Cycle Simulation Techniques

Samir Palnitkar
Sun Microsystems, Inc. Mountain View CA
samirp@eng.sun.com

Darrell Parham
Sun Microsystems, Inc. Mountain View CA
drp@eng.sun.com

Abstract

Nowadays, most hardware design is being done at a high level of abstraction, such as a hardware description language. Hence, simulation constitutes a significant part of the design verification process. In this paper, we study cycle simulation techniques that could potentially speed up simulation. Then we propose various metrics to predict the performance of a cycle based simulator which uses these cycle simulation techniques. A justification is provided as to why each metric is the best possible indicator of a certain characteristic of the design. Finally we summarize the results obtained from the analysis of various designs and draw inferences from them.

1.0 Introduction

With more and more hardware design being done using hardware description languages, simulation has become a very important part of the entire design cycle. Time spent in simulation constitutes a high percentage of the time required for each design iteration. Faster simulation speeds mean a notable decrease in the cycle time for each simulation run of the design and hence quicker turnaround times for the functional verification of the design. Efforts are being made by many simulator developers to push simulation speeds as high as possible. Various innovative techniques and algorithms have been utilized to build faster simulators.

In this paper we study two cycle based simulation techniques which we believe will result in simulation speedup. To study the effect of these techniques on a simulator, we have developed various metrics which will allow us to weigh the various trade-offs in implementing these techniques and to draw reasonable conclusions about the potential speedup that can be obtained using the discussed cycle based simulation techniques. A justification is provided as to why each measured quantity is the best possible indicator of a certain characteristic of the design.

In section 2 we study the two cycle based simulation techniques and list the underlying assumptions to apply these techniques to simulation. In section 3 we propose the various metrics, explain what they measure and justify why they were chosen. In section 4 we present the results obtained from the analysis of designs and draw inferences from these results. Finally in section 5 we state the conclusions of our study.

2.0 Cycle based simulation techniques

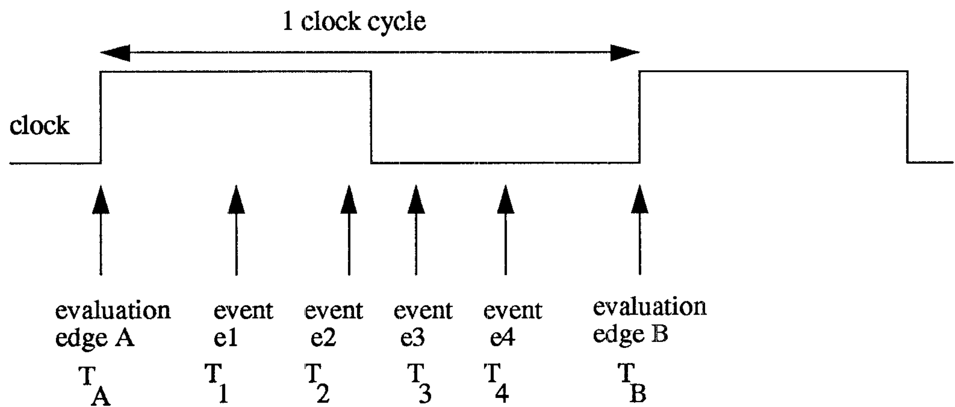
In this section we study two techniques that we believe will potentially speed up simulation. The first technique applies to an event driven simulation algorithm and the second technique applies to an oblivious simulation.

2.1 Event ordering and delayed evaluation

This technique suggests that all scheduled evaluations will take place only at the active edge of clock. Any event that is scheduled within a clock cycle will only be evaluated at the next active edge of clock. Timing information within a cycle is not available using this technique. This technique assumes implicitly that simulation is not being done to verify timing and that timing verification will be done seperately by a static timing analyzer.

0-8186-7082-7/95 $04.00 © 1995 IEEE

FIGURE 1. Set of events in a clock cycle



To understand this concept, let us take a look at a hypothetical set of events within a certain clock cycle.

In FIGURE 1. above we have shown the clock for a simple synchronous design which is clocked at its positive edge. Thus the positive edge of clock is the active edge in the design. One clock cycle ranges from time T_A+ to time T_B. Only one active edge is included in each clock cycle. For a hypothetical simulation run, say there are 4 events e1, e2, e3, e4 that are scheduled by the event scheduler. These events occur at times T_1, T_2, T_3 and T_4 respectively.

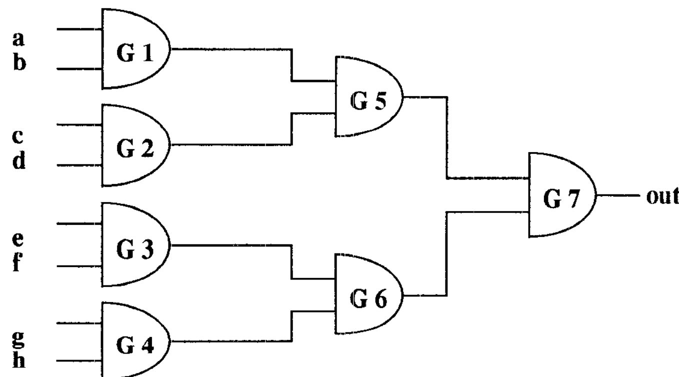
In a traditional event driven simulator, these events will be scheduled in the time wheel and evaluated at their respective scheduled times. We suggest different approach. Instead of scheduling events e1, e2, e3 and e4 at different times in the time wheel, we do the following:

1. All the events e1, e2, e3, e4 are scheduled to be evaluated at the next active edge.
2. All events are put in an event orderer which checks whether it can eliminate certain unnecessary evaluations by ordering the events correctly. The event orderer levelizes and orders the evaluations so that each element is evaluated at most once per clock cycle.
3. The active edge is the evaluation edge for each cycle. At the evaluation edge, the set of ordered events is evaluated.

The advantages of this technique are that the time wheel is sparse. All events are clubbed together at the active edges of clock. Very coarse granularity scheduling is required. Also, many unnecessary evaluations are eliminated by proper event ordering.

To prove this, let us consider an example of a simple combinational circuit in FIGURE 2.

FIGURE 2. Reducing unnecessary evaluations



Assume that the cycle time is 20 ns. Inputs a, b, c, d, e, f, g and h change at times t=2, 4, 6, 8, 10, 12, 14 and 16 respectively. Assume that there is no gate delay. When input a changes, in a traditional event driven simulator, gates G1, G5 and G7 will be scheduled to be evaluated at time t=2. Similar events will be scheduled to be evaluated at times t= 4, 6, 8, 10, 12, 14 and 16. Thus during one cycle 24 gate evaluations will take place.

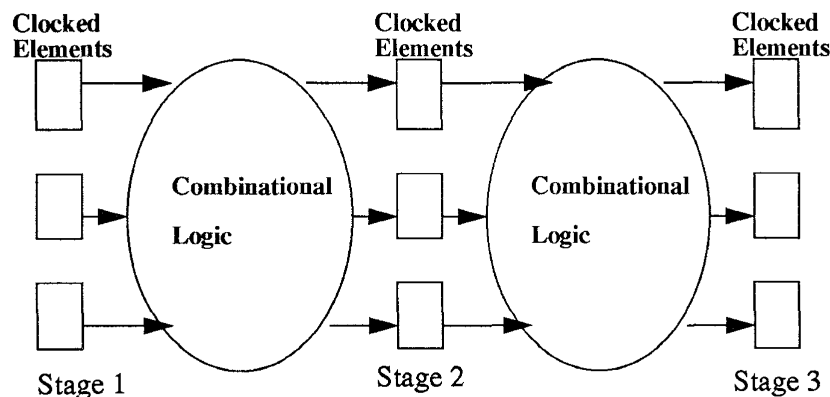
If we delay all the events till the evaluation edge, we realize that by ordering the events properly we need to evaluate each gate exactly once. Thus we have 7 gate evaluations. We save 17 unnecessary evaluations for this simple circuit in a single cycle.

Using this technique will ensure that each element in the circuit is evaluated only at the active edge of clock. Also each element is evaluated at most once during each clock cycle and is evaluated zero times if it's inputs do not change. Thus we guarantee that each element will not be evaluated more than once in a single clock cycle. Thus we eliminate unnecessary evaluations.

2.2 Cycle based oblivious simulation

The oblivious simulation technique is well understood [1]. Given a synchronous design, logic levelization is applied to the design. Logic levelization is a process of emulating the data flow from primary inputs and clocked element outputs to primary outputs and clocked element inputs. If there are any feedback loops, the levelization technique cannot be applied. After levelization is applied, each stage of the design looks like an array of latches with combinational logic in between. In FIGURE 3. we show the data flow in a levelized design.

FIGURE 3. View of a levelized design



In an oblivious simulation, the entire design is evaluated once every time unit or when inputs change. We study a modification of this technique. The entire design is evaluated once every clock cycle. One possible order of evaluation is the following.

At each stage of the design, during each clock cycle,

1. Clocked elements are clocked at the active edge of clock.
2. Combinational logic is evaluated during the cycle and the output of that logic is stable before the next active clock edge.

Thus data flows from the primary inputs to the primary outputs of the design with clocked elements and combinational logic at each stage.

3.0 Measurement techniques

Having studied the techniques for cycle simulation, it is important to estimate the performance gain obtained by applying such techniques to the existing simulators. These measurement techniques can be applied to any simulator. The important factors are:

1. Trade-offs in event driven vs. oblivious simulation.
2. Performance gain by eliminating multiple evaluations.

3.1 Event driven vs. Oblivious simulation

Two metrics are proposed to measure this trade-off. By obtaining the following measurements, it is possible to make intelligent decisions about the trade-off between event driven and oblivious simulation.

3.1.1 Activity in the design

This metric measures average percentage of design activated every clock cycle. For any design, the primary inputs will change according to the given stimulus vectors. Based on which inputs change, certain parts of the design will need to be reevaluated. This metric measure the average percentage that is evaluated every clock cycle.

For example, in a typical compiled code simulator, a number of C subroutines are created for each module to evaluate the various elements. If we treat each such subroutine as a unit of evaluation, then we can find out the number of subroutines evaluated every clock cycle.

From that we can define our measure as,

% Activity = Average # of subroutines executed per cycle X 100

Total # of subroutines in the entire design

A similar measurement can be made for an interpreted simulator by calculating the percentage of elements activated.

If the % activity is high, then an oblivious simulation approach should be chosen. If it is low, then an event driven approach should be chosen.

3.1.2 Scheduling Overhead

Scheduling overhead determines what percentage of the total run time is spent in the scheduler in an event driven simulator. This can be done by profiling the scheduler routines.

% Scheduling Overhead =

Time spent in scheduler X 100

Total time for simulation

If the scheduling overhead is high, and the activity level is high then an oblivious approach might be more suited. If the scheduling overhead is low and the activity level is low then an event driven approach will yield faster simulation.

3.2 Performance gain by eliminating multiple evaluations

Two metrics are suggested to measure the performance gain obtained by eliminating multiple evaluations of an element during a clock cycle. These metrics assume that the simulator on which these measurements are performed is currently an event driven simulator in which levelization and ordering is not being done to ensure that each element is evaluated at most once during a cycle.

TABLE 1. Routines executed for module instance A

Routine #	cycle 1	cycle 2	cycle 3	Avg. routine evals all cycles
1	2 times	0 times	1 times	----
2	3 times	0 times	1 times	----
3	0 times	3 times	2 times	----
4	1 times	3 times	2 times	----
total # of evals in the cycle	6 routine evals	6 routine evals	6 routine evals	6 routine evals

3.2.1 Estimating unnecessary evaluations

The assumption is that each element in the design should be evaluated at most once in each clock cycle. This is possible by levelizing and ordering the elements in the design. We wish to measure the inefficiency of the simulator in ordering evaluations by counting the number of elements that are multiply evaluated per clock cycle. In the case of a typical compiled code simulator, we assume that one subroutine corresponds to one element in the design. Thus we wish to measure how many subroutines are multiply evaluated per clock cycle.

To understand this metric for a compiled code simulator, assume that a verilog module A has been compiled and the compilation produces 4 subroutines for that module. Say we run this module for 3 clock cycles. We calculate TABLE 1. shown above.

Then we calculate for each cycle, the number of routines that were executed a certain number of times in TABLE 2..

From the above tables and given module A we can extract the following information.

1. Total # of subroutines in module A
   = 4
2. Avg # of subroutines evaluations per cycle
   = 6 (TABLE 1.)
3. Avg. # of subroutines evaluated 0 times per cycle
   =
   1 (TABLE 2.)

From these numbers we deduce the following:

Avg. # of subroutines evaluated *at least once* per clock cycle =

Total # of subroutines in module A -

Avg. # of subroutines evaluated 0

times per cycle

For this example we see that avg. # of subroutines evaluated *at least once* per clock cycle = 4 - 1 = 3. If we had an ideal simulator without multiple evaluations, we would see exactly 3 subroutine evaluations per cycle. But we see 6 subroutine evaluations per cycle which means there are some excess evaluations.

Hence,

Avg. # of excess evaluations per clock cycle =

Avg. # of subroutine evaluations per cycle -

Avg. # of subroutines evaluated *at least*

once per clock cycle

In this case

avg. # of excess evaluations per clock cycle=

6 - 3 = 3

% excess evaluations =

Avg. # of excess evaluations per clock

cycle X 100

Avg. # of subroutines evaluated *at least*

once per clock cycle

TABLE 2. Number of routines executed a certain number of times

# times exec	cycle 1	cycle 2	cycle 3	Avg. all cycles
0	1 routine	2 routines	0 routines	1 routines
1	1 routine	0 routines	2 routines	1 routines
2	1 routine	0 routines	2 routines	1 routines
3	1 routine	2 routines	0 routines	1 routines

For this example,

% excess evaluations = 3 X 100 / 3

= 100%

For a simulator for which there are many excess evaluations, the % excess evaluations could be much greater than 100.

3.2.2 Speedup factor

We assume that all elements are equivalent in terms of execution time. In case of a compiled code simulator, we assume that each subroutine take the same amount of CPU time. This assumption is not totally accurate but allows us to estimate the possible speedup we can obtain by reducing the number of unnecessary evaluations.

Speedup factor =

Avg. # of subroutine evaluations

per cycle

Avg. # of subroutines evaluated *at*

least once per clock cycle

In the example for module A, speedup factor = 6 / 3 = 2. Speedup factor gives an approximate number to quantify the simulation speedup that can be obtained by applying the levelization and ordering technique.

4.0 Results

We obtained the following results for 3 designs. They are tabulated in TABLE 3..

From the above results, the following points can be inferred.

1. % activity for our example designs were low. This would suggest that an event driven simulation might run faster.
2. Scheduling overhead is not so high to justify oblivious simulation.
3. About 25-40 % of the evaluated routines are multiply evaluated in a clock cycle.
4. If all elements had same execution time, we could expect a speedup by a factor of 1.25-1.40 by eliminating excess evaluations.

5.0 Conclusions

In this paper we discussed cycle simulation techniques. We proposed measurement techniques that will estimate the speedup obtained by applying cycle simulation techniques to the existing simulators. The proposed metrics will help to evaluate the various tradeoffs in choosing the correct simulation technology for your design. By drawing inferences from the results as shown in section 4, it is possible to identify the type of simulator that will run your design the fastest. The results also quantify the potential speedup due to the proposed techniques. If the gain is substantial, a move to include the proposed cycle simulation techniques in the simulator can be justified.

TABLE 3. Results for the designs

Design name	% activity	% scheduler overhead	% excess evaluations	Speedup factor
Design A	0.313 %	5-10 %	25.8 %	1.258
Design B	11.73 %	5-10 %	38.8 %	1.388
Design C	6.942 %	5-10 %	33.9 %	1.339

Further work is being done to identify areas that could potentially speed up simulation. Efforts are also being made to quantify the speedup obtained by optimizing these areas.

6.0 Acknowledgements

We wish to thank Chronologic Simulation for letting us use their simulator and for their part in valuable discussions. We wish to thank Ser-Hou Kuang and Paul Monschke for their valuable comments.

References

[1] Wang, L., Hoover, N. E., Porter, E. H., Zasio, J. J., "SSIM: A software levelized compiled-code simulator", *Proc. Design Automation Conference*, pp. 2-7, 1987.