A very abbreviated form of this paper has appeared in the India VLSI conference. I doubt that much of the material presented here has ever appeared in print.

THREE-VALUED SIMULATION WITH THE INVERSION ALGORITHM

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

ABSTRACT

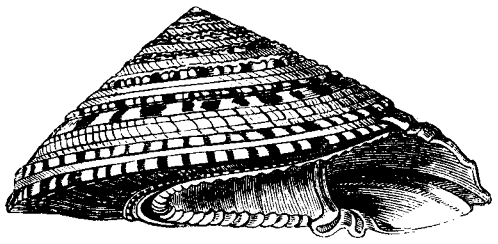
The Inversion Algorithm is an event-driven logic simulation technique that is competitive with Levelized Compiled Code Simulation. Previous versions of the Inversion Algorithm have been limited to purely binary simulation. The algorithm presented here extends the Inversion Algorithm to three-valued simulation while preserving the desirable properties of the two-valued algorithm. Because of the richer transformation structure used in three-valued simulation, the scheduling technique is significantly more complex than that of the two-valued algorithm. The procedure for collapsing simultaneous events is also significantly more complex. Once a three-valued net achieves a stable binary value, it is possible to replace the three-valued simulation with a more efficient two-valued simulation. Experimental data shows that the three-valued algorithm is also competitive with levelized compiled code simulation.

THREE-VALUED SIMULATION WITH THE INVERSION ALGORITHM

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

1. Introduction

The Inversion Algorithm[1] is an event-driven simulation technique that rivals, and sometimes exceeds the speed of levelized compiled code simulation[2-14]. The fundamental concept behind the Inversion Algorithm is that no gate will be simulated unless its output is guaranteed to change value. This condition is enforced my various means, the most important of which is a counting technique[15-17] that keeps track of the number of inputs carrying dominant values. The fact that no gate will be simulated unless its output changes value allows for several significant optimizations in the simulation process. If nets are assumed to carry only binary values, then the effect of a change in the net can be predicted without examining its value. Except for primary inputs and monitored nets, the Inversion Algorithm uses no permanent net values, and propagates neither values nor changes through the circuit. Instead, simulation consists of a wave-front of activity that proceeds from the primary inputs to the monitored nets. When this wave hits a monitored net, the value of the net toggles. An important consequence of this form of simulation is that several types of gates can be eliminated from the circuit without affecting the correctness of the simulation, and several other types of gates can be combined, significantly reducing simulation time.



Despite its efficiency, the all previous implementations of the Inversion Algorithm use a two-valued logic model. There may be situations in which a two-valued logic model will not give sufficient information to debug a circuit. The purpose of this paper is to show how the unknown value can be incorporated into the Inversion Algorithm with minimal impact on the advantageous features of the two-valued algorithm.

2. Three-Valued Simulation.

In three-valued simulation, it is assumed that each net can take the values **0**, **1**, or **U** (unknown). In most cases, the initial value of each net will be set to **U**, to allow the circuit to be properly initialized by the first input vector. The three-valued truth tables for the standard Boolean functions are illustrated in Figure 1.

AND	U	0	1
U	U	0	U
0	0	0	0
1	U	0	1

OR	U	0	1
U	U	U	1
0	U	0	1
1	1	1	1

XOR	U	0	1
U	U	U	U
0	U	0	1
1	U	1	0

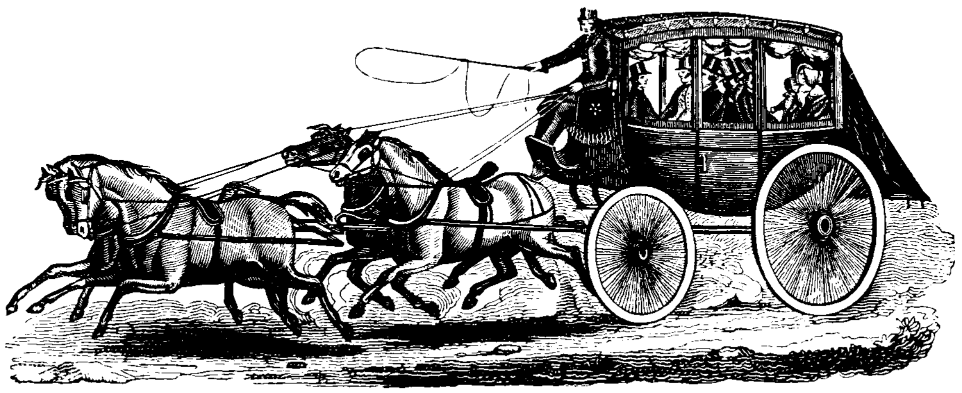
NOT	-
U	U
0	1
1	0

Figure 1. Three-Valued Truth tables.

The tables illustrated in Figure 1 illustrate the differences between two and three-valued simulation. For AND and OR gates, the single-level dominance that occurs in two-valued simulation is replaced by a two-level dominance. For AND gates, the value **0** dominates both **U** and **1**, while the value **U** dominates the value **1**. The XOR gate, which transmits all changes in two-valued simulation, is now dominated by the value **U**. The NOT gate propagates all changes.

One advantage of the two-valued inversion algorithm is that state changes are computed based on a change in a single gate input. It is not necessary to compute an *n*-ary operation on the gate inputs. This feature has a significant impact on the performance of the Inversion Algorithm. Furthermore, in two-valued simulation, it is not necessary to determine the type of change occurring in a net, because any change can be predicted from the current value of the net. Thus it is not necessary to propagate information regarding changes through the network. In three-valued simulation, two different changes may occur depending on the value of the net, so the type of change must be communicated between from one net to another.

Despite the added complexity of three-valued simulation, it is possible to use the internal state of a gate to determine whether an output change will occur. In two-valued simulation, it is sufficient to keep a count of the number of inputs having the dominant value. In three-valued simulation, it is also necessary to keep track of the number of inputs which have the unknown value. By using these two counts, it is possible to compute the output change of an AND or an OR gate without examining all inputs. Figure 2 summarizes the actions that must be taken in response to each type of input change. The variable “DC” contains the dominant count for the gate, while the variable “UC” contains the unknown count. The variable “OC” appearing in the XOR column is the ones-count which is explained in Section 4.



Change	AND	OR	XOR	NOT

0-1	**DC = DC -1
	if DC = 0 then
	Output = 0-1**
		**DC = DC + 1
		if DC = 1 then
		Output = 0-1**
			**OC = OC + 1
			if UC = 0 then
			Output = Toggle**
				Output = 1-0

0-U	**DC = DC -1
	UC = UC + 1
	if DC = 0 then
	Output = 0-U**
		**UC = UC + 1
		if DC = 0 and
		UC = 1 then
		Output = 0-U**
			**UC = UC + 1
			if UC = 1 then
			Output = CurrVal-U**
				Output = 1-U

1-0	**DC = DC + 1
	if DC = 1 then
	Output = 1-0**
		**DC = DC -1
		if DC = 0 then
		Output = 1-0**
			**OC = OC - 1
			if UC = 0 then
			Output = Toggle**
				Output = 0-1

1-U	**UC = UC + 1
	if DC = 0 and
	UC = 1 then
	Output = 1-U**
		**DC = DC -1
		UC = UC + 1
		if DC = 0 then
		Output = 1-U**
			**OC = OC - 1
			UC = UC + 1
			if UC = 1 then
			Output = CurrVal-U**
				Output = 0-U

U-0	**UC = UC - 1
	DC = DC + 1
	if DC = 1 then
	Output = U-0**
		**UC = UC - 1
		if UC = 0 and
		DC = 0 then
		Output = U-0**
			**UC = UC - 1
			if UC = 0 then
			if OC mod 2 = 1 then
			Output = U-1
			else
			Output = U-0**
				Output = U-1

U-1	**UC = UC - 1
	if UC = 0 and
	DC = 0 then
	Output = U-1**
		**UC = UC - 1
		DC = DC + 1
		if DC = 1 then
		Output = U-1**
			**OC = OC + 1
			UC = UC -1
			if UC = 0 then
			if OC mod 2 = 1 then
			Output = U-1
			else
			Output = U-0**
				Output = U-0

Figure 2. Counter Actions for Input Changes.

3. AND OR gates.

In two-valued simulation, it was sufficient to provide a single event-handler for each net, and a single type of event called **Toggle**. When an event occurs, the current event handler is replaced with the correct event handler for the next event that will occur on the net. Because there are two different events that may occur for each net in three-valued simulation, it is necessary to provide two event-handlers for each net. When an event occurs, both event handlers are replaced. There are six types of events that may occur for any net, each of which requires its own event handler. For the sake of brevity, these events and event handlers will be denoted as **DtoN**, **NtoD**, **UtoD**, **UtoN**, **DtoU**, and **NtoU**, where **D**, **N** and **U** stand for **Dominant**, **Non-dominant**, and **Unknown**. These events must be paired in the following way **(DtoN,DtoU), (NtoD,NtoU), (UtoD,UtoN)**. Since each net is initialized to **U**, each event structure will be initialized with the pair **(UtoD,UtoN)**. When this pair is replaced, it will be with one of the other two pairs listed.

As in the two-valued algorithm, each event in the three-valued algorithm represents one fanout-branch of a net. This is necessary because a **DtoN** event on one fanout branch may be an **NtoD** event on another. When an event is queued, it is necessary to specify which event-handler of the pair will be used to process the event. For simplicity, the first event handler in each of the pairs listed above will be termed the *upper* event handler, and the second will be termed the *lower* event handler. **DtoN** and **NtoD** events always queue the upper event handler, while **DtoU** and **NtoU** events always queue the lower event handler.

This procedure presents some problems when processing **UtoD** and **UtoN** events. Consider the situation illustrated in Figure 3.

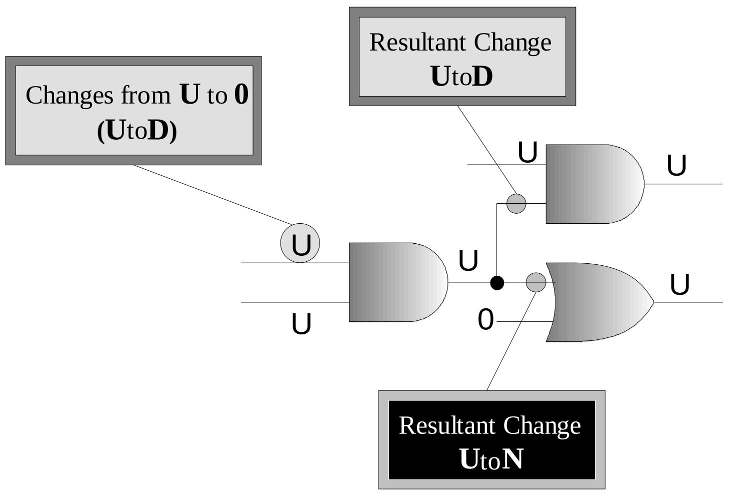


Figure 3. Conflicting Changes in Fanout Branches.

In Figure 3, the **UtoD** event occurring at the input of the first AND gate must schedule a UtoN event at the input of the OR gate, and a **UtoD** event at the input of the second AND gate. If it is necessary to dynamically compute the type of event-handler at run time, simulation performance could be severely impaired. This problem is solved by providing two sets of upper/lower pairs, **(UtoD,UtoN)** and **(UtoN,UtoD)**. Appropriate use of these two pairs will allow the **UtoD** event to *always* queue the upper event handler, and the **UtoN** event to always queue the lower event handler. The assignment of the event pairs depends on whether a connection is a homogeneous or a heterogeneous connection.

Reference [1] defines homogeneous and heterogeneous connections in terms of the way changes to the dominant counts propagate from one gate to another. Homogeneous and heterogeneous connections are connections between two gates of the types AND, OR, NAND, or NOR, possibly with some intervening NOT gates. Suppose G1 and G2 are gates taken from these four gate-types, and that the output of G1 is the input of G2. If a change in the input of G1 is propagated to the input of G2, it is necessary to compute a new dominant count for both gates. If the dominant counts move in the same direction (i.e., both are incremented or both are decremented) then the connection is homogeneous. If the counts move in opposite directions, the connection is heterogeneous. The homo/heterogeneous property is a property of fanout branches, not of entire nets. If a connection is homogeneous, then the **(UtoD,UtoN)** upper/lower pair is used, while if the connection is heterogeneous, the **(UtoN,UtoD)** pair is used. For primary input connections, those attached to AND and NAND gates are treated as homogeneous, while those attached to OR and NOR gates are treated as heterogeneous. This allows the choice of upper or lower event handler in the input processing routine to be independent of gate-type.

The upper/lower pairs automatically load the correct upper/lower pair to handle the next event. The DtoN event handler loads the pair **(NtoD,NtoU)**, and the **NtoD** event handler loads the pair **(DtoN,DtoU)**. The event handlers **DtoU** and **NtoU** load the default pair, **(UtoD,UtoN)** or **(UtoN,UtoD)**, for the connection. The event handler **UtoD** loads the pair **(DtoN,DtoU)**, and the event handler **UtoN** loads the pair **(NtoD,NtoU)**.

It is important to verify that this scheme for queuing events will allow consecutive events to interact properly. For each of the event types, it is necessary to verify that the event handler will select the appropriate event handler for any propagated events. The following four lemmas establish the correctness of the event propagation procedures. These lemmas are expressed in terms of AND and OR gates to simplify the terminology. It is further assumed that the output of each gate is a single homogeneous connection. Extending these lemmas to NAND and NOR gates and heterogeneous connections is a straightforward exercise which is left to the reader.

Lemma 1. *For* AND *and* OR *gates, neither an* **DtoU** *nor an* **NtoU** *event can propagate an event other than another* **NtoU** *or* **DtoU**.

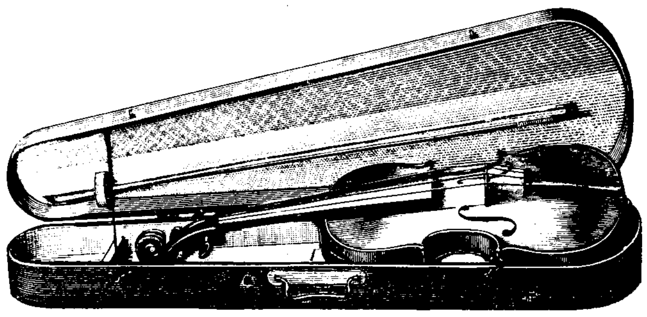
Proof. Suppose a **DtoU** event on the input of a gate causes a change on the output of a gate. The original value of the gate must be the dominant value, which will cause the output to be the dominant value as well. Therefore if the **DtoU** causes an a change in the output, it must be either a **DtoN** event or a **DtoU** event. Since the value **U** dominates the non-dominant value, the new value, **U**, of the input will prevent the output from changing to the non-dominant value. Therefore the only possible output event is **DtoU**. Now suppose that an **NtoU** event on an input causes a change in the output of a gate. No other input can have the dominant value, otherwise no change would be possible. Therefore, after the event, all inputs have the non-dominant or the **U** value. Because the **NtoU** event causes a change, no other input can have the **U** value. This implies that the original value of the gate must have been the non-dominant value, and the new value must be the **U** value. Therefore, the only event that can be caused by an **NtoU** event is another **NtoU** event.■

Lemma 1 verifies the correct handling of propagated events when the lower event handler of the pairs **(DtoN,DtoU)** and **(NtoD,NtoU)** is executed. If the event is propagated, the event-handler pair of the output must also be one of **(DtoN,DtoU)** and **(NtoD,NtoU)**. When propagating an event, the event handlers for DtoU and NtoU events always select the lower event handler. Lemma 1 verifies that this behavior is correct. Lemmas 2 and 3 concert the upper half of the **(DtoN,DtoU)** and **(NtoD,NtoU)** pairs.

Lemma 2. *For* AND *and* OR *gates, an* **NtoD** *event can propagate only another* **NtoD** *or a* **UtoD** *event.*

Proof: Since the event propagates, the output of the gate cannot be the dominant value. If the output is the non-dominant value, then an **NtoD** event is propagated, otherwise a **UtoD** event is propagated.■

When propagating an event, the **NtoD** event handler always selects the upper event handler for the output. If the output value is **U**, then the upper event handler will be **UtoD**. If the output is set to the non-dominant value, then the upper event handler will be **NtoD**. As Lemma 2 illustrates, the **NtoD** event handler always chooses the correct event handler for the propagated event.



Lemma 3. *For* AND *and* OR *gates, a* **DtoN** *event can propagate only another* **DtoN** *or a* **DtoU** *event.*

Proof: The output must have the dominant value before the event is processed. If the event propagates, no other input can have the dominant value. The other inputs must consist of non-dominant values and **U**'s. If some other input has the value **U**, then a **DtoU** event occurs, otherwise a **DtoN** event occurs.■

The event handler for the **DtoN** event can choose either the upper or lower event handler based on the state of the gate. However, as Lemma 3 shows, when an **DtoN** event propagates, the event-handler pair for the output must be **(DtoN,DtoU)**, permitting a correct choice to be made for all states.

Lemma 4. *For* AND *and* OR *gates, neither a* **UtoD** *event nor an* **UtoN** *event can propagate an event other than another* **UtoD** *or* **UtoN**.

Proof: Because the value of the input is **U** before the **UtoD** or **UtoN** event is processed, the only allowable output values are **U** and the dominant value. If the output value is the dominant value, then some other input must be set to its dominant value. For a **UtoD** event, this would imply no change in the output, preventing the event from propagating. If the output value is the unknown value, then the only types of events that can propagate are **UtoN** and **UtoD**.■

When the **DtoN** and **NtoD** event handlers, they assume that they are propagating an identical event to the output, thus **DtoN** events always queue the upper event handler, and NtoD events always queue the lower event handler. (If the connection is heterogeneous, reversing the **(UtoD,UtoN)** pair takes care of moving from the dominant to the non-dominant value.) Lemma 4 shows that this procedure is guaranteed to produce correct behavior.

4. XOR and NOT gates.

Because the XOR gate no longer propagates all events, the event handlers for XOR gates are no longer trivial. The value **U** will dominate the XOR gate preventing any events on the known inputs from propagating. Furthermore, when the inputs of the XOR gate become known, the type of event propagated will depend on the number of inputs that have the value 1, which we will call the *Ones-Count*. Although the ones-count is not needed when all inputs have known values, the possibility of an input becoming unknown requires that the ones-count be maintained at all times. Figure 2 details the event-handling procedure for XOR gates.

Six event handlers are required, **1to0**, **0to1**, **1toU**, **0toU**, **Uto0**, and **Uto1**. These events are paired in the expected fashion, **(1to0,1toU)**, **(0to1,0toU)**, and **(Uto1,Uto0)**. As with AND and OR gates, the **(Uto1,Uto0)** pair is actually two pairs, **(Uto1,Uto0)** and **(Uto0,Uto1)**. AND-XOR connections are treated as heterogeneous, while OR-XOR connections are treated as homogeneous. (Since different event handlers are involved, the choice is a matter of taste.) The **1to0** and **0to1** handlers always queue the upper event handler for propagated events, while the **0toU** and **1toU** handlers always queue the lower event handler. The handlers **Uto1** and **Uto0** may queue either the upper or lower event handler, depending on whether the ones-count of the gate is an even number.

The correctness of the XOR event-handling procedure can be established using lemmas similar to those presented in the previous section. To avoid duplicating these arguments, these lemmas are left as an exercise for the interested reader.

As in the two-valued algorithm, the handling of NOT gates is trivial. NOT gates simply pass any incoming event through to the output. It is important, however, to note that a NOT gate transforms a homogeneous connection to a heterogeneous connection, and vice versa. Figure 4 illustrates how one determines the type of each connection for a chain of NOT gates. This Figure makes it clear that NOT gates can be collapsed out of the simulation simply by changing connection types. NAND, NOR and XNOR gates are handled by treating them as AND, OR and XOR gates followed by a collapsed NOT gate.

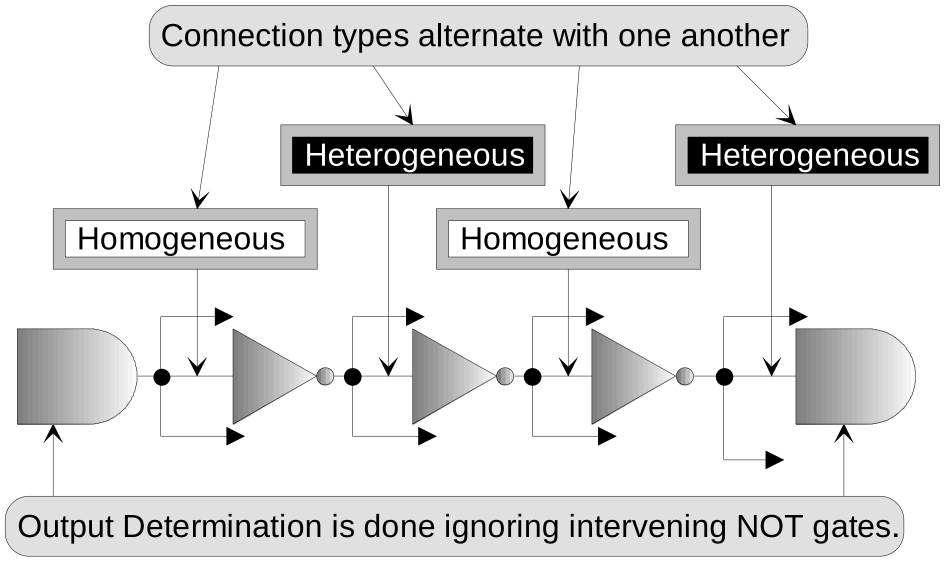

Connection types alternate with one another
Heterogeneous
Heterogeneous
Homogeneous
Homogeneous
Output Determination is done ignoring intervening NOT gates.

Figure 4. A Chain of Not Gates.

5. Collapsing Simultaneous Events.

Another aspect of the Inversion Algorithm that leads to its high performance is the collapse of consecutive events. If two consecutive simultaneous events result in no change in the affected net, then the two events are removed from the event queue, and neither is processed. In two-valued simulation, this procedure is simple because any two consecutive events cancel one another. When an event is propagated, a new event will be queued if none is already queued. If an event is already queued, the existing event will be removed from the queue.

As should be expected, collapsing events in three-valued simulation is somewhat more complicated than in two-valued simulation. It is important to remember that simultaneous events on a net are caused by simultaneous events on the inputs of a gate propagating to the output of the gate. To determine how to collapse events it is necessary to understand the effect of simultaneous events on the state of the gate. When an event of a certain type is queued, it is necessary to know what types of events can already be queued for the net, and how this event will combine with the new event. It is also important to note that event compression will already have been performed for the events on the inputs of a gate, so at most one event will be processed for each input net at any specific time. Figure 5 illustrates all possible event-collapsing actions. Note that certain combinations of New Event, Old Event and Current Output Value can never occur.

New Event	Current Output	Prev. Event:	Actual Change	Collapse Action:
NtoD	Non-Dominant	**DtoN**	D to D	Delete Queued Event
	Non-Dominant	**UtoN**	U to D	Replace Lower event handler in queued event with Upper event handler.
	U	**NtoU**	N to D	Replace Lower event handler in queued event with Upper event handler.
	U	**DtoU**	D to D	Delete Queued Event.
DtoN	No prior event can be queued.			
NtoU	Non-Dominant	**DtoN**	D to U	Replace Upper event handler in queued event with Lower event handler
	Non-Dominant	**UtoN**	U to U	Delete Queued Event
DtoU	No prior event can be queued.			
UtoD	**U**	**DtoU**	D to D	Delete Queued Event
UtoN	**U**	**DtoU**	D-to-N	Replace Lower event handler in queued event with Upper event handler.

Figure 5. Event Collapsing for AND and OR Gates.

The primary problem encountered in event collapsing is **NtoD** event processing when the output of the gate is **U**. This is the only case in which it is not possible to determine the correct action by examining the current state of the gate. In this case it is necessary to know whether the **U** output value was caused by a **DtoU** or an **NtoU** event. Although it may be possible to determine the correct behavior by analyzing the nature of the queued events, a more efficient procedure is to add a "last event" field to the state of the gate, and modify the **DtoU** and **NtoU** event-handlers to store a "D" or an "N" in this field whenever they queue an event. This code can be used by the **NtoD** event handler to determine whether to delete the currently queued event or to replace it with a **toggle** event.

New Event	Current Output	Prev. Event:	Actual Change	Collapse Action:
0to1	0 or 1	**0to1**	None	Delete Queued Event
	0 or 1	**1to0**	None	Delete Queued Event
	0 or 1	**Uto1**	Complement of previous	Swap Upper and Lower event
	0 or 1	**Uto0**	Complement of previous	Swap upper and lower events.
1to0	Same as 1 to 0			
0toU	0 or 1	**0to1**	Previous to U.	Replace upper event handler in queued event with lower event handler
	0 or 1	**1to0**	Previous to U	Replace upper event handler in queued event with lower event handler
		Uto0	None	Delete Queued Event
		Uto1	None	Delete Queued Event
1toU	Same as 0 to U.			
Uto1	No prior event can be queued.			
Uto0	No prior event can be queued.			

Figure 6. Event Collapsing for XOR Gates.

Event collapsing for XOR gates requires a knowledge of the type of event that created the existing event. The "last event" field must be used to record this information. The **0to1** events record a "T" (for toggle) in this field, while the **Uto1** and **Uto0** events record a "U".

6. Transformation from Three to Two Values.

Although the Inversion Algorithm's three-valued event processing is efficient, it is more complex and time consuming than two-valued processing. It is possible to speed up the simulation by observing that although all nets are initialized to the value **U**, for most circuits, most nets will be permanently set to a known binary value by the first input vector. Even highly sequential circuits are normally initialized with a special reset-sequence that will cause most nets to have binary values after a few cycles. Most nets in the circuit cannot be set to the **U** value once they have achieved a known value. In many cases even though it is possible to set a net to the **U** value, this action should be treated as an error. In these cases it is possible to replace the three-valued event handlers with their two-valued counterparts, thereby speeding up the simulation.

The underlying assumption that permits three-to-two valued transformation is that some nets are *known-to-be-binary*. When a known-to-be-binary net achieves a known value, the **U** value is no longer needed for the net. This property is especially important for primary inputs and for the outputs of certain flip-flops. When a primary input is known-to-be-binary, this implies that every input vector will supply a known binary value for the input. If the output of a flip-flop is known-to-be-binary, the output should be initialized to a known value, and the internal circuitry and usage of the flip-flop should guarantee that, under normal operating conditions, the output never oscillates or achieves a metastable state. The known-to-be-binary property can be propagated through a combinational network by observing that when the inputs of a combinational gate are known, the output is known as well.

Because all nets are initialized to the **U** value, all nets must initially be treated as trinary nets. Although the transformation from trinary to binary could be done simultaneously with event processing, this would unnecessarily complicate the processing of all three-valued events. A better method is to perform the transformation as a separate step which is isolated from normal event processing. The mechanism for performing this transformation is to propagate a special event called a *Meta-Event*, through the network. Conceptually, a Meta-Event is created whenever a known-to-be-binary net is first assigned a known value. Each gate has an associated count of untransformed input nets. When a meta-event is processed for a net, the event-handler is replaced with a binary event-handler, and the count of each fanout gate is decremented. When the count reaches zero, a new meta-event is generated for the output net of the gate.

Meta-Event processing will be performed between input vectors, and as Lemma 5 shows, it is sufficient to perform this process once after the processing of the first input vector.

Lemma 5. *Let* C *be a circuit with known-to-be-binary primary inputs. Assume further that no other net has the known-to-be binary property. The known-to-be-binary property will not propagate to any net after the first input vector is processed.*

Proof. If it is possible for the known-to-be-binary property to propagate to a net, then it must be possible for all inputs of the gate to be assigned the known-to-be-binary property. These nets must either be primary inputs, or outputs of gates whose inputs can themselves be assigned the known-to-be-binary property. This implies that any net that can be assigned the known-to-be-binary property must be dependent only on the primary inputs of the circuit. This also implies that there can be no cyclic path between the net and the primary inputs. If a net depends only on the primary inputs, and if all primary inputs have been assigned known values, then it is possible to compute a known value of the net using only the known values of the input vector. After the first input vector is processed, the correctness of the simulator implies that the known value will have been computed for the net. Therefore, if it is possible to propagate the known-to-be-binary property to the net, the net will have been assigned a known value during the processing of the first input vector.■

Lemma 5 is also applicable to sequential circuits containing flip-flops with known-to-be-binary outputs, as long as the outputs of these flip-flops are initialized with known values. If the circuit contains flip-flops with known-to-be-binary outputs that are initialized to the **U** value, then Lemma 5 does not apply. In this case, it may be necessary to perform Meta-Event processing several times.

For sequential circuits, Lemma 5 implies that Meta-Event processing will only be partially successful. This is a reflection of reality, because when a circuit contains feedback loops, it is usually possible to generate unknown values, either due to “don’t care” conditions in the circuit, or due to design flaws. In either case, the propagation of a **U** value to certain nets should be treated as an error. Suppose that it is technically possible to propagate an unknown value to net **N**, but propagating such values is to be treated as an error. For such nets it is possible to perform a “weak” transformation from trinary to binary. The upper event handler will be replaced with a binary event-handler, just as if the net were assigned the known-to-be-binary property, but the lower event handler will be preserved and replaced with a routine which generates an error message alerting the designer to the error. Since the **U** value is an error, the behavior of the gate in response to the value is unimportant, and no event will be propagated to the outputs. Correct behavior is no longer guaranteed once this condition occurs. It is possible to transform the net back to a trinary net, and propagate this transformation as far as necessary, but because the circuit has ceased to function correctly this is probably wasted effort.

7. Experimental Data.

We have implemented two versions of the three-valued inversion algorithm, one which performs a full three-valued simulation for all vectors, and one which uses meta-events to perform three-to-two valued conversion after the first input vector. We have compared these algorithms to various other algorithms using the ISCAS85 combinational benchmarks[18]. Experiments were all run on the same, dedicated machine, a SUN IPC with 12 Megs of memory and an internal hard disk. The results of these experiments are reported in Figure 7. The same data is presented graphically in Figure 8. The numbers are expressed in terms of CPU Seconds of execution time. These numbers do not include the time required to read input vectors or write output vectors. Five thousand randomly generated vectors were used for each simulation. The input-activity rate (percentage of primary inputs that change on each vector) is approximately 50% for all vector sets. Each experiment was performed five times and the results were averaged to obtain the results illustrated in Figure 7. In addition to comparing the Inversion Algorithm to two-valued LCC simulation, a special three-valued oblivious LCC simulator was constructed to allow comparison of the three-valued algorithm with three-valued LCC simulation. The two-valued LCC simulation results were obtained from the FHDL LCC simulator, which has been used here to support both CAD tool development and VLSI research for several years.

Circuit	2-Val Inv.	3-Val Inv.	3-2 Conv.	Not-Elim	2-Val LCC	3-Val LCC	Activity
c432	1.7	3.2	2.4	2.2	0.5	0.8	59.4
c499	2.0	4.9	2.9	2.5	0.6	1.0	63.2
c880	3.8	7.2	5.7	5.1	1.2	2.3	57.1
c1355	6.5	10.6	8.3	7.4	1.9	3.4	56.5
c1908	8.1	16.2	11.9	7.6	4.4	6.1	56.8
c2670	17.7	33.2	25.9	20.0	5.3	16.2	55.7
c3540	16.5	30.3	22.4	17.2	8.4	24.8	52.4
c5315	36.9	61.8	47.6	35.5	21.7	38.9	63.8
c6288	40.4	67.0	54.6	52.0	30.1	48.5	61.5
c7552	52.6	88.0	67.1	53.4	40.7	58.6	60.7

Figure 7. Raw Experimental Data.

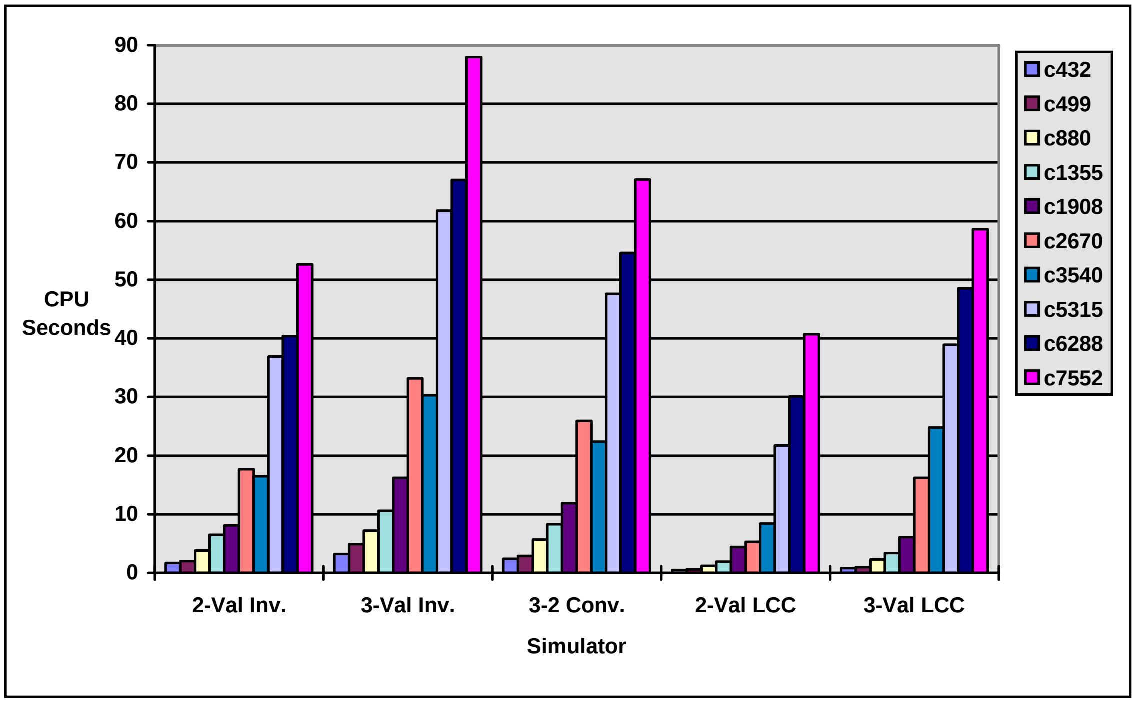


Figure 8. Graph of Experimental Data.

Several observations about the three-valued inversion algorithm can be made from this data. First, the full three-valued Inversion Algorithm requires approximately twice as much simulation time as the two-valued algorithm. This is to be expected, because the simulation code is roughly twice as large. (However, as in the two-valued algorithm, the amount of run-time code is minuscule.) For the three-to-two valued conversion, the simulation times are roughly comparable. It must be emphasized that neither of the three-valued simulators included the NOT-Elimination or the Connection Collapsing optimizations described in reference [1]. In spite of the lack of optimizations, the three-valued algorithm, with meta-events, has outperformed three-valued LCC simulation for one circuit, c3540. Finally, it is important to note that the activity rates of these circuits are much larger that would probably be encountered in practice. As activity rate declines, the performance of the Inversion Algorithm will improve proportionally, while the performance of the LCC algorithm will remain constant. Figure 9 illustrates how the performance of the three-valued Inversion Algorithm improves as activity rate decreases to more realistic values. The data of Figure 9 was obtained by running circuit C7552 on several vector sets with differing rates of input activity. The data of Figure 9 is presented graphically in Figure 10.

Input Activity:	5%	10%	15%	20%
3-Val Inv.	16.4	28.0	38.0	44.9
3-2 Conv.	12.6	21.4	28.8	33.9
3-Val LCC	58.5	58.5	58.5	58.5

Figure 9. C7552 with various input activity rates.

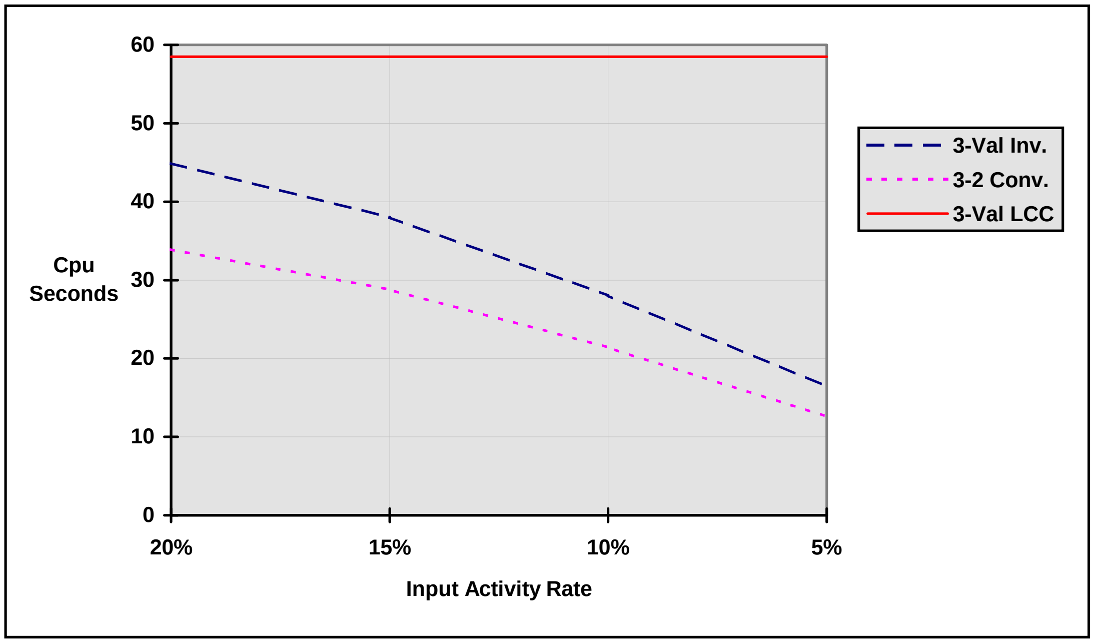


Figure 10. Graph of decreasing input activity.

8. Conclusion.

One important benefit of the three-valued Inversion Algorithm is its unconditional ability to handle asynchronous sequential circuits. Although the two-valued algorithm is technically capable of handling such circuits, it requires that all nets in the circuit be initialized to consistent values. For combinational and synchronous circuits, the initialization can be done by simulating the circuit with a zero vector at compile time. For asynchronous circuits, a sequence of vectors may be required to initialize the circuit properly, and automatically generating the correct sequence at compile time is a difficult task. The three-valued algorithm will function correctly without a compile-time simulation, permitting asynchronous circuits to be handled as easily combinational and synchronous circuits. (The implementations described in Section 7 do not include compile-time simulations.)

Work is proceeding on the three-valued algorithm. Our highest priority is to create a version of the algorithm that includes all optimizations described in [1]. These optimizations are identical in the two and three-valued algorithms. Work is also proceeding on unit and multi-delay Inversion simulators that include the three-valued algorithms described here.

The Inversion Algorithm is a new, unique approach to simulation that we believe will be extremely important in constructing high-speed simulators for tomorrow’s large scale integrated circuits. Although the early implementations of the Inversion Algorithm have not included every feature required for by a commercial simulator, our on-going research continues to show that these features can be incorporated with an acceptably small impact on its performance. We believe that the existing work has shown the Inversion Algorithm to be a viable alternative to other more traditional simulation techniques.

9. References.

1. P. Maurer, “The Inversion Algorithm for Digital Simulation,” *ICCAD-94*, pp. 258-61.
2. R. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.
3. D. M. Lewis, " A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.
4. Chiang, M., R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.
5. W. Y. Au, D. Weise, S. Seligman, "Automatic Generation of Compiled Simulations through Program Specialization," *Proceedings of the 28th Design Automation Conference*, 1991, pp. 205-210.
6. A. W. Appel, "Simulating Digital Circuits with One Bit Per Wire," *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.
7. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715
8. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator", *Proceedings of the 24th Design Automation Conference*, 1984, pp. 473-478.
9. Z. Barzilai, J. L. Carter, B. K. Rosen, J. D. Rutledge, "HSS -- A High Speed Simulator," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 4. July 1987, pp. 601-617.
10. Maurer, " Two new techniques for unit-delay compiled simulation," *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 9, pp. 1120-1130, Sept. 1992.
11. Y. Lee, P. Maurer, “Two New Techniques for Compiled Multi-Delay Simulation,” *Proceedings of Southeastcon 92*, Apr, 1992.
12. Y. Lee, P. Maurer, "Two New Techniques for Compiled Multi-Delay Logic Simulation," *Proceedings of the 29th Design Automation Conference,* 1992, pp. 420-423.
13. P. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," *IEEE Transactions on Computer Aided Design*, in press.

14. Z. Wang, P. Maurer, " LECSIM : A Levelized Event Driven Compiled Logic Simulator", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 491-496.
15. D. Schuler “Simulation of NAND Logic,” *Proceedings of COMPCON 72,* Sept. 1972, pp. 243-245.
16. M. Breuer, A. Friedman *Diagnosis and Reliable Design of Digital Systems,* Computer Science Press, Rockville, MD, 1976.
17. M Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing, and Testable Design,* Computer Science Press, New York, 1990.
18. Brglez, P. Pownall, R. Hum, “Accelerated ATPG and Fault Grading via Testability Analysis,” *International Conference on Circuits and Systems*, 1985, pp. 695-698.