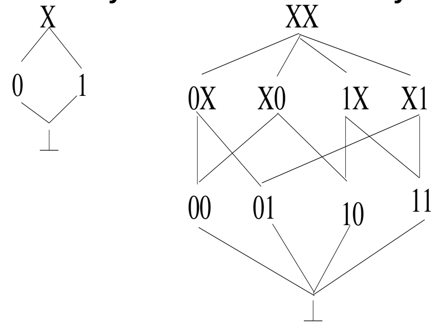


Fig. 2. Excitation Graph

B. Expression Generation for Recursively Triggering Always Blocks

We firstly introduce the idea of ternary model used in STE [2] and then present an extension of this model which we use to generate word level expressions. Randal.E.Bryant and C.J.Seger used the **Ternary Model**[2] to perform symbolic simulation over switch and gate level circuits. As an extension to the binary domain which contains the values $\{0, 1\}$, the ternary domain (**T**) contains the values $\{0, 1, X\}$. X is typically used to denote an unknown or an indeterminate value. A partial ordering relation is defined on the three values 0, 1 and X on the basis of "weakness in information content" denoted by $\sqsubseteq$.
$\mathbf{0} \sqsubseteq X$ and $\mathbf{1} \sqsubseteq X$
Also, to make this partial ordering a lattice, we introduce a new value $\perp$ and define $\perp \sqsubseteq \mathbf{0}$ and $\perp \sqsubseteq \mathbf{1}$
A simple extension to the ternary domain (**T**) to allow for operation on words is shown in figure 3 . Let a and b be two words each of length n. The i^{th} bit of a is addressed as $a[i]$. We define $a \preceq b$ if and only if :
for all i in 1 to n, $a[i] \sqsubseteq b[i]$ and
Any value $\sqsubseteq \perp$
Figure 3 summarizes the partial ordering relation ($\preceq$) on ternary and extended ternary models

Fig. 3. Ternary and Extended Ternary Models



Indeed, an extended ternary domain for n-bits may actually be obtained doing a n-time product ($\mathbf{T}^n$) of the simple ternary domain. However, we prefer to use a smaller lattice by representing a set of values in $\mathbf{T}^n$ by a single value. In this case,the least upper bound of every element in the power set of $\mathbf{T}^n$ is used as a representative value for that element.

We assume that *each always block executes for a finite number of times between every two clock ticks.* This means that after a certain number of executions, the variables in the sensitivity list reach their fixed-points, ie., further executions of always blocks do not produce any change in the values assumed by these variables. (The formulation presented below however holds even if this assumption is violated.) We call an *epoch* as the uniform time period during which all signals stabilize in their values. For convenience, we may assume that each epoch aligns with a clock tick. We assume that each word takes values from the extended ternary lattice. The summarization of two values v_1 and v_2 in our domain is obtained by finding the least upper bound of both the values, **lub**(v_1,v_2).

Let us assume that there exists a module M with a single always block. Let $x_1, x_2, ..., x_n$ be the n variables in the module M. Let us symbolically execute the always block *once* with the initial values of the variables $x_1, x_2, ..., x_n$ to be $v_1, v_2, ..., v_n$, represented in vector form as $\overline{v}$. The new values for each variable after executing the always block once can be given as function of $\overline{v}$. Let $E_i(\overline{v})$ be the expression generated for the variable x_i by executing the always block once.

The new values for variables $x_1, x_2, ..., x_n$ are therefore $E_1(\overline{v}), E_2(\overline{v}), ..., E_n(\overline{v})$ respectively. This is represented in vector form as $\overline{E}(\overline{v})$. Let $\overline{I}$ be the vector of initial values for the variables in the current epoch.

Ideally, we would like to find the value of $\overline{E}^n(\overline{I})$ as n tends to infinity. However, since we donot know when it is going to converge, we keep summarizing the values a variable assumes during a simulation run. If the always block is executed zero or one time, its effect on a variable x_i is , given by, **lub**(v_i,$E_i(\overline{v})$). In a similar way, we can capture the effect of executing the always blocks for zero or more number of times by repetitively applying the **lub** operator. We repeat this process until the application of **lub** reaches a fixed point. The expressions for the set of variables defined in a recursively triggered always blocks are therefore of the form :
lfp(**lub**($\overline{I}, \overline{E}(\overline{v})$))
where **lfp** is the least fixed point operator.

For the above formalism to be applied, we need that the expressions E_i satisfy the following :
1. $E_i(\bot) = \bot$
2. $E_i(\textbf{lub}(a, b)) \preceq \textbf{lub}(E_i(a), E_i(b))$
Both these properties are satisfied by the operators in Verilog since they are described only on the concrete binary domain.

A key point to note here is that the fixed point of the function above is obtained in a finite sequence of steps. This is because, each time we apply a **lub** operation over two operands, we either return one among the operands or move higher than the two operands in the extended ternary lattice. In the former case, we reached a fixed point. In the later case, we apply **lub** once again on the resulting value and some other value. As this process continues we are sure to hit a fixed point, because the lattice is a finite structure and once the topmost point is reached, applying **lub** would return the same value. Thus, in the extreme case, once could go up to the value of XXX and stop there. Termination of fixed point computation process in a finite time is indeed a desirable feature of using the extended ternary lattice.

The caveat however is that, by moving higher in the lattice, we are over generalizing the values and loosing information. In the worst case, we could end up saying that a variable assumes the value XXX which gives absolutely no information about the actual value of the variable. It should be noted that it is not always true that $\overline{E}(\overline{I})$ is $\preceq$ than $\overline{I}$. In other words, $\overline{E}(\overline{I})$ could be equally strong in information content as is $\overline{I}$. Hence applying a **lub** operation over these values can result in summarizing values too much. A simple enhancement could be that, we find $\overline{E}^k(\overline{I})$ for some value k and if a fixed point is not reached by then, we start summarizing using the **lub** operator.

From the above formulation, we get the stabilized values at the end of one epoch. For the variables that do not represent external inputs, the values at the end of this epoch serve as the initial values for the successive epoch. It should be noted that the function $\overline{E}$ is obtained by symbolically executing each always block in every module *once*. Thus, parsing through the CDFG of the design once should be sufficient to generate the required expressions.

We present below the expressions obtained by one time execution of always block for the example design described above. These expressions are actually the definitions for the function E_i presented in the above formalism. For a variable v in module M, let the initial value of the variable be M_v and the final value obtained after executing the always block once be M_v'. All the operators used in the expressions have the same meanings as those in Verilog HDL.
For module A :
A_x$'$ = (A_b==000) ? (A_c << 1) : A_x
A_a$'$ = (A_b!=000) ? (A_y & A_b) : A_a
For module B :
B_y$'$ = (B_x==100) ? (B_x << 1) : B_y
From the port connection statements we have that :
B_x$'$ = A_x$'$ and
A_y$'$ = B_y$'$
Therefore, by making the port assignments in order (input followed by output), we get the following values for the variables in terms of primary inputs :
For module B
B_y$'$ = (((A_b==000) ? (A_c << 1) : A_x)==100) ? (((A_b==000) ? (A_c << 1) : A_x) << 1) : B_y
For module A :
A_x$'$ = (A_b==000) ? (A_c << 1) : A_x
A_a$'$ = (A_b!=000) ? (((((A_b==000) ? (A_c << 1) : A_x)==100) ? (((A_b==000) ? (A_c << 1) : A_x) << 1) : B_y) & A_b) : A_a

C. Other Extensions

Edge Triggered Always Statements : To deal with edge triggered always blocks we propose a simple extension to the above approach. We assume that our domain has two more values $\uparrow$ (for a positive edge) and $\downarrow$ (for a negative edge). Also we introduce three new functions :
posedge(x) - returns true, false or indeterminate states
negedge(x) - returns true, false or indeterminate states

ite'(c,t,e,i) - evaluates condition c. If it is true, t is evaluated and returned. If it is false e is evaluated and returned. If it is indeterminate i is evaluated and returned.
As an example, consider the design:

```
always@(posedge clock)
begin
    out = in;
end
```

The expression for the new value of variable out may be given as : **ite'**(posedge($clock$),in,out,lub(in, out))

Ports and Continuous Assignments : According to Verilog semantics, continuous and port assignments can be made as soon as there is a change in the value of its operands. Since these assignments always represent equations that hold true in a stable state, we model each continuous assignment and port assignment as a constraint.

D. Comparison with Gate-Level Symbolic Simulation

The technique of symbolic simulation used in Symbolic Trajectory Evaluation (STE)[2] operates on the gate level model of a circuit. Besides the level of abstraction on which the simulation is carried on, there is also a difference in the semantics of the expressions generated by STE as compared to our approach. In STE, expressions are generated by *propagating* the input symbolic values into the circuit according to the circuit excitation functions. At any given instant of time, a snapshot of this simulation represents a set of possible scalar value simulations which can be obtained by instantiating the input symbols to various values.

On the other hand, our approach to simulation attempts to capture symbolically the local relations within a behavioral construct of the design. An actual simulation run on hardware corresponds to a particular ordering these local relations. Unlike in STE, we cannot get a valid simulation run by merely instantiating all the symbols to some values. It is necessary to impose an ordering among these expressions.

IV. Excitation Dependency Analysis

The fixed point formulation for expression generation leads to summarizing of values and hence may result in approximate outputs. We propose certain techniques to improve the accuracy of the approximations by separating those always statements which need a fixed point formulation from those that do not. Clearly, the always blocks that do not lie on a cycle in the excitation dependency graph do not need a fixed point formulation as they are executed atmost once in a single clock tick.

A naive approach to identify the recursively triggering always blocks is to first create an excitation dependency graph between various entities of the design as mentioned before. Once we obtain the initial excitation dependency graph, finding the transitive possibilities of excitation is simply obtained by finding the transitive closure of this graph. If after finding the transitive closure, we see that an always block excites itself, then it means that this always block can potentially re trigger and hence expressions are to be generated using the fixed point formulation. The caveat in this approach, however, is that this does not scale well with the size of the design. Assuming that there are n entities in the design, finding transitive closure in the worst case could be O(n^3).

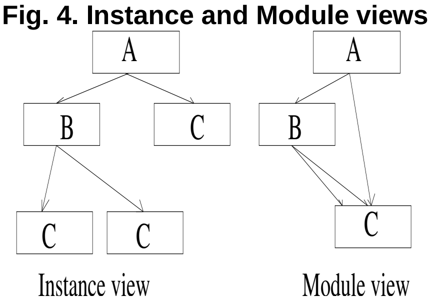
A. An Approximate Algorithm

We now propose an approximate algorithm which works much faster than the naive approach but with an approximation. This algorithm uses the distinction between the instance level view and module level view of the design. An *instance level view* would mean the picture that we get by instantiating every module according to each instance statement in the design. In this view, each instance(except the topmost one) has one and only one parent. On the other hand, in *module level view*, we instantiate each module only once and allow the module to have multiple parents if it is instantiated more than once. The following example clarifies the difference between both the views.

module A	**module** B(i, o)	**module** C(i, o)
.	.	.
B b(b_i,b_o)	C c1(c_i1,c_o1).	.
C c(c_i,c_o)	C c2(c_i2,c_o2)	**endmodule**
.	.	
endmodule	**endmodule**	

The instance and module views of the above design are shown in figure 4.

Fig. 4. Instance and Module views



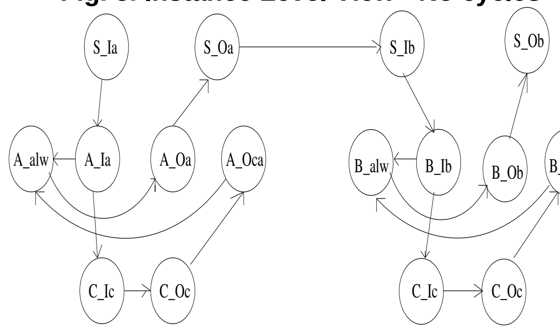
The approximate algorithm detects cycles by finding transitive closure on the module level view of the design.

Since each module is considered only once, we get a much smaller excitation dependency graph and hence transitive closure computation will be faster. From the properties of instance and module level views it can be proved that *All cycles in the instance level view indeed occur as cycles in the module level view.* However, some of the dependencies here are inexact. For instance, in the example presented earlier one could wrongly infer that variable c_i1 in module B can have an edge with the variable c_o in module A thorough module C which has multiple parents(both A and B). Thus, because of these stray dependencies one could arrive at flagging more always blocks to be recursively triggered than those that actually do. Also, after the always blocks that recursively trigger themselves in the module level view are identified, we assume that each instantiation of this always block in the instance level view is also recursively triggered. This is yet another approximation, because not all instantiations of this always block need to be on cycles.

Below is an example of how the module level view algorithm can infer cycles wrongly. Consider the following Verilog code.

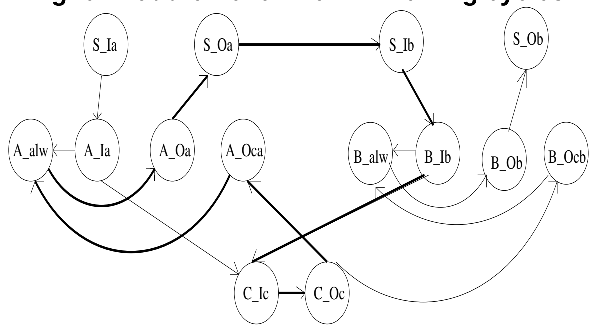
module S reg [0:9]Ia,Ib,Oa,Ob; **assign** Ib = Oa; A a(Ia,Oa); B b(Ib,Ob); **endmodule**	**module** A(Ia, Oa) reg [0:9]Ia,Oa,Oca; C c_a(Ia,Oca); **always**@(Ia or Oca) Oa = Ia + Oca; **endmodule**
module B(Ib, Ob) reg [0:9]Ib,Ob,Ocb; C c_b(Ib,Ocb); always@(Ib or Ocb) Ob = Ib + Ocb; **endmodule**	**module** C(Ic, Oc) reg [0:9]Ic,Oc; **assign** Oc = Ic $<<$ 1; **endmodule**

Fig. 5. Instance Level View - No cycles



The excitation dependency graphs obtained by instance and module level views of the above example design is shown in figure 5 and 6. It can be observed that in the instance level view, one does not infer a cycle, but in the

Fig. 6. Module Level View - Inferring cycles.



module level view, however, a cycle is inferred in the path B_Ib → C_Ic → C_Oc → A_Oca → A_alw → A_Oa → S_Oa → S_Ib → B_Ib.

Assuming there are k entities in the design at module level, the worst case time complexity of this algorithm is O(k^3), since this is also a problem of finding the transitive closure.

V. Experimental Results

The simulation was run on parts of an example design called "**x86**". The **x86** project was an effort by the VLSI team at IIT Madras to create a public domain RTL synthesizable for the x86 CPU. Details of this project can be found at http://vlsi.cs.iitm.ernet.in/x86_proj/x86HomePage.html. Table II is a brief description of the complexity of the project at both instance level and module level views.

Table III shows the sizes of the largest expressions obtained by one time execution of always blocks of certain modules. These are the sizes of E_is as referred in section 3.2. Since these expressions are represented as DAGs, each common sub expression is counted only once. The number of common subexpressions as well as the tree sizes(expanded dags) for these modules is also presented.

Table IV compares the number of always blocks that actually exist in the design with the number that were detected to be on cycles by the approximate and naive algorithms. The naive algorithm on the complete x86 couldnot terminate before the timeout period of 6 hours when run on an IBM PC (Pentium 4 - 2.4GHz, 1GB RAM). In the case of simulating the complete x86, the size of the graph is 150,472 (which is the sum of the number of always blocks and the number of atoms in the instance view as shown in table II). Comparatively, the size of the corresponding graph used by the approximate algorithm

TABLE II. Module and instance views of x86

	Module View	Instance View
No of instances	122	12,111
No of always blks	123	357
No of words	5,888	150,115

TABLE III. Word level expression sizes

Module	Word	Size	No of common sub-exps	Tree size
conUnit	infp2[0:79]	938	34	29305
conUnit	buffer[0]	762	6	1646
conUnit	dataWr[0:31]	1449	86	4102
conUnit	address[27:35]	1184	61	24598
myAlu	cla1op1[0:31]	1472	4	2700
myAlu	twoscompIn1[0:31]	972	3	1739
myAlu	zfin[0:31]	1890	32	4608
myAlu	Result[1:3]	3524	4	6390
Decoder	MemOpEn[0]	964	6	1309
Decoder	ARPLEn[0]	678	2	921

TABLE IV. Detecting always blocks on cycles.

Module	#always blks	#detected on cycle	
		Approx Algo	Naive Algo
full x86	357	221	-
multiply32	33	33	33
Decoder	11	11	11

has only 6011 ndoes. It is worthnoting that, despite a huge reduction in graph size by the approximate algorithm, it was still able to infer that 136 $always$ blocks are not on cycle. In the decoder unit each module is instantiated only once and hence, the approximate algorithm performs the same as the naive algorithm.

VI. Conclusion

Our focus of current research is on building up of a constraint solver which can reason on lattice operators introduced by the fixed point formulation. SMT solvers which operate on boolean as well as bit vector domains are being studied. The expressions that the simulator generates are yet large and it is necessary to optimize or approximate them for the idea of word-level symbolic simulation to be made applicable on large designs such as microprocessors.

Acknowledgment

The authors would like to thank Ashutosh Kulkarni and Kaustubh Nimkar for their inputs and help in the implementation of the symbolic simulator.

References

[1] R. E. Bryant, "Symbolic simulation - techniques and applications," in *Design Automation Conference*, 1990, pp. 517–521. [Online]. Available: citeseer.ist.psu.edu/bryant90symbolic.html

[2] R. E. Bryant and C.-J. H. Seger, "Formal verification of digital circuits using symbolic ternary system models," in *Proc. 2nd International Conference on Computer-Aided Verification (CAV'90)*, E. M. Clarke and R. P. Kurshan, Eds. Springer-Verlag, 1991, vol. 531, pp. 33–43. [Online]. Available: citeseer.ist.psu.edu/643147.html

[3] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design: An International Journal*, vol. 6, no. 2, pp. 147–189, March 1995. [Online]. Available: citeseer.ist.psu.edu/seger93formal.html

[4] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant, "Formal verification of powerpc arrays using symbolic trajectory evaluation," in *Design Automation Conference*, 1996, pp. 649–654. [Online]. Available: citeseer.ist.psu.edu/article/pandey97formal.html

[5] Z. Z. et al., "Functional test generation based on word level sat," in *Journal of Systems Architecture*, 2005, pp. 488–511. [Online]. Available: citeseer.ist.psu.edu/su96automatic.html

[6] C. Wilson, D. L. Dill, and R. E. Bryant, "Symbolic simulation with approximate values," in *Formal Methods in Computer-Aided Design*, 2000, pp. 470–485. [Online]. Available: citeseer.ist.psu.edu/wilson00symbolic.html

[7] J. X. Su, D. L. Dill, and C. W. Barett, "Automatic generation of invariants in processor verification," in *First international conference on formal methods in computer-aided design*, M. Srivas and A. Camilleri, Eds., vol. 1166. Palo Alto, CA, USA: Springer Verlag, 1996, pp. 377–388. [Online]. Available: citeseer.ist.psu.edu/su96automatic.html

[8] P. Cousot and R. Cousot., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252, 1977. [Online]. Available: citeseer.ist.psu.edu/bryant86graphbased.html

[9] I. std 1364-2001, *Verilog Hardware Description Language*, IEEE Computer Society.