

Chapter 8

The Inversion Algorithm

8.1 Introduction

In logic simulation, there are two things that affect performance, the number of gates simulated, and the speed of the individual simulations. Most techniques discussed so far address the problem of speeding up the individual simulations. Event-Driven simulation is the exception because it addresses the problem of reducing the number of gate simulations.

Event-Driven simulation does not eliminate all useless simulations. If a gate is simulated and its output doesn't change, the simulation was useless. If a gate is simulated its output changes, but its output is never used, or never affects the value of a monitored signal, the simulation was useless. Simulations that do not result in changes in the gate output are called *useless simulations of the first kind*, while simulations that produce changes that do not propagate to a monitored net are called *useless simulations of the second kind*.

The Inversion Algorithm is designed to eliminate useless simulations of the first kind. When an event is processed for a net, a pre-computation is done to determine whether the event will cause a change in the output of the net. If no change will occur, the gate is not simulated. The trick is to make the pre-computation more efficient than actually simulating the gate. It turns out that this is fairly easy to do.

8.2 AND and OR Gates

AND and OR functions exhibit the phenomenon of dominance. In both cases, there is an input value that will force the output of the gate to a known value regardless of the value of the other inputs. This value is known as the *dominant value of the gate*. For AND and OR gates, it is possible to keep a count of how many inputs have the dominant

value, and update the count for every change in input. If the count changes from one to zero or from zero to one there will be a change in the output. Otherwise the output will remain constant.

If one looks only at the changes in a net the value of the net alternates between one and zero. AND and OR gates will view the net as alternating between the dominant and non-dominant value.

Interestingly enough, the changes in a gate-input are completely predictable from its initial value. A dominant to non-dominant change always follows a dominant to non-dominant change, and vice-versa. As long as one keeps track of the next type of change that will occur in a net, it is not necessary to maintain a value for the input, or test the value of the input for dominance. The change in the output is completely determined by the dominant count of the gate, and the fact that a change of a specific type happened on a single input.

8.3 NOT Gates and XOR Gates.

The output of a NOT gate will change whenever its input changes. Similarly the output of an XOR gate will change whenever one of its inputs changes value. Dominant counts are unnecessary. The only problem is dealing with the output of an XOR when two or more inputs change value. Of course, this is also a problem with AND and OR gates. It is possible for an AND gate to undergo two transitions as illustrated in xxx. The net effect of both transitions will be to leave the output of the AND gate unchanged. It is true for all multiple-input gates, that it is necessary to deal with the possibility of two or more changes in the input which cancel each other out, even though either change would cause an output change if it occurred in isolation.

The way to handle multiple input gates is to use event cancellation. When an input change triggers a change in the output, an event will be queued for the gate output, unless there is already an event queued for the output. If there is already an event queued for the output, the event will be cancelled, and no change will occur.

8.4 Net Values.

For gate inputs, it is not necessary to test net values for dominant counts. For gate outputs, it is not necessary to compare net values to determine whether an event will be queued. In fact, net values are not needed at all, except for primary inputs, and nets that are monitored by the user. For monitored nets, it is necessary to maintain net values, so the value can be shown to the user. Net values are maintained by inverting the net value every time a change occurs. That's how the Inversion Algorithm got its name.

For primary inputs it is necessary to maintain the current value of the net to determine whether a change occurs on a primary input when a new vector is read. Changing net values can be done as if it were an event, so gate simulation is entirely eliminated and replaced with event processing.

Once net values and gate processing is eliminated, AND and NAND gates become the same. The only difference between an AND and a NAND gate is the value of the output. Since output values have been eliminated, there is no difference left. The same for

OR/NOR, NOT/BUFFER, and XOR/XNOR pairs. The inversion algorithm does not distinguish between the members of these pairs.

Eliminating net values also has the effect of making AND gates and OR gates be the same. The difference between AND and OR gates can be couched in terms of inverting net values, as illustrated in xxx. Once net values have been eliminated, the inverted inputs and outputs are no longer different from the non-inverted inputs and outputs, so AND and OR gates are now the same.

8.5 Eliminating NOT and BUFFER Gates.

When a the input of a NOT gate changes, an event is processed. The only action taken as a result of processing this event is to schedule an event for the output of the NOT gate. It stands to reason that one could save time by simply scheduling the second event in the first place. Doing this effectively deletes the NOT gate from the circuit. Since the Inversion algorithm does not distinguish between NOT and BUFFER gates, BUFFER gates can be eliminated in the same way. NOT and BUFFER gates can simply be dropped from the circuit.

8.6 Eliminating XOR Gates.

Just as NOT and BUFFER gates can be eliminated, so can XOR gates. The only thing that happens when an input event is processed for an XOR, is that an event is scheduled at the output. Eliminating XOR gates can cause mechanical scheduling problems for the gates that feed the eliminated gate, since some nets may need to be scheduled while others are descheduled.

8.7 Homogeneous and Heterogeneous Connections.

It's obvious that if I combine two AND gates as illustrated in xxx, that the resultant function can be computed using a single AND gate, with all of the inputs of both gates combined into a single gate as illustrated in xxx. In a conventional simulator, it is debatable whether the combined gate can be simulated any more efficiently than the two independent gates. However, in the Inversion Algorithm the simulation of the combined gate is more efficient, because net events are no more complicated in the combined gate, and events have been eliminated for the connection between the two gatesd.

The Inversion Algorithm also allows gates to be combined in ways that are not possible in a conventional simulator. Although AND and OR gates are the same, it is still not possible to combine an AND and an OR gate as illustrated in xxx. The reason this combination won't work is because in the AND-AND combination a change that propagates through both gates increments (or decrements) the counts of both gates. In the AND-OR combination, one count is incremented, while the other is decremented.

Connections between two gates of the type AND, NAND, OR, and NOR can be characterized as homogeneous or heterogeneous depending on whether the counts of the two gates move in the same direction or opposite directions when an event propagates through both gates. All Homogeneous connections can be collapsed in the same way as

an AND-AND connection. There are four types of homogeneous connections, AND-AND, OR-OR, NAND-OR, and NOR-AND. In the NAND-OR connection, it is important that the NAND gate come first, followed by the OR. Similarly for the NOR-AND connection. For all four types of connections, it is possible to replace the second gate with its complemented-output equivalent. An OR can be replaced with a NOR, and an AND can be replaced with a NAND, but only if it is the second gate. In all four cases, the connection can be collapsed by simply expanding the size of the second gate, and adding the inputs of the first gate to the second gate. xxx illustrates this.

The other connections, AND-OR, OR-AND, NOR-OR, and NAND-AND are heterogeneous connections. (The second gate can be replaced with its complemented output equivalent.)

8.8 Eliminating Heterogeneous Connections

Although heterogeneous connections can't be collapsed as easily as homogeneous connections, it's still possible to partially eliminate them. Linear collapsing can be done by assigning fractional values to certain inputs. Layered collapsing can be done by using more than one counter per input, and incrementing and decrementing the counters in series. An event propagates only if all counters are triggered. xxx illustrates the first, xxx illustrates the second.

8.9 Introduction.

Of all the tools available to the modern VLSI designer, simulation is probably most important. The cost of fabricating a VLSI design is so high, that it is necessary to verify and debug the product before committing it to silicon. Despite steady improvements in simulator performance, it is not unusual for a VLSI designer to spend more time on simulation than on any other activity. There are many different styles of simulation, from high-level simulation at the algorithmic level, to electrical simulation using systems of differential equations. As a general rule, the more detailed the simulation, the more time consuming it becomes. Logic simulation represents a compromise between the extremes of algorithmic simulation and electrical simulation. Although more time consuming than algorithmic simulation, it is efficient enough to be used as a primary debugging tool. While it is not as detailed as electrical simulation, the logic gates that comprise the logic model can be mapped one-to-one into the electrical components of the final product.

Over the past several years, there has been a steady flow of papers describing new more efficient methods of logic simulation[**Error! Reference source not found.-Error! Reference source not found.**]. This research makes it obvious that there are two methods for improving the performance of logic simulation: speed up the simulation of individual gates, or simulate fewer gates. Until now, these two methods have worked at cross-purposes to each other. Some simulators have used relatively complex algorithms for reducing the number of gates simulated, thereby increasing the simulation time for each gate. Other simulators have improved the speed of individual gate simulations by reducing or eliminating scheduling code, thereby increasing the number of gate simulations that must be performed for each input vector. When all scheduling code is

eliminated the simulation time for each input vector becomes constant, and is no longer dependent on changes in the inputs. Such simulators are termed *Oblivious*. In contrast, simulators whose performance varies from one input vector to another are termed *Event-Driven* [Error! Reference source not found.]. (This term is used for simplicity, and does not necessarily imply that the simulator processes events.)

The simplest form of oblivious simulation is Levelized Compiled Code (LCC) simulation, in which each gate in the circuit is simulated once per input vector. Many of these gate simulations produce no useful output, because they do not cause a change in any output net. (Any net visible to the user can be considered an output net, regardless of whether it is an actual output of the circuit.) Event-driven simulation is the most common method for eliminating useless simulations. In a typical event-driven simulation, an event is generated whenever a net changes value. A gate is simulated if and only if an event occurs on one of its inputs. Although this technique eliminates many useless simulations, it does not eliminate all of them. Even if the inputs of a gate change, this change may not propagate to an output net. This can occur in two ways. First, the change in the gate-input may not produce a change in the gate output. This situation is illustrated in Figure 1. Second, the change may propagate through the gate, but be “absorbed” by some other gate before reaching an output net, as illustrated in Figure 2.

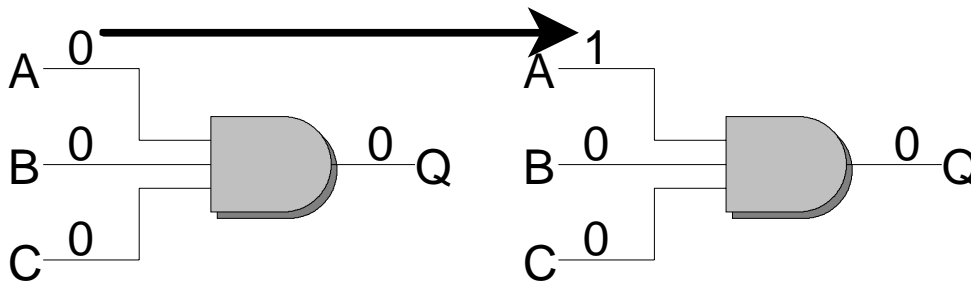


Figure 1. A Useless Simulation

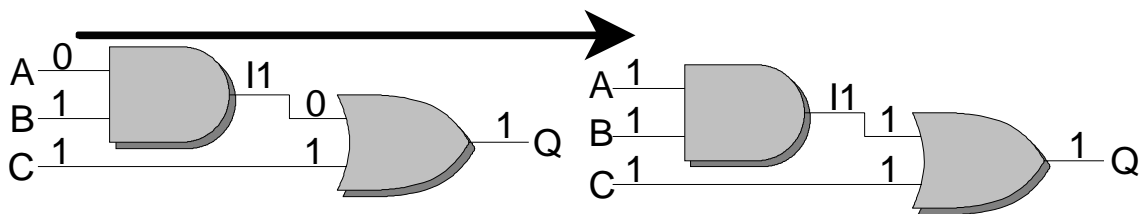


Figure 2. An Unpropagated Change.

The *Inversion Algorithm* is able to eliminate all useless simulations of the first kind, and some useless simulations of the second kind. The problem of eliminating *all* useless simulations is currently under study.

8.10 An Overview of The Inversion Algorithm.

Because the aim of the Inversion Algorithm is to eliminate gate simulations that produce no change in the output of a gate, it was designed around the underlying

principle that no gate should be simulated unless its output is guaranteed to change value. Thus, when an event occurs on the input of a gate, it is necessary to determine whether the change will propagate through the gate, and suppress the gate-simulation if no propagation will occur. It is not immediately clear that making such a determination is any more efficient than simply simulating the gate, but it turns out that this is indeed the case.

When an event occurs on a gate input, the Inversion Algorithm performs a set of tests to determine if the event will propagate through the gate. The tests must be individualized for different gate-types, but the number of different kinds of tests that must be performed is surprisingly small. In its most basic form, the Inversion Algorithm supports the eight gate types AND, NAND, OR, NOR, XOR, XNOR, NOT, and BUFFER. These gate-types provide all essential functions and can be used as building blocks to construct more complex gate-types. However, it is not necessary to provide specific tests for each of these eight types. One set of tests is provided for NOT and BUFFER gates, a second set of tests is provided for XOR and XNOR gates, and a third set of tests is provided for AND, NAND, OR, and NOR gates.

The tests for the XOR, XNOR, NOT, and BUFFER gates are trivial, because any change in an input implies a change in the output. An identical set of tests could be used for all four gate-types, but because XOR and XNOR gates have more than one input, it is possible to propagate two or more simultaneous events through the gate. Because two consecutive simultaneous events cancel one another, the tests for XOR and XNOR have been optimized to eliminate consecutive events on the gate output. When an event propagates through the gate, the algorithm tests the queue to determine whether there is already an event queued for the net. If so, the existing event is removed from the queue, and no new event is queued. Since NOT and BUFFER gates have a single input, no test for collapsed events is necessary. The event-handlers for NOT/BUFFER gates and XOR/XNOR gates are illustrated in Figure 3.

```
NOT/BUFFER:  
  Schedule OutputEvent;  
  
XOR/XNOR:  
  if OutputEvent Not Scheduled Then  
    Schedule OutputEvent;  
  else  
    Deschedule OutputEvent;  
  Endif
```

Figure 3. Event Handlers for NOT/XOR.

The tests for AND, OR, NAND and NOR are more complex and are based on the counting algorithm originally described by Schuler[Error! Reference source not found.,Error! Reference source not found.,Error! Reference source not found.]. The counting algorithm, which is illustrated in Figure 4, was originally intended for use in a conventional event-driven simulation. In Figure 4 it is assumed that there has been a change in an input X to the gate G. The dominant value, 1 for OR/NOR, and 0 for AND/NAND, is a parameter to the algorithm.

```

If Value.of.X = Dominant.Value.of.G Then
    Count.of.G := Count.of.G + 1;
    If Count.of.G = 1 Then
        Output.of.G := Dominant.Value.of.G;
    EndIf;
Else
    Count.of.G := Count.of.G - 1;
    If Count.of.G = 0 Then
        Output.of.G := NOT Dominant.Value.of.G;
    EndIf;
EndIf;

```

Figure 4. The Original Counting Algorithm.

The counting algorithm of Figure 4 assigns a value to the output of G if and only if the output changes value. This algorithm is extremely efficient because it uses the value of a single input and an internal state to compute the value of the output, rather than computing a function using all input values. The counting algorithm used by the Inversion Algorithm is used to predict changes rather than compute output values, and is much simpler than the algorithm shown in Figure 4. One reason for the simplification is the underlying scheduling technique, which is based on the shadow algorithm[**Error! Reference source not found.**]. In the shadow algorithm, each event is represented by the structure pictured in Figure 5.

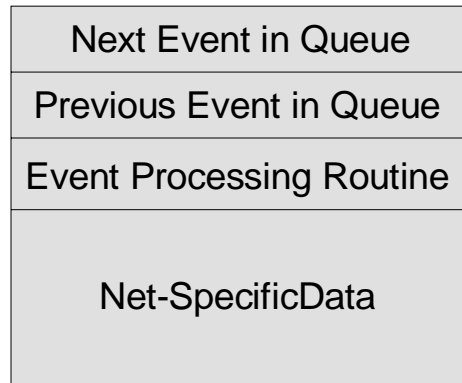


Figure 5. An Event Structure.

The Inversion Algorithm generates a dedicated event structure for each input net, each of which contains an indirect pointer to the event-processing routine for the net. The use of indirect pointers allows event handlers to be changed at run time. In the counting algorithm, the test for dominant values can be eliminated by observing that successive events on an input net will cause it to alternate between its dominant and non-dominant value. The Inversion Algorithm uses two event-handlers, one for dominant values and one for non-dominant values. When the dominant-value event-handler executes, it replaces the Event-Processing-Routine address in the event structure with the address of the non-dominant event handler. The non-dominant-value event handler performs similarly. The two event handlers execute in strict alternating fashion for each input net, with no test for dominant value required. The event-processing for AND, NAND, OR

and NOR gates uses the same event-collapsing procedure as XOR and XNOR gates. The event-handlers for dominant and non-dominant values are illustrated in Figure 6.

```
Dominant:  
Decrement DominantCount;  
If DominantCount=0 Then  
  If OutputEvent Not Scheduled Then  
    Schedule OutputEvent  
  Else  
    Deschedule OutputEvent  
  EndIf  
EndIf  
EventProcessingRoutine := AddressOf NonDominant;  
  
NonDominant:  
Increment DominantCount;  
If DominantCount=1 Then  
  If OutputEvent Not Scheduled Then  
    Schedule OutputEvent  
  Else  
    Deschedule OutputEvent  
  EndIf  
EndIf  
EventProcessingRoutine := AddressOf Dominant;
```

Figure 6. The AND/OR Event Processors.

The event handlers of Figure 3 and Figure 6 do not contain separate code for scheduling gate simulations. Because no gate is scheduled for simulation unless its output is guaranteed to change value, gate simulations are reduced to simple inversion operations. (This assumes that a two-valued logic model is being used. The Inversion Algorithm supports more complex logic models, but this is beyond the scope of this paper[**Error! Reference source not found.**].) Surprisingly, because the correct operation of the Inversion Algorithm does not require net-values, it is possible to eliminate most gate simulations entirely. The event handlers pictured in Figure 3 and Figure 6 do not need to test net values to schedule new events. There are only two cases where net-values are required by the Inversion Algorithm. Net values are required for all primary inputs, because it is necessary to compare new and old net values when a new input vector is read. It is also necessary to maintain net values for any net visible to the user so that correct output values can be printed after an input vector has been simulated. Since the processing of events does not depend on net values, gate simulations can be performed during event processing without affecting the correctness of the algorithm. Thus, when the output of a gate is visible to the user, the event handlers will schedule a special event to invert the value of the output.

The elimination of net values has some surprising consequences which are worth noting. Inverted outputs and non-inverted outputs are identical. Therefore, for simulation purposes AND is identical to NAND, OR is identical to NOR, XOR is identical to XNOR, and NOT is identical to BUFFER. Because the increment and decrement operations are performed with respect to dominance rather than specific values, AND and OR gates are also identical for simulation purposes.

In a sense, the Inversion Algorithm performs *no* traditional gate evaluations, but simply processes a series of events. These events differ in one important way from the events that occur in traditional event-driven algorithms. In traditional event-driven simulation, each event corresponds to a change in a single net, while in the inversion algorithm each event corresponds to a change in a single fanout branch of a net. Thus a single event in a traditional event-driven algorithm may correspond to several events in the Inversion Algorithm. Figure 7 illustrates why this is necessary.

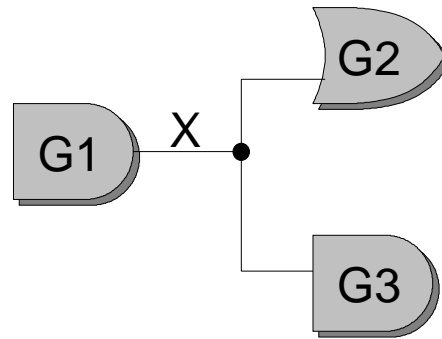


Figure 7. A Circuit Fragment.

In Figure 7, **X** is the output of gate **G1**, and the input for **G2** and **G3**. However, the dominant value for **G2** is the non-dominant value for **G3**, and vice-versa, so when an increment operation is performed for **G2**, a decrement operation must be performed for **G3**. Although both of these operations could be performed during the processing of a single event, it is more straightforward to treat them as separate events. This implementation style also facilitates the incorporation of inversion events for computing required net values. In Figure 7, if net **X** were visible to the user, the simulator would add a third event to compute the value of **X**.

Initialization for the Inversion Algorithm is somewhat more complex than for more conventional simulation algorithms. Although the dominant, non-dominant sequence is predictable for the inputs of AND, NAND, OR and NOR gates, it is necessary to commence the simulation with the correct event handler. As Figure 8 illustrates, a simple default is not sufficient to guarantee correct operation of the algorithm.

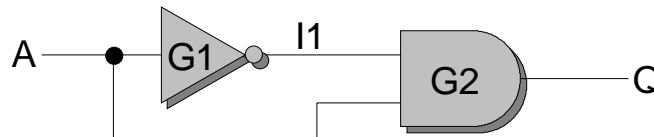


Figure 8. Complex Initialization Requirements.

In Figure 8, any time net **A** changes to the dominant value, net **I1** changes to the non-dominant value, and vice-versa. It is necessary to initialize the simulation in such a way as to guarantee that any time the non-dominant event handler is used for net **A**, the dominant event-handler will be used for net **I1**. The determination of this requirement cannot be made simply by examining gate **G2**. It is necessary to examine the entire circuit to determine the correct initialization state for the inputs of **G2**. To determine the correct initialization for all gate inputs, a single simulation is performed at compile time

to determine consistent values for all nets in the circuit. These net values are then used to determine the initial event handler to be used for each fanout branch, and the initial dominant count for each gate. (The three-valued Inversion Algorithm eliminates the preliminary simulation step, but this is beyond the scope of this paper.)

8.11 Implementation Details.

The Inversion Algorithm consists of two major phases, the Translation Phase which prepares the circuit for simulation, and the Simulation Phase which performs the simulation. The primary function of the Translation Phase is to prepare the data structures used by the Simulation Phase. Most current implementations of the Inversion Algorithm also generate run-time code, but this code could just as easily be loaded from a library of precompiled routines.

The first step in the translation phase is to parse the circuit description, and translate it into internal data structures. Once this has been done, the circuit is leveled, the gates of the circuit are sorted into leveled order, and each gate is simulated once to generate a set of consistent values. Next, a SIMULATION fanout branch is added to each net visible to the user. Finally, a data structure known as a shadow is generated for each fanout branch of each net in the circuit.

Figure 9 illustrates the structure of a shadow. The **next** and **previous shadow** fields are used to link the shadow into the event list. The event list is doubly linked to facilitate dequeuing of events. The **subroutine** field points to the event processing routine for this fanout branch. The **first** and **last fanout branch** fields contain pointers to the first and last shadow that will be scheduled when an event propagates through the gate. All fanout branches of a net are scheduled and descheduled simultaneously. To make this process more efficient, all shadows for a net are statically linked during the translation phase. This allows all shadows for a net to be inserted into the queue or deleted from the queue as a unit.

The **lock address** field contains the address of the dominant count for the gate associated with the fanout branch. For NOT, BUFFER, XOR and XNOR gates, this field is unused. Finally, the **queue address** identifies the queue into which the shadow is to be inserted.

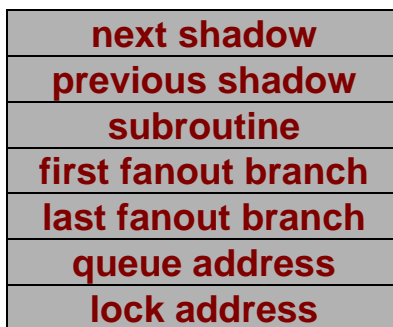


Figure 9. The Structure of a Shadow.

Eight different event processors are used during the simulation phase of the Inversion Algorithm. These occur in pairs and are called INCREMENT, INCREMENTX, DECREMENT, DECREMENTX, NOT, NOTX, XOR and XORX. The second routine of each pair is used for shadows that are at the end of a subchain, while the second is used for the other shadows. The two subroutines of each pair are identical, except that the second routine removes the subchain from the queue. The routines were created in pairs to allow dequeuing to be performed without a conditional test. These routines are more detailed versions of the algorithms presented in Figure 3 and Figure 6.

Most existing implementations of the Inversion Algorithm use the zero-delay timing model, and are based on the LECSIM simulator developed by Wang[**Error! Reference source not found.**]. (Unit-delay implementations of the Inversion Algorithm exist, but are beyond the scope of this paper.) LECSIM is a zero-delay event-driven levelized compiled code simulator. In LECSIM, gates are levelized and a queue is created for each level in the circuit, including the zero level. When a gate is queued for simulation, it is placed in the queue that corresponds to its level. Queues are processed in order by level. For asynchronous cyclic circuits, queues may be processed more than once.

Like LECSIM, the zero-delay Inversion Algorithm levelizes the circuit and creates one event queue per level. Each queue consists of a doubly linked list of shadows terminated by a special shadow known as the queue-trailer. The queue-trailer is responsible for advancing the simulation from one queue to the next and for terminating the simulation when appropriate. Figure 10 illustrates the structure of the queue. As Figure 10 illustrates, the queue headers are organized as an array of pointers, each of which points to a doubly linked list of shadows.

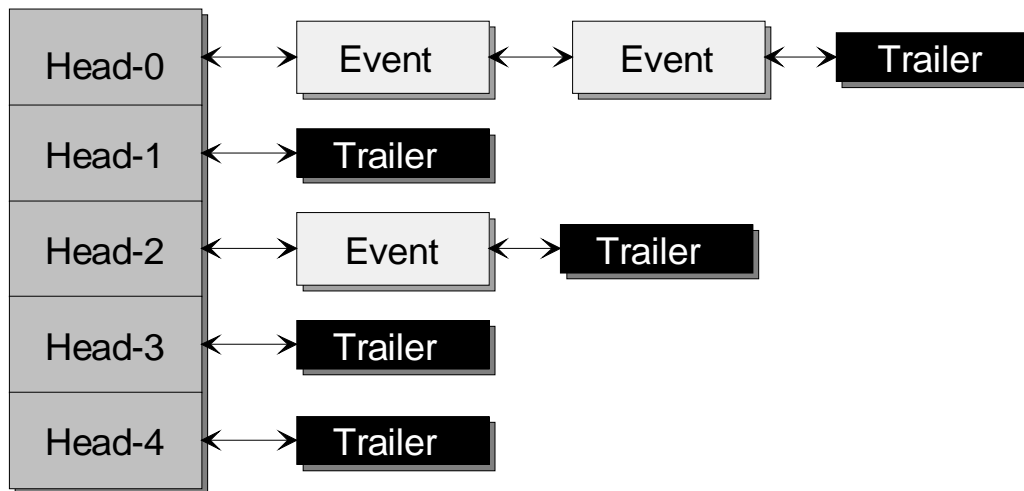


Figure 10. The Structure of the Simulation Queue.

The simulation of an input vector begins with the primary input tests. The value of each primary input is compared with the value from the previous vector (or with zero for the first vector) and if there is a change, the fanout branches of the primary input are inserted into queue zero. Once all primary input tests are complete, the simulator loads the address of the first shadow in queue zero into the **current-shadow** register and branches to the subroutine address contained in the shadow.

Design Automation: Logic Simulation

When an event is processed, additional shadows may be inserted into other queues. Once the last queue has been processed, simulation of the current vector terminates and a new vector is read. Prior to reading the new vector, the value of each net visible to the user is printed. Figure 11 gives the code for the INCREMENTX routine.

```
INCREMENTX:
Current_Shadow->subroutine = &DECREMENTX;
(*Current_Shadow->Lock)++;
if ((*Current_Shadow->Lock) == 1)
{
  if (Current_Shadow->first_fanout->next == NULL)
  {
    Current_Shadow->last_fanout->next =
      Current_Shadow->Queue->next;
    Current_Shadow->Queue->next->previous =
      Current_Shadow->last_fanout;
    Current_Shadow->Queue->next =
      Current_Shadow->first_fanout;
    Current_Shadow->first_fanout->previous =
      Current_Shadow->Queue;
  }
  else
  {
    Current_Shadow->last_fanout->next->previous =
      Current_Shadow->first_fanout->previous;
    Current_Shadow->first_fanout->previous->next =
      Current_Shadow->last_fanout->next;
    Current_Shadow->last_fanout->next = NULL;
  }
}
Temp = Current_Shadow->next;
Current_Shadow->next = NULL;
Current_Shadow = Temp;
Goto *Current_Shadow->subroutine;
```

Figure 11. The INCREMENTX event handler.

8.12 Optimizations of the Inversion Algorithm.

There are several simple optimizations that can significantly increase the performance of the Inversion Algorithm. The most important of these are the elimination of NOT and BUFFER gates, the elimination of XOR and XNOR gates, and the collapsing of homogeneous and heterogeneous connections.

8.12.1 The Elimination of NOT and Buffer Gates.

As the event-handler of Figure 3 illustrates, the processing of NOT and BUFFER gates is a no-op operation in the Inversion Algorithm. When the input of a NOT or a BUFFER gate is processed, the only action that is taken is scheduling the fanout branches of the output of the gate. The same simulation result can be achieved by eliminating the scheduling of NOT and BUFFER inputs and scheduling their fanout branches instead.

Figure 12 illustrates this procedure. For the Unit-Delay and Multi-Delay timing models, special procedures are required to preserve the delay of the gate.

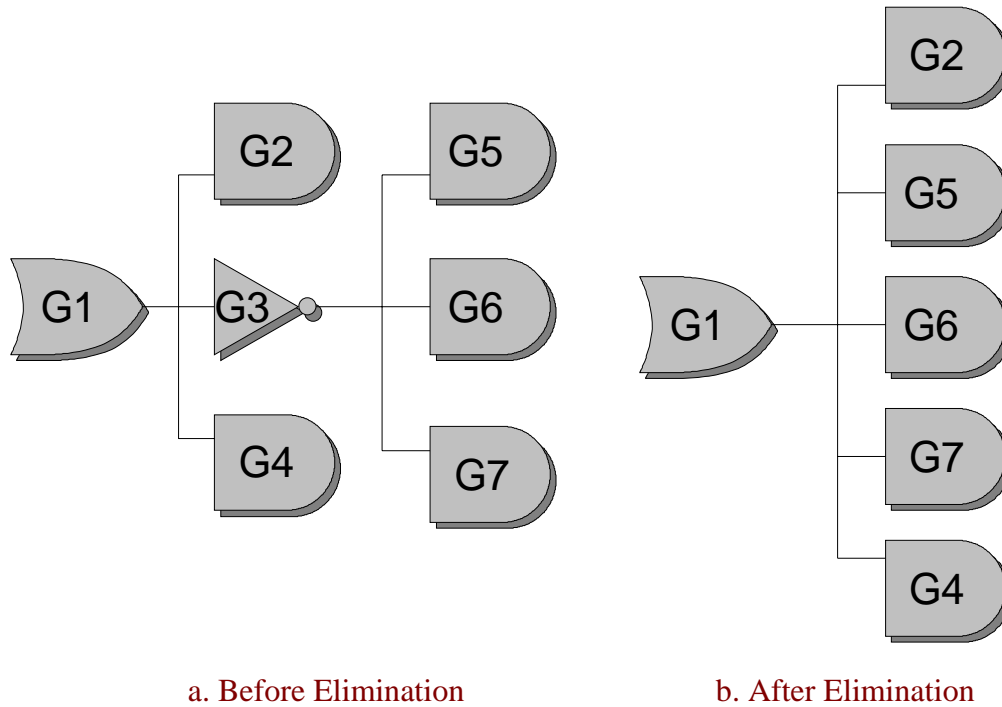


Figure 12. The elimination of a NOT gate.

8.12.2 The Elimination of XOR and XNOR gates.

It is also possible to eliminate all XOR and XNOR gates. As with NOTs and BUFFERS, the only action taken when processing the input of an XOR or XNOR is scheduling or descheduling the fanout branches of the gate. One can eliminate the processing of the input branch by scheduling or descheduling the output branches of the gate instead. This can interfere with block scheduling of fanout branches as Figure 13 illustrates.

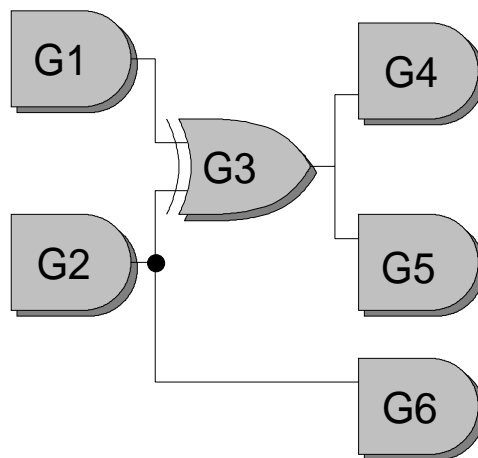


Figure 13. The Elimination of XOR Gates.

Suppose that gate **G3** of Figure 13 has been eliminated, and that events propagate through both **G1** and **G2**. Assume that the event for **G1** is processed first. When the event for **G1** is processed, it is necessary to schedule events for the inputs of **G4** and **G5**. When the event for **G2** is processed, it is necessary to *deschedule* the events for the inputs of **G4** and **G5**, and schedule an event for the input of **G6**. If events are to be collapsed properly, the fanout branches of **G3** must maintain their identity, and cannot be grouped with the fanout branches of **G1** and **G2**. This problem arises only if one or more XOR input nets fan out to other gates. One could simply ignore event collapsing in these situations, but because of the relative rarity of XOR and XNOR gates, XOR/XNOR elimination has not been implemented in any current realization of the Inversion Algorithm.

8.12.3 Homogeneous and Heterogeneous Connections.

Once all NOT, BUFFER, XOR and XNOR gates have been eliminated from the circuit, all fanout branches other than primary inputs and outputs must be connections between AND, OR, NAND and NOR gates. These types of connections can be further categorized as *homogeneous* and *heterogeneous* connections. To distinguish between the two, suppose that net **A** is the output of **G1** and the input of **G2**, and suppose that an event on the input of **G1** propagates to **A**. This will cause the dominant counts of both gates to change. If both counts are incremented or both are decremented, then the connection is homogeneous, otherwise it is heterogeneous. Because a net may fan out to different types of gates, the heterogeneous and homogeneous properties apply to fanout branches rather than to entire nets. It is possible to categorize connections at compile time using the tables illustrated in Figure 14 and Figure 15. When using these tables, it is necessary to do the categorization *before* NOT gates are eliminated. An intervening NOT gate changes a heterogeneous connection to a homogeneous connection, and vice-versa. Two consecutive NOT gates cancel one another. A connection that passes through an XOR/XNOR gate cannot be categorized as either heterogeneous or homogeneous, because the dominant-counts of the two gates will sometimes move in the same direction and sometimes move in opposite directions.

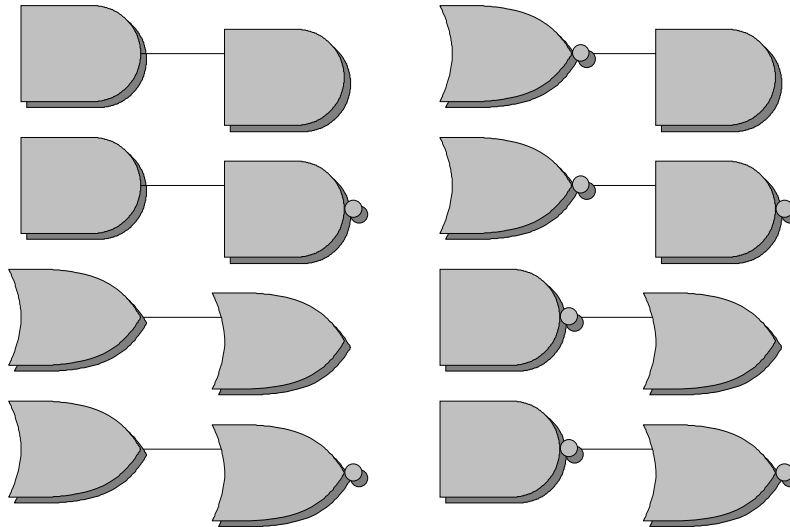


Figure 14. Homogeneous Connections.

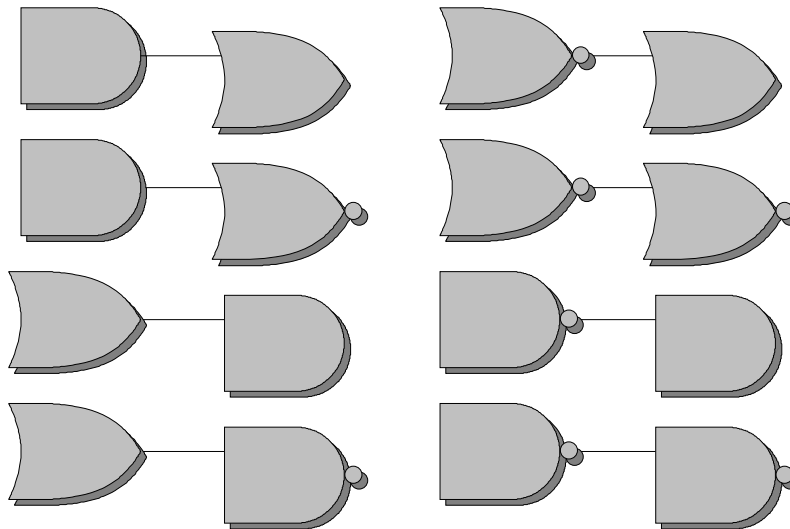


Figure 15. Heterogeneous Connections.

8.12.4 The Elimination of Homogeneous Connections.

It is possible to eliminate all homogeneous connections from a circuit using the procedure illustrated in Figure 16. To eliminate the connection **I1**, one simply removes gate **G1**, and treats the inputs **A**, **B**, and **C** as if they were inputs of **G2**. The initial value of the dominant-count for **G2** is recomputed by adding the initial dominant-counts of **G1** and **G2** and subtracting one.

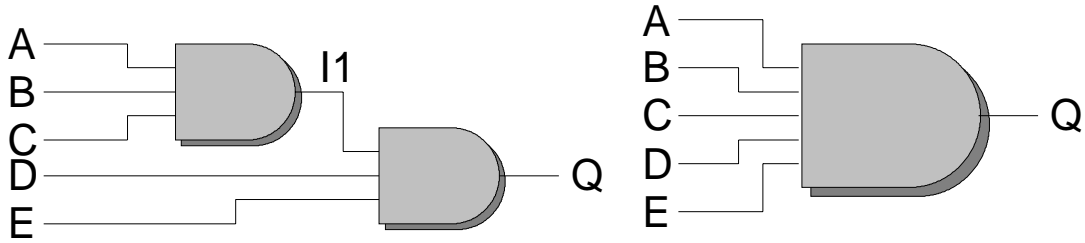


Figure 16. Eliminating a Homogeneous Connection.

The operation illustrated in Figure 16 could be performed conventional simulator, but the benefits are not as clear. After the collapse, any event on inputs **A**, **B**, and **C** would result in the simulation of a 5-input gate, regardless of whether the event would have propagated to **I1** in the uncollapsed circuit. In the Inversion Algorithm, the processing of events on the inputs **A**, **B**, and **C** is identical in both circuits, and all processing for net **I1** is eliminated in the collapsed circuit. Even if collapsing of heterogeneous connections proved to be beneficial in a conventional simulation, the ability to collapse connections is limited to those types illustrated in column 1 of Figure 14. The connections in column 2 and connections with intervening NOT gates would pose a problem.

8.12.5 Eliminating Heterogeneous Connections.

The procedures for eliminating heterogeneous connections are more complex than those for homogeneous connections, and do not always eliminate all operations for the connection. There are two procedures for eliminating heterogeneous connections, the *linear method*, and the *layered method*. The layered method allows more connections to be eliminated, but retains more operations for eliminated connections.

As Figure 17 illustrates, the linear method can eliminate *only one* input from any gate. It is possible to collapse either **G2** or **G3** into **G4**. However, once **G2** has been collapsed into **G4**, it is no longer possible to collapse **G3** into **G4**. It is, however, possible to collapse gates in linear fashion by first collapsing **G1** into **G2**, and then collapsing the **G1/G2** combination into **G4**.

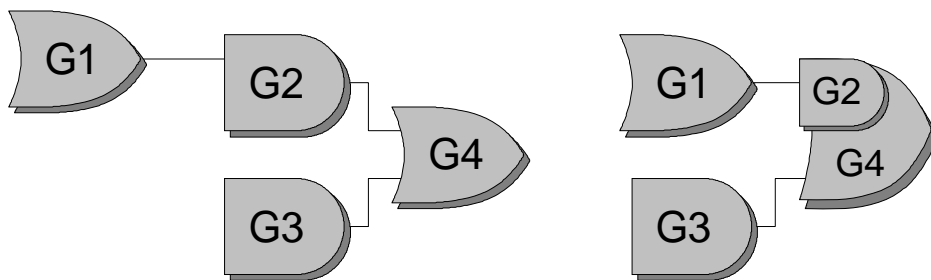


Figure 17. Eliminating a Heterogeneous Connection.

The linear method operates by changing the increment and decrement values used to update the dominant count of a gate. Instead of using a uniform value of one, the linear method uses different values for different inputs. Consider the collapsed gate illustrated in Figure 18.

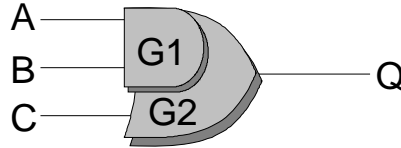


Figure 18. A Collapsed AND-OR Connection.

In Figure 18, a single dominant-count will be maintained for the combined gate **G1/G2**. Suppose that all three inputs, **A**, **B**, and **C**, have the value 0. The dominant-count corresponding to this input state is zero. For the output of **G1/G2** to change, it is necessary for **C** to change or for *both* **A** and **B** to change. The total effect of both changes in **A** and **B** must equal the effect that **G1** would have had in the original circuit, and must also equal the effect that **C** has on the collapsed gate. Neither **A** nor **B** by itself can have enough effect on the dominant-count to cause a change in the output of **G1/G2**. Because of the symmetry of the circuit, the effect of **A** and **B** must be the same.

There are many ways to assign increment/decrement values to the input nets **A**, **B** and **C** that will achieve these requirements. One acceptable procedure is to assign the value 1 to the input **C**, and 0.5 to the inputs **A** and **B**. The output **Q** changes when the dominant count changes from a value less than 1 to a value greater than or equal to one, or when it changes from a value greater than or equal to one to a value less than one.

Collapsed connections with more than two levels follow the same principles as two-level collapsed connections. Values are assigned to inputs depending on their relative power to change the output of the collapsed gate, and there are many different assignments that will achieve the desired results. Figure 19 illustrates a three- and a four-level collapsed gate.

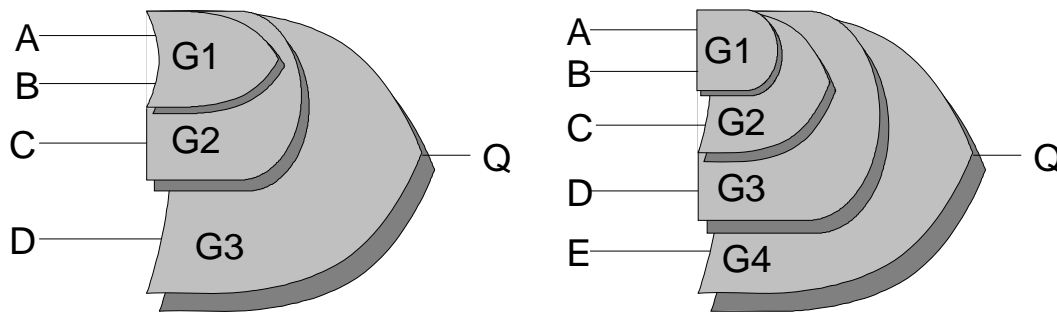


Figure 19. Multi-Level Collapsed Connections.

The values for the inputs of gate **G1/G2/G3** can be calculated in the following manner. Assume that **A**, **B**, **C**, and **D** have been initialized with the logic value of 0. The increment of **D** is set to 1. The total effect of the **G1/G2** combination must be equal to the effect of **D**. Since **A** and **B** are symmetric inputs, the increments assigned to **A** and **B** should be equal. Since the output of **G2** changes from 0 to 1 (thereby changing the output of **G1/G2/G3**) when **C** changes to 1 and either **A** or **B** changes to a 1, the sum of the increments assigned to **A** and **C** must equal 1. Since a change in both **A** and **B**, without an accompanying change in **C**, will not cause the output of **G1/G2/G3** to change, it is necessary that the sum of the increments assigned to **A** and **B** be less than 1. This implies that the increment assigned to **C** must be greater than the increments assigned to **A** and **B**.

To achieve these requirements, an increment of .25 is assigned to both **A** and **B**, while an increment of .75 is assigned to **C**. Using similar principles, the increments assigned to the inputs of **G1/G2/G3/G4** are **A**->.125, **B**->.125, **C**->.25, **D**->.75, **E**->1. As with the two level collapsed gate, the output changes value only when the gate count changes from a value less than one to a value greater than or equal to one, or from a value greater than or equal to one to a value less than one.

The layered method of collapsing heterogeneous connections allows arbitrary collapsing of connections, as illustrated in Figure 20, but requires more computation in the simulation phase than the linear method.

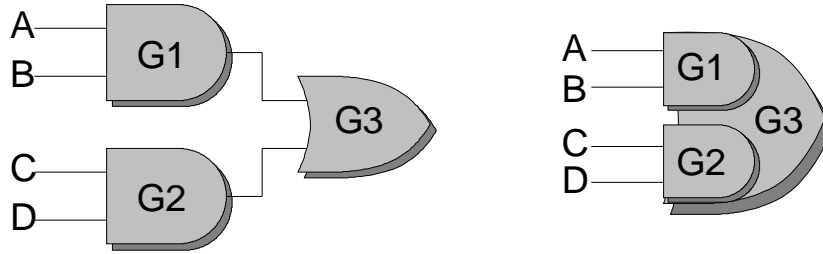


Figure 20. Layered Collapsing of Connections.

Unlike the linear method, which uses a single dominant-count for each gate, the layered method preserves the dominant-counts of the original gates. Because the original dominant counts are preserved, the number of run-time increment/decrement operations is not reduced, nor is the number of tests reduced. However the operations are performed hierarchically without any intermediate scheduling, which improves run-time performance. To illustrate assume that in Figure 20, input **A** of the collapsed gate of changes from 0 to 1. The dominant-count of **G1** is decremented, and if the new value is not zero, no further processing is done. However if the new value *is* zero, then the dominant-count of **G3** is incremented. If the new value is 1, then the fanout branches of **G3** are scheduled. No scheduling is done for the fanout branches of **G1** or **G2**.

The shadow of a layered connection differs from that illustrated in Figure 9 in that the **Lock** component of the shadow is an array of pointers rather than a single pointer. Figure 21 illustrates the code for a two-level layered connection. This code corresponds to the increment processor of a simple connection. As is the case for simple connections, both increment and decrement processors are used, the increment and decrement processors alternate with one another.

```

INCREMENT_LAYERED_2:
Current_Shadow->subroutine = &DECREMENT_LAYERED_2;
(*Current_Shadow->Lock[1])++;
if ((*Current_Shadow->Lock[1]) == 1)
{
    (*Current_Shadow->Lock[0])--;
    if ((*Current_Shadow->Lock[0]) == 0)
    {
        if (fanouts not on queue)
        {
            insert fanouts into queue;
        }
        else
        {
            remove fanouts from queue;
        }
    }
}
Current_Shadow = Current_Shadow->next;
Goto *Current_Shadow->subroutine;

```

Figure 21. The INCREMENT Routine for Two-Level Connections.

8.12.6 When to Collapse Connections.

Although it is possible to collapse all homogeneous and heterogeneous connections in a circuit, it is not always advantageous to do so. Consider the two circuits illustrated in Figure 22.

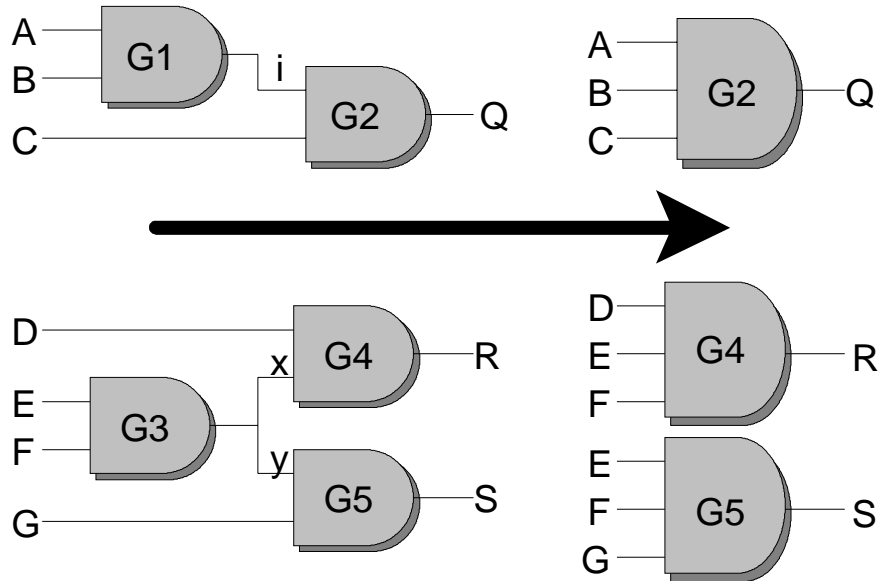


Figure 22. Collapsing Gates with Fanout.

In the first circuit of Figure 22, the net *i* does not fan out, so the connection can be collapsed by moving the inputs *A* and *B* of *G1* to *G2*. In the second circuit, the output of *G3* fans out into two branches *x* and *y*. To collapse this net it is necessary to move the two inputs *E* and *F* to both *G4* and *G5*. Because the Inversion Algorithm requires one

event per fanout-branch, this doubles the number of events that must be processed when either **E** or **F** changes value. In the original circuit, a change in net **E** will cause one event to be processed, while in the collapsed circuit two events will be processed. For the first circuit, a change in net **A** will cause one event to be processed in both the original and the collapsed circuit.

It is not immediately clear that the elimination of the events on **x** and **y** won't offset the effect of doubling the inputs **E** and **F**. To further characterize the situation, the average number of events on the inputs and outputs of **G3** were calculated, assuming all input vectors to be equally likely. For the uncollapsed connection, the average number of events on **E**, **F**, **x**, and **y** is 1.75 events per input vector. After collapsing the connection, the average number of events on **E** and **F** is 2.00 events per input vector. On the average, the uncollapsed connection will be more efficient. Under the same assumptions, the average number of events on **A**, **B**, and **i** is 1.375 for the uncollapsed connection and 1.000 for the collapsed connection. In this case the collapsed connection is more efficient.

Extending this analysis to gates with larger numbers of inputs and larger fanouts, an uncollapsed 2-input AND with a fanout of 3 averages 2.125 events per vector, while the collapsed gate averages 3.00 events per vector. For a 3-input AND with a fanout of 2, the uncollapsed gate averages about 1.94 events per vector, while the collapsed gate averages 3.00 events per vector. A 3-input AND with a fanout of 3, gives even more striking results with an average of 2.16 events per vector for the uncollapsed gate and 4.50 events per vector for the collapsed connection. This analysis shows that it is advantageous to eliminate a connections only if it does not fan out to more than one gate.

8.13 Performance Evaluation.

Four prototype simulators were constructed to test the performance of the Inversion Algorithm and the various optimizations discussed in the previous section. The first prototype is unoptimized; the second eliminates NOT and BUFFER gates; the third eliminates homogeneous connections, NOT gates, and BUFFER gates; and the fourth heterogeneous connections, homogeneous connections, NOT gates, and BUFFER gates. Heterogeneous connections were eliminated using the layered method. The linear method of eliminating heterogeneous connections was not tested. All prototypes are levelized event-driven zero-delay simulators based on the LECSIM model.

The ISCAS-85 benchmarks[**Error! Reference source not found.**] were used to certify the correctness of the prototypes. Each circuit was simulated with 5000 randomly generated input vectors, and the outputs were compared to those of the FHDL simulator[**Error! Reference source not found.**], a levelized compiled code simulator that has been in use for several years. These same circuits and input vectors were used to evaluate the relative performance of the four prototypes and the FHDL simulator. Each simulation was run on a SUN-4 IPC running SunOS with 12 megabytes of memory and a dedicated disk drive. This system was isolated from outside influences as much as possible during the execution of the tests. To isolate the effects of each algorithm, each simulation was done three different ways. First, a complete simulation was done with full input and output. Next the output functions of the simulators were disabled leaving all input and simulation functions intact, and a second set of simulations was run. Finally,

both the simulation functions and the output functions were disabled, leaving only the input functions intact, and a third set of simulations was run. Each of these simulations was performed five times and the results were averaged to minimize errors in the UNIX `/bin/time` command, which was used to report the timings. The "user" field from the output of the `/bin/time` command was used to determine the execution time. The results of the read-only simulations were subtracted from the results of the no-print simulations to obtain the results reported in Figure 23. All simulations were run as compiled code simulations. The C language was used as the target language for all of the simulators. No optimization flags were used when compiling the simulators.

Circuit	Unopt.	NOT Elim.	Hom. Elim.	Hom/Het Elim.	LCC	Activity
c432	1.7	1.6	1.4	1.2	0.5	59.4
c499	2.0	1.9	1.9	1.9	0.6	63.2
c880	3.8	3.5	3.2	2.7	1.2	57.1
c1355	6.5	5.4	5.4	4.2	1.9	56.5
c1908	8.1	5.8	5.6	4.5	4.4	56.8
c2670	17.7	13.2	12.2	11.7	5.3	55.7
c3540	16.5	11.6	10.0	9.3	8.4	52.4
c5315	36.9	28.8	28.1	22.8	21.7	63.8
c6288	40.4	40.0	39.7	33.8	30.1	61.5
c7552	52.6	40.6	39.4	33.5	40.7	60.7

Figure 23. Experimental Results.

As Figure 23 indicates, the activity rates of the circuits tested ranged from just over 50% to over 60%. At this level of activity, Levelized Compiled Code simulation (the LCC column) typically outperforms event driven simulation by a significant margin. However, for the Inversion Algorithm with deletion of homogeneous and heterogeneous connections, the timings are essentially the same for the circuits c1908, c3540, c5315, and c6288. For circuit c7552, the Inversion Algorithm actually outperforms Levelized Compiled Code simulation.

8.14 Conclusion.

As the results of the previous section indicate, the Inversion Algorithm is competitive with Levelized Compiled Code simulation, even at very high activity rates. When the activity rate is reduced to a more reasonable level, the performance of the Inversion Algorithm will increase correspondingly while the performance of LCC simulation will remain the same. Since the activity rates reported here are significantly higher than those that are likely to be encountered in practice, the Inversion Algorithm can be expected to outperform Levelized Compiled Code simulation in most practical situations.

In addition to its performance, the Inversion Algorithm has several other desirable properties. The Inversion Algorithm can be run interpretively with only a minimal impact on performance. Therefore, it can be used for fast debugging of circuits during the initial phases of the design cycle, and for more massive testing later in the design cycle with only minor differences in performance.

Because of the small size of the run-time code, the Inversion Algorithm will be beneficial to tool developers who must support many different types of development platforms. The run-time code of the inversion algorithm can be written and debugged in a few hours, making it feasible to have several different assembly-language implementations of the same code.

The results shown here suggest that the Inversion Algorithm will be an effective tool for high-performance simulation of large circuits. In spite of this, there is a massive amount of research that must be done to realize the full potential of the techniques described in this paper. Work is currently under way to extend the Inversion Algorithm to more complex timing models, and multiple-valued logic models. There is also work in progress to identify further optimizations, and to further characterize the underlying theoretical issues. The work described here should provide the foundation for much new research in high-performance simulation of VLSI circuits.

8.15 Three-Valued Simulation

Unlike two-valued simulation, in three-valued simulation the state-transition sequence is more complicated.

8.16 Introduction

The Inversion Algorithm[**Error! Reference source not found.**] is an event-driven simulation technique that rivals, and sometimes exceeds the speed of leveled compiled code simulation[**Error! Reference source not found.-Error! Reference source not found.**]. 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 by various means, the most important of which is a counting technique[**Error! Reference source not found.-Error! Reference source not found.**] which keeps track of the number of inputs carrying dominant values. Because no gate will be simulated unless its output changes value, several significant optimizations can be made in the simulation process. Unfortunately, many of these optimizations appear to depend on the two-valued logic model.

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 two-valued algorithm uses no permanent net values, and does not need to propagate either values or changes through the circuit. Simulation consists of a wave-front of activity that proceeds from the primary inputs to the outputs. When the wave hits a monitored net, the value of the net toggles. This style of simulation allows several types of gates to be eliminated or combined with other gates without affecting the correctness of the simulation.

Although the two-valued logic model appears to provide substantial benefits, it also limits the usefulness of the Inversion Algorithm. For event-driven simulation to work correctly, nets must be initialized to consistent values. For example, the inputs and outputs of **NOT** gates must have opposite values. In three-valued simulation, all nets are initialized to the **Unknown** value. For many circuits, this forces all gates to be simulated

during the processing of the first input vector. The Inversion Algorithm sidesteps the problem by simulating a single vector at compile time using a conventional leveled simulation algorithm. This works fine for combinational and synchronous sequential circuits, but often fails for asynchronous sequential circuits. Asynchronous circuits often require several input vectors for proper initialization, and the initialization sequence may be difficult or impossible to determine automatically.

If the Inversion Algorithm is to be considered a serious alternative to conventional event-driven simulation, it must be able to handle asynchronous circuits. However, it is not immediately apparent that the algorithm can be extended to three-valued simulation (**0,1** and **Unknown**) without destroying its most advantageous features. The purpose of this paper is to show that three-valued simulation is indeed possible with the Inversion Algorithm, and that the advantageous features of the two-valued algorithm can be preserved in the three-valued algorithm. To achieve this goal, it was necessary to design an entirely new algorithm around the three-valued model. There are superficial similarities between the two-valued and three-valued algorithms because they are both based on the underlying principle that no gate should be simulated unless its output is guaranteed to change value. However, the three-valued algorithm is considerably more complex than the two-valued algorithm, and moving from the two-valued model to the three-valued model is neither straightforward nor obvious.

The design of the three-valued Inversion Algorithm had several goals, the most important of which was to preserve the valueless nature of the two-valued algorithm. As in the two-valued algorithm, simulation proceeds as waves of activity proceeding from primary inputs to primary outputs. The preservation of valueless simulation permits many of the optimizations of the two-valued algorithm to be carried over into the three-valued algorithm. Another goal of the three-valued algorithm was to permit three-valued simulation to be collapsed back into two-valued simulation once the circuit has been properly initialized. This will not always be possible, since in asynchronous circuits, the unknown value can arise due to oscillations or other errors. However, for the overwhelming majority of gates, once the inputs and outputs of the gate have achieved known values, three-valued simulation is no longer necessary.

The remainder of this paper describes how these goals have been met, and analyzes the performance of several different implementations of the three-valued algorithm. With three-to-two-valued collapsing, the performance of the three-valued algorithm is comparable to that of the two-valued algorithm, and competitive with two-valued Levelized Compiled Code simulation.

8.17 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 24.

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 24. Three-Valued Truth tables.

The tables illustrated in Figure 24 demonstrate 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 from one net to another.

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 25. Counter actions for input changes.

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 25 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 8.19.

8.18 AND 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 26.

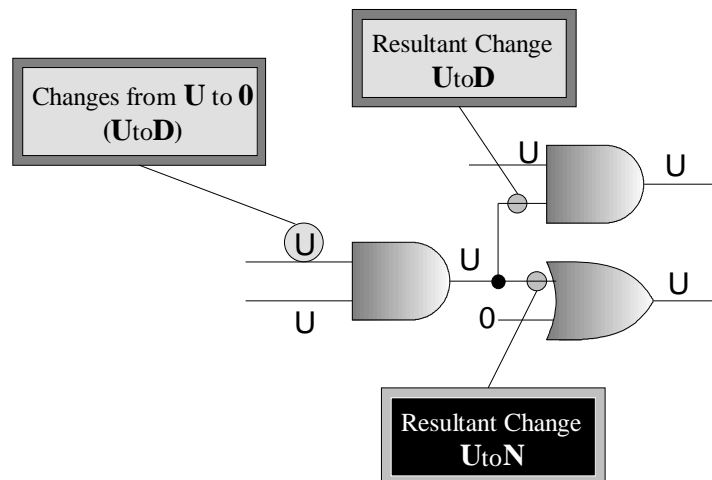


Figure 26. Conflicting Changes in Fanout Branches.

In Figure 26, 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 [**Error! Reference source not found.**] 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 heterogeneous, while those attached to OR and NOR gates are treated as homogeneous. 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 a **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 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 consider 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 **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 **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 a **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 a **UtoN** event can propagate an event other than a **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 either a **UtoD** or **UtoN** 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**. ■

The **UtoD** and **UtoN** event handlers assume that they are propagating an identical event to the output, thus **UtoD** events always queue the upper event handler, and **UtoN** 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.

8.19 XOR and NOT gates.

In the two-valued inversion algorithm, XOR gates were trivial because any change in the input of the XOR gate guaranteed a change in the output. In the three-valued inversion algorithm this is no longer true. 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 25 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 27 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.

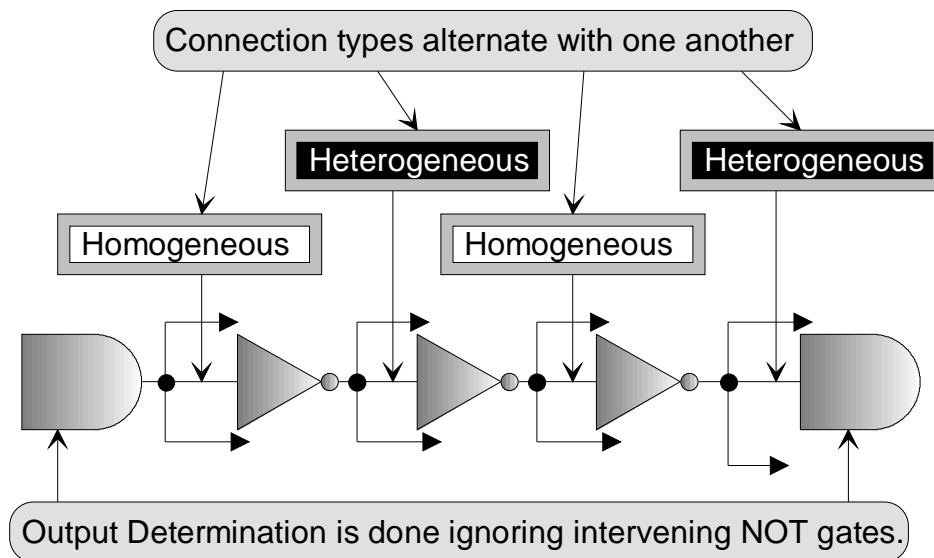


Figure 27. A Chain of Not Gates.

8.20 Collapsing Simultaneous Events.

Another aspect of the Inversion Algorithm that leads to its high performance is the collapse of events. As the input nets of a gate are processed, these nets can cause events to be added to the queue. If two of the input nets queue events that will 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 in three-valued simulation, it is necessary to understand the effect of simultaneous events on the

Design Automation: Logic Simulation

state of the gate. When an event of a certain type is to be 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 prior to this particular event, so that at most one event will be processed for each input net for each new input vector. Figure 28 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 28. Event Collapsing for AND and OR Gates.

To determine whether event collapsing is needed, the algorithm checks to see if the upper or lower event of the affected net is queued. If there is a queued event, the algorithm follows the collapse action for the new event. The only 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 simply examining the currently queued event since both the **DtoU** and **NtoU** events are lower events. 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 the opposite 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 29. 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”.

8.21 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 after 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 back 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 propagates a special event called a *Meta-Event*, through the network. After the first input vector has been applied, when many of the nets will have been assigned known values, a Meta-event can be applied to the circuit. Meta-event processing begins by analyzing all the input nets to determine the known-to-be-binary property. Processing then continues by passing a wave front, similar to simulation processing, through the inputs that are known-to-be-binary. The wave-front tests each encountered net for the known-to-be-binary property. When all inputs to a gate have the known-to-be-binary property, the gate and its outputs are assigned the known-to-be-binary property. The wave-front then continues through the gate outputs.

Meta-Event processing will be performed between input vectors. 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.

Another important point is that a circuit can have both trinary and binary event-handlers. Suppose that an input to a circuit is kept at an unknown state while a meta-event is processed through the circuit. This input will create a cone of nets through the circuit that get their input, either directly or indirectly, from the input with the unknown state. These nets will retain their trinary event-handlers because the meta-event will not propagate through gates with unknown inputs or outputs. All the other nets in the circuit will have their event-handlers transformed to binary. The binary event-handlers have been set up to schedule only the upper event-handlers of the output nets. Because the binary event-handler will only cause a **DtoN** or **NtoD** event to occur on the output of the net, and because these events are always upper events, the binary event-handler will schedule the correct trinary event-handler. If a trinary event-handler is used for any input of a gate, then trinary event handlers must be used for all inputs of the gate. This is due to the extensive event collapsing procedures that the trinary event-handlers must employ. However, binary event-handlers can schedule a trinary event-handlers for output nets.

8.22 Optimizations.

As shown in Section 8.19 elimination of NOT gates is trivial. Elimination of homogeneous connections is also simple. As shown in Figure 30 collapsing a homogeneous connection collapses the gates into a single gate with all the inputs from both gates going into the single collapsed gate. In the original configuration, an event propagating from Net **c** to Net **o** would schedule an intermediate event for Net **d**, which would schedule the event for Net **o**. In the collapsed gate, Net **c** would directly schedule an event for Net **o**.

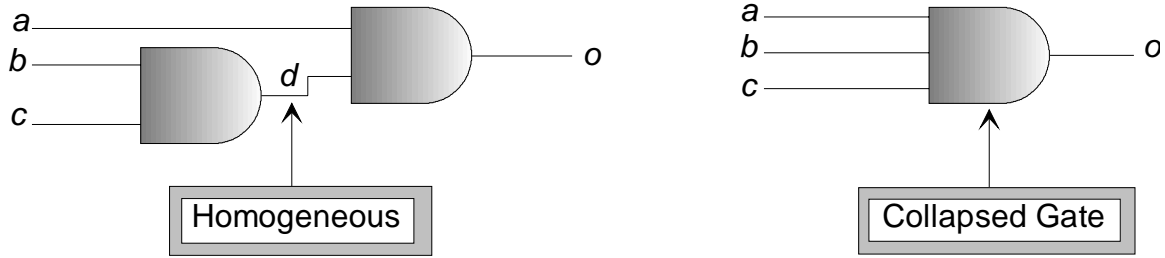


Figure 30. Homogeneous Collapsing

Collapsing heterogeneous connections is far more difficult than the other optimizations. As in the Two-Valued algorithm, the layered method of collapsing the connections was employed. In section 8.20 it was shown that all input nets to a gate will have been processed prior to the output net being processed. This causes a net to be evaluated at most once during processing of an input vector. This assumption fails for the layered method of heterogeneous connections, because the layered method must retain *some* processing for collapsed connections. As shown in Figure 31, net *a* changes first from dominant to non-dominant. This causes an event to occur on net *d*. Since net *d* has been collapsed, it will be immediately processed and will schedule an **NtoD** event on the output net *o* (assuming the output connection is homogeneous). Now net *b* is processed and causes a **DtoN** event on net *d*. Again, net *d* will be immediately processed and will attempt to schedule a **DtoN** event on the output net *o*. In essence net *d* has had two events processed on it during a single input vector. This example shows that the event collapsing of Figure 28 does not necessarily hold for heterogeneously collapsed gates. In this example, an **NtoD** event was scheduled before a **DtoN** which cannot occur according to Figure 28. Figure 32 shows all the event collapsing possibilities for heterogeneously collapsed circuits.

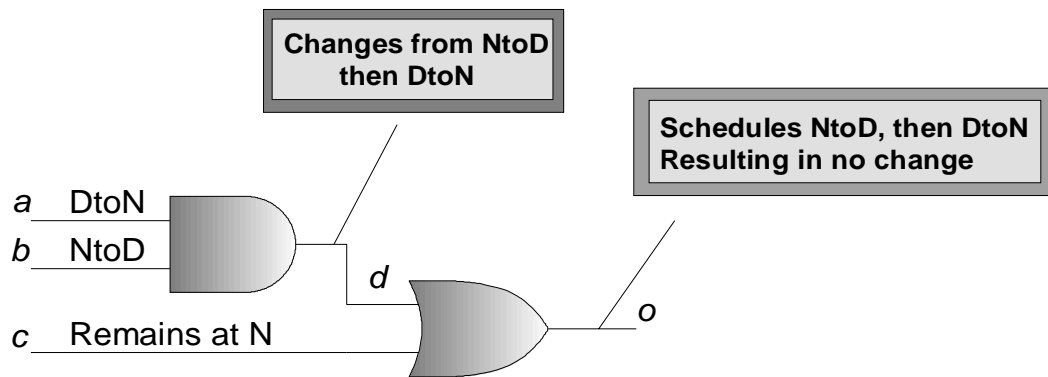


Figure 31. Heterogeneous Collapsing

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	Dominant	UtoD	U to N	Replace Upper event handler in queued event with Lower event handler
	Dominant	NtoD	N to N	Delete Queued Event
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	Dominant	UtoD	U to U	Delete Queued Event
	Dominant	NtoD	N to U	Replace upper event handler in queued event with Lower event handler
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.
	U	NtoU	N to N	Delete Queued Event

Figure 32. Collapsing of Events for Heterogeneous Collapsed Circuits

8.23 Experimental Data.

We have implemented five versions of the three-valued inversion algorithm: unoptimized three-valued simulation, elimination of not gates, elimination of homogeneous connections, collapsing heterogeneous connections, and three-to-two valued conversion after the first input vector. Each version of the algorithm contains all optimizations present in the earlier versions. 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 33.

The same data is presented graphically in Figure 34. 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 33 and Figure 34. In addition to comparing the Inversion Algorithm to two-valued LCC simulation, a special three-valued oblivious LCC simulator was constructed to

Design Automation: Logic Simulation

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	Elim Homog	Coll Hetero	2-Val LCC	3-Val LCC	Average
c432	1.7	3.2	2.4	2.2	1.9	1.7	0.5	0.8	59.4
c499	2.0	4.9	2.9	2.5	2.4	2.6	0.6	1.0	63.2
c880	3.8	7.2	5.7	5.1	4.5	3.8	1.2	2.3	57.1
c1355	6.5	10.6	8.3	7.4	7.3	5.7	1.9	3.4	56.5
c1908	8.1	16.2	11.9	7.6	7.2	5.8	4.4	6.1	56.8
c2670	17.7	33.2	25.9	20.0	18.8	16.0	5.3	16.2	55.7
c3540	16.5	30.3	22.4	17.2	15.5	12.4	8.4	24.8	52.4
c5315	36.9	61.8	47.6	39.5	38.3	33.2	21.7	38.9	63.8
c6288	40.4	67.0	54.6	52.0	51.9	44.8	30.1	48.5	61.5
c7552	52.6	88.0	67.1	53.4	50.8	43.6	40.7	58.6	60.7

Figure 33. Raw Experimental Data.

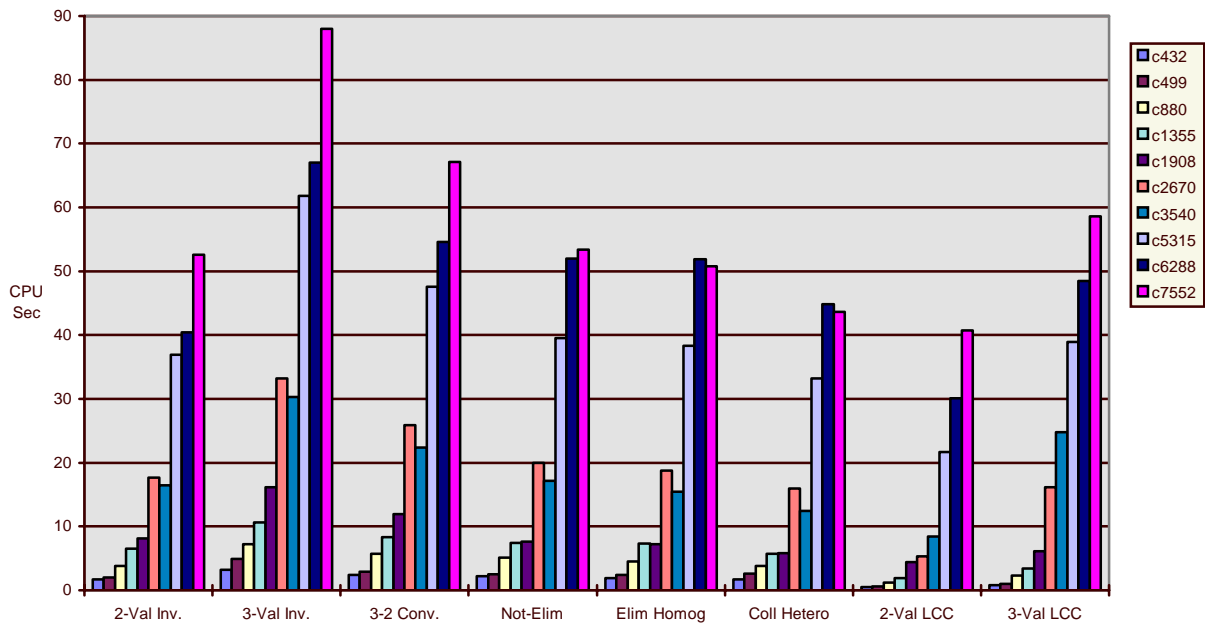


Figure 34. Graph of Experimental Data.

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 to the three-valued LCC algorithm. The three-valued algorithm, with meta-events, has outperformed three-valued LCC simulation for one circuit, c3540. The three-valued algorithm, after all eliminations and collapsing of heterogeneous connections outperforms the three-valued LCC in all circuits above, and

including, c1908. Due to the large amount of overhead code that needs to be loaded at the start of the inversion algorithm, LCC simulation will fair much better than the Inversion algorithm for very small circuits. 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 35 illustrates how the performance of the three-valued Inversion Algorithm improves as activity rate decreases to more realistic values. The data of Figure 35 was obtained by running circuit C7552 on several vector sets with differing rates of input activity. The data of Figure 35 is presented graphically in Figure 36.

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
Coll Hetero	7.8	13.1	17.6	21.0
3-Val LCC	58.5	58.5	58.5	58.5

Figure 35. C7552 with various input activity rates.

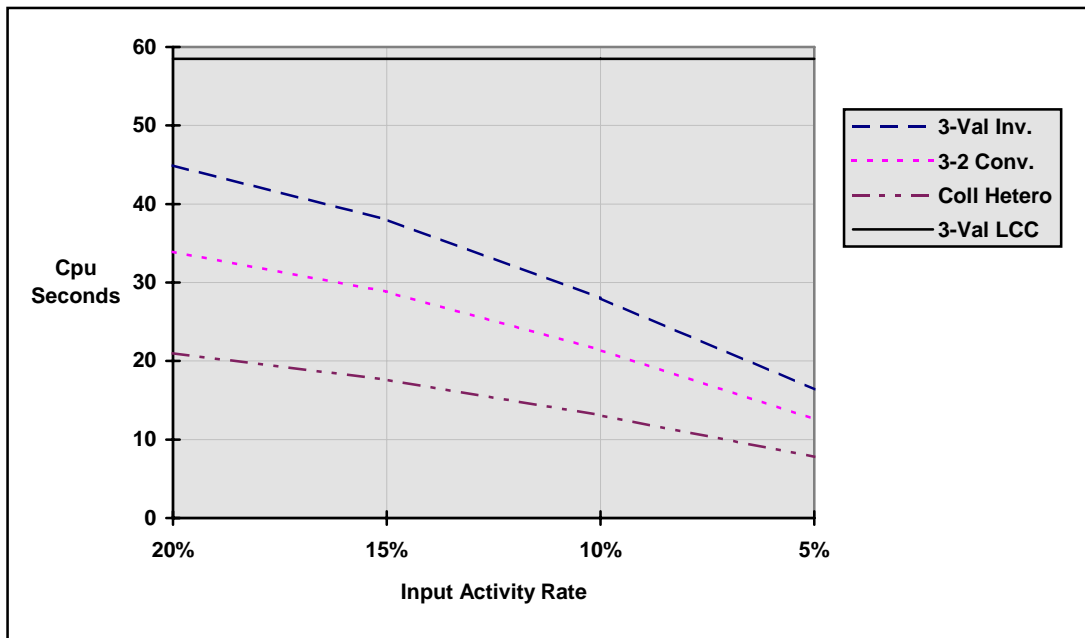


Figure 36. Graph of decreasing input activity.

8.24 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 as combinational and synchronous circuits. (The implementations described in Section 8.23 do not include compile-time simulations.) Work is 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.

8.25 Timing Models

8.26 Introduction

The Inversion Algorithm[35] is an event-driven logic simulation algorithm that provides significant advantages over existing simulation techniques[1-60]. Although it is event-driven, its performance is comparable to that of Levelized Compiled Code (LCC) simulation[21], even when the activity rate is unrealistically high. At lower activity rates, the performance of the Inversion Algorithm improves, while the performance of LCC simulation remains constant. Furthermore, the amount of run-time code required by the Inversion Algorithm is only a tiny fraction of that required by other simulation algorithms, particularly LCC simulation. The amount of code is small enough to permit assembly language routines to be used at run time without sacrificing code portability. A separate run-time module would be required for each new platform, but such a module would require no more than a few days to create.

Despite its advantages, the current implementations of the algorithm have the drawback that all of the existing implementations of the Inversion Algorithm are based on the zero-delay timing model. This does not permit one to detect static and dynamic hazards, nor does it permit one to do detailed timing analysis.

Although the two and three-valued zero-delay model can be enormously effective in diagnosing and fixing design problems, the Inversion Algorithm must be extended to include non-zero delays if it is to achieve its full effectiveness as a design tool. The problem of adding the unknown value to the simulation has been solved in a recent paper[36]. The purpose of the current paper is to extend Inversion Algorithm to the unit-delay timing model. Incorporation of the unit-delay timing model is an important first step in extending the Inversion Algorithm to more complex timing models. The unit-delay model allows one to detect hazards, and thus is more accurate than the zero-delay model, without incurring the severe performance penalty usually associated with more detailed models. As it turns out, certain optimizations of the Inversion Algorithm[35] will

require the simulator to handle delays greater than one, thus many of the problems that appear in more complex timing models must be handled by the unit delay simulator.

8.27 Fundamental Problems.

The primary obstacle in adapting the Inversion Algorithm to Unit-Delay scheduling, is that the Inversion Algorithm is essentially a *single-list* algorithm. Most event-driven logic simulation algorithms can be categorized as *single-list* or *double-list* algorithms.

In double-list scheduling two queues are used, one for events and one for gates. Any change in the value of a net generates an event, which is represented by an event structure. Event structures can be placed on the event queue during input-vector processing or during gate simulation. During input vector processing, events are generated by comparing the new input values to those of the input previous vector. Any change in an input net generates an event. No events are processed until the input vector has been completely examined. Once all events have been generated, the event handler is called to process the event queue. When an event is processed for a net N, all gates in the fanout list of N are added to the gate queue. Once the event queue has been exhausted, the gate-simulation routine is invoked to process the gate queue. After a gate is simulated, its output nets are examined for changes, and an event is added to the event queue for each changed net. Once the gate queue is exhausted, the simulator invokes the event processor to process any events that might have been queued during gate simulation. Simulation terminates when both queues become empty simultaneously. No new events are added to the event queue during event processing, and no gates are added to the gate queue during gate simulation.

In contrast, single-list simulation uses only an event queue. During the processing of an event for net N, all gates in the fanout of N are simulated, and any new events are immediately inserted into the event queue. When an event for a net N is placed in the queue, it is possible for there to be an unevaluated event for net N already in the queue. Single-list scheduling may evaluate a particular gate several times at one instant of simulated time. In double list scheduling it is possible to guarantee that no more than one event is queued for a particular net at any given time and it is possible to prevent unnecessary simulations. (Despite these drawbacks, single-list scheduling is considered by many to be faster than double-list scheduling.)

Because the Inversion Algorithm performs no gate evaluations (except for monitored nets), there is no gate evaluation step. Thus the Inversion Algorithm *must be* implemented as a single-list algorithm. This causes no difficulties in zero-delay simulation, because no event can be scheduled more than once. However, when it is possible to schedule two or more events for the same net simultaneously, the static storage management techniques used by the Inversion Algorithm may fail, leading to errors in simulation. In the zero delay version of the algorithm, a data structure is created for each fanout branch of each net. Events are queued by linking the data structures directly to other data structures already in the queue. It is possible to switch to a dynamic storage management technique, but this could impair the performance of the algorithm. The alternative is to retain the static storage mechanisms, but provide extra data structures to handle the simultaneous queuing of events.

8.28 The Red/Green Method.

The main problem with static storage management is not simultaneous queuing of events, but simultaneous queuing of event *structures*. Therefore it should be possible to avoid duplicate scheduling of event structures by creating more than one structure per fanout branch. Any mechanism to permit simultaneous queuing of several events for a single net, must also support *event collapsing*. (In the Inversion Algorithm, events are queued on a net by net basis, but are processed one fanout branch at a time. See reference [35] for details.) Any time an event is to be queued for a time slot t , the algorithm must examine the queue to determine whether there is an event queued for time slot t . If so, the new and old events must be combined into a single event. In two-valued simulation this results in both events being dropped.

The number of distinct structures that are required for each net is equal to the maximum number of events that could be simultaneously queued for any net. As Lemma 1 indicates, in unit delay scheduling, no more than two event structures will be required.

Lemma 1. In Unit Delay Scheduling with event collapsing and events processed in ascending order by time, there can be no more than two events queued for any net at any time.

Proof. Due to event collapsing, there can be no more than one event queued for any net in time-slot t . Any attempt to queue a second event for the net at time-slot t will result in the two events being combined. Since the delay of every gate is at most 1, when event is processed during time-slot t , the latest time-slot for which events can be queued is time-slot $t+1$. Since events are processed in ascending order by time, there can be no events queued for any time-slot $s < t$. Thus, when an event E is processed during time-slot t , there can be events queued for time slot t and for time slot $t+1$, but there can be no events queued for any other time slot.

Since no net can have more than two queued events at any time, we have adopted an odd/even strategy for event queuing. In this strategy, each fanout branch is assigned two event structures known as the Red and Green event structures. As in the zero delay algorithm, each of these structures contains the operating data required to process an event. However, the structures are queued in alternating fashion. When a green structure is processed, any propagated events will be queued using the red data structures, and vice versa. Input vector processing schedules only the red event structures. This strict alternation in structures, effectively, assigns colors to time-slots, as illustrated in Figure 37.

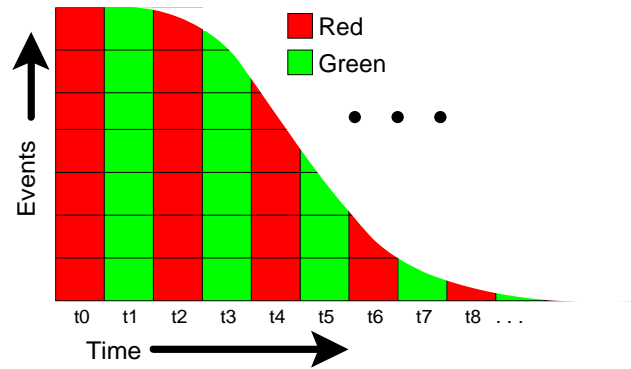


Figure 37. The Odd/Even Strategy.

8.29 Eliminating Useless Event Structures.

The main advantage of the Red/Green scheme is that it can be implemented using static pointers which require little run-time management. The main disadvantage is that it doubles the amount of storage required for static data structures. Fortunately, it is possible to eliminate many of these data structures, because not all nets can undergo both a red and a green event. For example, consider the circuit pictured in Figure 38.

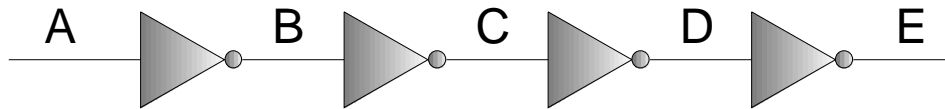


Figure 38. A Sample Circuit.

A simulation of the circuit of Figure 38 will result in at most one event being processed for each of the nets A-F. A red event will be processed for net A, a green event for net B, and so forth. Because no green event will ever be queued for net A, and no red event will ever be queued for net B, it is possible to eliminate the event structures for those events. Since the eliminated structures are never referenced by the simulation code, it may be possible to eliminate the unused structures using “dead code” techniques, however, there is a more effective method for finding and eliminating unusable structures.

The first step in eliminating useless structures is to compute the PC-Set for each gate and each net of the circuit[41]. First, each net is assigned the set $\{0\}$. Next, the circuit is processed in leveled order proceeding from the primary inputs to the primary outputs. A gate is processed when each of its inputs have been assigned PC-Sets. When a gate is processed, the algorithm computes the union of the PC-Sets of the gate-inputs and increments each PC-Set element by 1. This set becomes the PC-Set of the gate. The PC-Set of a net is computed when all gates that drive the net have been assigned PC-Sets. The PC-Set of a net is simply the union of the PC-Sets of the driving gates. Since most nets have a single driving gate, the PC-Set of a net is usually identical to the PC-Set of the driving gate.

The PC-Set of a net can be used to determine which event structures are required. If the PC-Set contains an even number, then a red event structure is required for each fanout branch, and if it contains an odd number, a green event structure is required.

8.30 Elimination of Additional Structures.

The reason for using two event structures per fanout branch is the possibility of queuing two events for the same net simultaneously. Since this occurs only when events are queued in two consecutive time-slots, it is possible to eliminate one set of event structures for nets that have no consecutive PC-Set elements. However one must use caution when performing this operation, as Figure 39 illustrates. In the network of Figure 3, the net Y has a PC-Set of {1,4}. Because events can occur at both an even and odd time, it is possible for events on Y to occur during both red and green time-slots. However, since these events can never be queued simultaneously, it may appear possible to eliminate one of the event structures for the net. The difficulty is that every event structure must specify the *color* of the events that will be scheduled when an event propagates. If the following net or nets require a strict alternation in red and green events, it is possible for the single event structure to schedule the wrong event structure at certain times, introducing errors into the simulation.

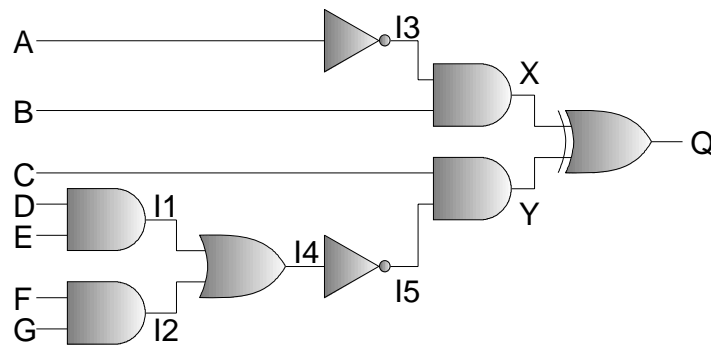


Figure 39. Non-Consecutive PC-Set Elements.

Suppose that the circuit of Figure 39 has been simulated with the input vector (A=0, B=0, C=0, D=0, E=0, F=0, G=0), and then the vector (A=1, B=1, C=1, D=0, E=0, F=0, G=0) is applied. This should cause the output Q to change from 0 to 1. The simulation is illustrated in Figure 40. The reader is encouraged to refer to both Figure 39 and Figure 40 while reading the following discussion, which may be difficult to follow otherwise.

The PC-Set of net X is {1,2}, while that of net Y is {1,4}. Suppose the green structure has been eliminated for net Y and that the red structure will be used for all events. This implies that any event on net Y will propagate a green event to net Q. Since Net X has consecutive PC-Set elements, both the red and the green structure are retained for this net. The change on Net B will cause Net X to change from 0 to 1 at time 1. The green event structure will be queued for this net. Similarly the change in net C will cause Net Y to change from 0 to 1, which will cause an event to be queued for Y at time 1. Since only the red event structure has been retained, the Red structure will be queued for net Y. An event will also be queued for Net I3 at time 1. The event on I3 will cause Net

X to change from 1 to 0 at time 2 (a static hazard). This will cause the Red event structure to be queued for Net X at time 2. Processing the Green event structure for Net X will cause the Red event structure for net Q to be queued at time 2, and processing the Red event structure for net X will cause the green event structure for Net Q to be queued at time 2. Thus, three events will be queued at time 2, a red event for net X and both a red and green event for net Q. Suppose that the event for Net X is processed first. This event will cause an event to be queued for net Q at time 3. However since the green event for net Q is already queued, it will be dequeued from the time 2 slot rather than being queued at time 3. This will leave the red event queued for Net Q at time 2. Although the final result will be the same, the change in Net Q will appear to take place at time 2 rather than time 3, which is an error.

Despite the difficulties, it is possible to eliminate some data structures for nets that have no consecutive PC-Set elements.. For example, consider a circuit identical to the cone of net Y in Figure 39. This circuit would require only a single event structure per net, even though the PC-Set for net Y is {1,4}.

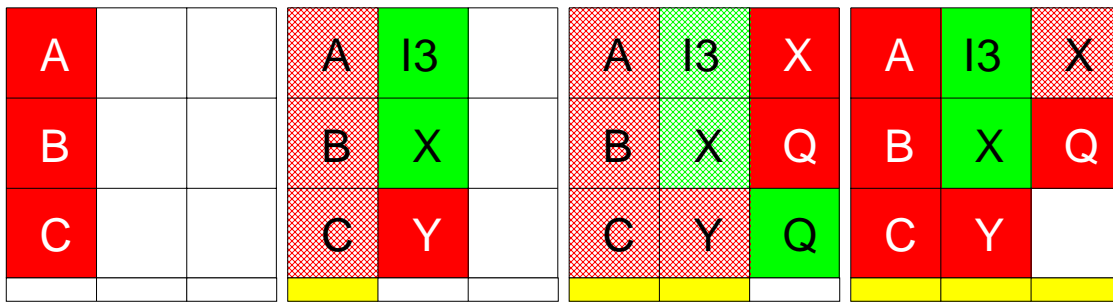


Figure 40. Timing Error due to Event Merging.

Color analysis is used to determine the number of event structures required for the fanout branches of each net. As a part of this process, the elements of all PC-Sets are set to one of three values: Red, Green, or Colorless. If a PC-Set contains consecutive elements, the even-numbered element of each consecutive pair is set to Red, while the odd-numbered element is set to Green. All other PC-Set elements are set to Colorless. The objective of color analysis is to assign a color state, Bicolored, Monochromed or Colorless, to every net in the circuit, such that, the minimum number of event structures are created to correctly simulate the circuit. BiColored nets require two event structures per fanout branch, while Monochrome and Colorless nets require only one. It is necessary to distinguish between Monochrome and Colorless nets because the process for generating data structures is somewhat different.

In addition to the three final net-states, there are two intermediate states, M+ and B+, which are assigned to Monochrome and BiColored nets with some colorless PC-Set elements. Before a final status can be assigned to M+ and B+ nets, it is necessary to assign colors to the colorless PC-Set elements. It is also necessary to avoid scheduling conflicts such as those illustrated in Figure 40. This is done by identifying and properly coloring PC-Set elements that could cause scheduling conflicts if left colorless. Five techniques are used to color PC-Set elements, Consecutive Element Coloring as described above, Demand Coloring, Sympathetic Coloring, Minimal Coloring, and Parity Coloring.

Design Automation: Logic Simulation

Consecutive element coloring is performed once at the beginning of color analysis. The other coloring procedures are executed in the order given above, until no more elements can be colored. The procedure advances to a new coloring method only when no more nets can be colored using the previous method. For example Sympathetic Coloring is done only if no net can be colored by Demand Coloring.

Demand Coloring is applied to all gates G with a BiColored output N . For each Red(Green) element k of N , all elements $k-1$ are selected from the PC-Sets of the inputs of G . These elements are colored Green(Red). Demand coloring is scheduled whenever an element of a BiColored net is assigned a color, and is performed repeatedly until no more nets can be colored. Demand coloring may change a Colorless net to a Monochrome net or a Monochrome net to a BiColored Net.

Sympathetic Coloring is applied to all gates with an $M+$ output, N , of color Red(Green). All colorless elements of the PC-Set of N are colored Red(Green). Sympathetic coloring must be propagated to the primary outputs of the circuit. If the element k of net N is colored Red(Green) by Sympathetic Coloring, it is necessary to examine the outputs of any gate which uses N as an input, and select all PC-Set elements of value $k+1$. These elements must be colored Green(Red). During propagation of Sympathetic coloring, if a BiColored gate is encountered and contains an element of value $k+1$ that is colored Green(Red), all the inputs, other than the input that originated the propagation, of the BiColored gate must be examined. Any input net with elements of value k must be colored Red(Green) and be propagated to the primary outputs. The necessity of this back propagation can be seen in the circuit of Figure 41. The BiColored gates, $G4$ and $G5$, will cause the coloring of the gates $G1$ and $G2$. Gate $G1$ will be colored green and gate $G2$ will be colored red due to demand coloring. Because the level 2 element in both $G4$ and $G5$ is not a consecutive element, it will not cause the coloring of any of the level 1 inputs. If gate $G1$ is encountered first in sympathetic coloring, the level 1 element will be colored green to be consistent with the level 7 element. This will propagate to gate $G4$, and color its level 2 element red. If the coloring is not propagated back to gate $G2$, then gate $G2$'s level 1 elements will remain colorless. When gate $G3$ is encountered during sympathetic coloring, the level 1 element will be colored red to be consistent with the level 8 element. This coloring will be propagated to gate $G5$ coloring the level 2 element green. Again, suppose the coloring is not propagated back to gate $G2$. Upon completing sympathetic coloring, demand coloring will be called to color any BiColored nets that were affected or created. During demand coloring upon reaching gate $G4$, gate $G2$ will be colored green. Continuing onto gate $G5$, gate $G2$ will be recolored to red. A conflict has arisen requiring the level 1 elements of gate $G2$ to assume both the red and green color, which is not possible. Also, because demand coloring is recalled if a gate is colored, the recoloring of gate $G2$ will cause the compiler to loop infinitely on demand coloring.

As can be seen in Figure 41, all of the level 1 and level 2 elements need to be colored consistently so that during simulation, correct event collapsing will occur. If the level 1 element of gate $G1$ were to be colored green and the level 1 element of gate $G2$ were colored red, events caused by gates $G1$ and $G2$ would not be collapsed thereby giving incorrect results. By propagating back one gate upon encountering a BiColored gate, a consistent coloring of all elements that affect the BiColored gate can be achieved. Back propagating in the circuit of Figure 41 will cause gate $G2$ to be colored green, given

that gate G1 is colored before gate G2, and the level 1 element of gate G3 will be colored green as well. This also has the affect of changing gate G3 to a BiColored gate, which is necessary to ensure correct event collapsing in gate G5. It is not necessary to back propagate when encountering a MonoColored gate because the data structure for a MonoColored net has both the red and green labels. It is only necessary to ensure the correct color is used for queueing a BiColored net.

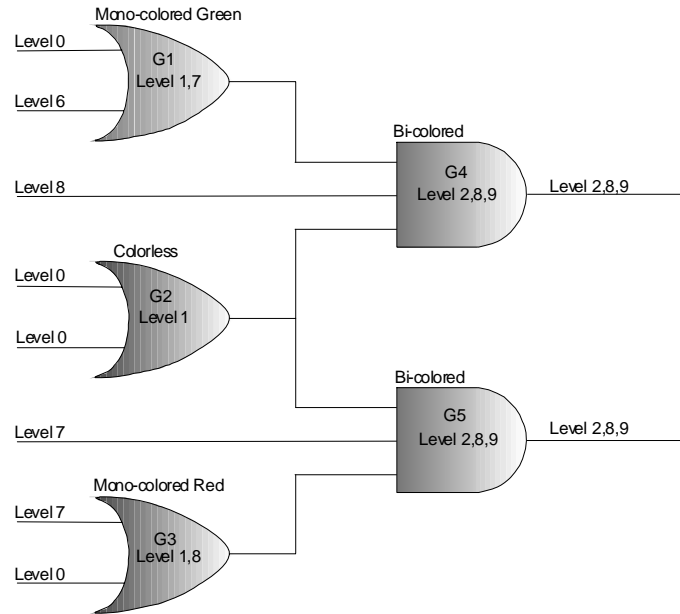


Figure 41. Sympathetic Coloring

Minimal coloring is applied to colorless nets N whose PC-Sets have more than one element. N must be the input of a gate with a BiColored output. The lowest PC-Set value of N is colored consistently with its parity, and the remainder of the PC-Set elements are colored using Sympathetic Coloring. Minimal coloring must propagate to the primary outputs.

Parity coloring is used only when no other type of coloring applies. Parity coloring is applied to the PC-Sets of B+ nets. The colorless elements are colored consistently with their parity, and the color is propagated to the primary outputs. Parity coloring may create nets that can be colored by one of the other methods.

When none of the above coloring techniques can be applied, the coloring process stops, and data structures are generated for the fanout branches of each net in the circuit. Both red and green data structures are generated for BiColored nets. A single Red(Green) data structure is generated for the fanout branches of monochrome nets. The Red(Green) data structure will schedule a Green(Red) structure for any propagated events. To simplify the scheduling of events for monochrome nets, a second dummy data structure of opposite color is overlaid on top of the generated structure. This allows the data structure to be scheduled either as a Red or a Green data structure.

A single data structure is also generated for each the fanout branch of a colorless net. As for monochrome nets, a dummy data structure is overlaid on top of the first, allowing the data structure to be scheduled either as a red or a green data structure. For colorless

nets, propagated events may schedule either the red or green data structure, whichever is convenient. The following lemma shows why this is possible.

Lemma 2. Let G be a gate with input net N and output net M . Suppose that N is a colorless net. Then M must be either colorless or monochrome.

Proof: The only other possibility is that M is BiColored. If this were the case then demand coloring would color at least one PC-Set element of N . Once this were done, N would no longer be colorless.

As Lemma 2 shows, whenever an event is propagated from a colorless net, the net to which it is propagated has a single data structure. Since each single data structure is overlaid with a dummy structure of the opposite color, the structure can be scheduled as either a red or a green data structure with identical results.

It is possible to eliminate even more data structures by performing color analysis on individual fanout branches instead of whole nets. Since data structures are generated for each fanout branch instead of each net, it is possible for a BiColored net to have only monochrome fanout branches. Under the current scheme, these nets would have two data structures generated for each fanout branch when, strictly speaking, only one is necessary.

8.31 Elimination of Gates and Connections

In the zero-delay Inversion Algorithm it is possible to eliminate NOT, BUFFER, XOR and XNOR gates. It is also possible to collapse heterogeneous and homogeneous connections. (See reference [35] for an explanation of this terminology.) However in the Unit-Delay model these optimizations cause a fundamental change in the timing model. Although it is possible to eliminate all NOT gates from the simulation without changing the final values computed by the simulator, it is necessary to retain the delay of all eliminated gates to avoid invalidating the hazard analysis. For example, a chain of NOT gates may have been added to the circuit to balance path-lengths and eliminate hazards. If the NOT gates are simply deleted from the simulation without retaining the delay, the NOT chain will appear to have no effect on the dynamic behavior of the circuit.

Eliminating gates and connections effectively transforms the unit-delay simulation into a multi-delay simulation under the transport-delay model. (The inertial delay model is inappropriate, because gates exhibiting delays larger than 1 are actually collections of simpler gates.) This compounds the problems that led to the adoption of the red/green model. Figure 42 illustrates the collapsing of gates and connections. Once the NOT gate and the homogeneous connection have been collapsed out of the circuit, the resulting gate has three inputs, one (C) with a delay of 1, and two (A and B) with delays of 3. These delays, which are constant and associated with gate inputs, indicate the number of time-slots that must be added to the current slot to find the appropriate queue location for propagated events. It is possible for different fanout branches of a net to have different delays.

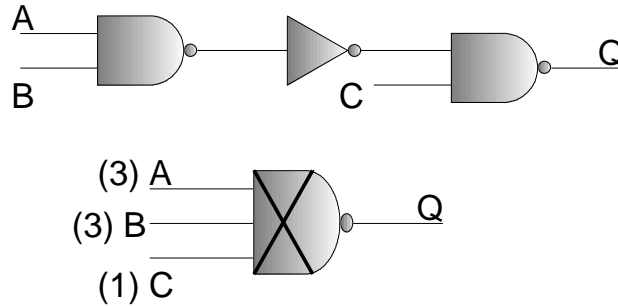


Figure 42. Collapsed Gates and Connections.

The following lemma is the multi-delay extension of Lemma 1.

Lemma 3. Let G be a gate with output net N and let k be the maximum delay on any input of the gate. The maximum number of events that can be queued simultaneously for the net is $k+1$.

Proof. Similar to that of Lemma 1.

More important than the lemma itself is the following corollary.

Corollary 3.1. Let k be the maximum delay value over all inputs of all gates. Then the maximum number of events that can be queued for any net is $k+1$.

Corollary 3.1 allows us to extend the idea of the red/green technique to multiple colors. Instead of red and green we will use the set of colors $\{0,1,2,\dots,k\}$, where k is the maximum delay described in Corollary 3.1. We then allocate $k+1$ event structures for each net which are colored with the colors 0 through k . If i is less than k , then all data structures of color i will schedule structures of color $i+1$. Structures of color k will schedule structures of color 0.

8.32 Eliminating Data Structures for Collapsed Networks.

Using k structures for each fanout-branch in the circuit is sufficient to avoid conflicts, but not necessary. As is the case for uncollapsed networks, it is possible to identify unused structures and eliminate them. It is first necessary to compute the PC-Sets of the collapsed network, using a slightly modified algorithm. Because delays are associated with the inputs of a gate rather than with the gate itself, it is necessary to add the delay to each PC-Set element before forming the union of the PC-Sets of a gate's inputs. Since delays are not necessarily one, it is the actual delay associated with the input that must be added to each element of the input's PC-Set before forming the union.

Each element e of a PC-Set is categorized using the function $(e \bmod k+1)$, where k is the maximum color value. If the PC-Set for a net N has no elements of category c , then structures of color c are not required. More structures may be eliminated by performing an operation similar to the search for adjacent PC-Set elements described above. However, because delays are not uniformly 1, the PC-Set does not provide enough information to determine the number of colors needed by a particular net. The maximum

Design Automation: Logic Simulation

number of colors needed by a net is equal to the maximum number of events that may be simultaneously queued for the net. The PC-Set gives information about when events will be processed, but no information about when events are queued. To determine the maximum number of events that can be queued at any one time, it is necessary to know both when an event enters the queue and when it leaves the queue. To make this determination, it is necessary to compute the Queue Density Function $D_N(i)$. For each net N , $D_N(i)$ gives the maximum number of events that can be queued at time i . Technically, the domain of the queue density function is the entire set of natural numbers, but the only domain elements that are of interest are $0-m$, where m is level number of the net in question.

The first step in computing $D_N(i)$ is to compute a modified PC-Set for each net in the circuit. The modified PC-Set consists of duples containing the level at which the net will be queued and the delay of the net. These duples indicate the time at which a net will be queued and duration for which the event must exist before being processed. Once all modified PC-Sets have been computed, they can be used to compute the queue density functions in the following manner. Let G be a gate with n inputs. If the largest delay on any of the n inputs is k , then create a queue Q with $k+1$ elements. The queue consists of a collection of Boolean values, which indicate whether the corresponding time slot is empty or full, along with pointers to all the duples that occupy that time slot. The queue density function is computed by Algorithm 1.

```

Initialize the set Modified-PC to empty;
For each input i do
    For each element x of the PC-Set of i do
        d := DelayOf(i);
        Add (x,d) to Modified-PC;
    EndFor
EndFor
/* Sort by PC Set Value */
Sort the elements of Modified-PC into ascending order
    by the value of the first coordinate then the second coordinate;

For k := 0 to n do
    Push empty onto tail of Q;
EndFor;
CurrentQueuePosition := 0;
/* Process elements of Modified-PC in sorted order */
For each (x,d) in Modified-PC do
    While CurrentQueuePosition < x do
        k := The number of full positions in Q;
        Set the value of  $D_N(\text{CurrentQueuePosition})$  to k;
        Pop Head of Q;
        Push empty onto tail of Q;
        CurrentQueuePosition := CurrentQueuePosition + 1;
    EndWhile
    /* First Queue Position is 0 */
    Set the dth element of Q to full and add a pointer to (x,d);
EndFor
While Q is not empty do
    k := The number of full positions in Q;
    Set the value of  $D_N(\text{CurrentQueuePosition})$  to k;
    Pop Head of Q;
    CurrentQueuePosition := CurrentQueuePosition + 1;
EndWhile

```

Algorithm 1. Queue Density Function Computation.

Algorithm 1 can also be used to compute the Queue Population Function $P_N(i)$, which will be useful in assigning colors to slots. The function $P_N(i)$ is similar to $D_N(i)$, but $P_N(i)$ returns the set of filled queue positions with the pointers, instead of the number of filled positions. The function $D_N(i)$ can be computed from $P_N(i)$, but the reverse is not true.

Once all queue density functions have been computed, it is possible to determine the maximum number of colors required by each net. Let N be a net with queue density function f . Assume further that the level of net N is m . Let c be the maximum of $f(x)$ $0 \leq x \leq m$. The events of N can be scheduled without conflict using no more than c colors. Unfortunately, c colors may not be enough to prevent scheduling conflicts in successor gates. Even if C is the maximum value over all queue density functions, it may be necessary to use more than C colors to schedule the entire circuit without conflict.

Design Automation: Logic Simulation

Two types of conflicts arise when combining gates into networks: parent-child conflicts and sibling conflicts. The circuit pictured in Figure 43 illustrates a parent-child conflict. In this figure, the numbers in curly braces are the PC-Sets of the corresponding nets, while the numbers indicated by “d=“ are the delays associated with the inputs.

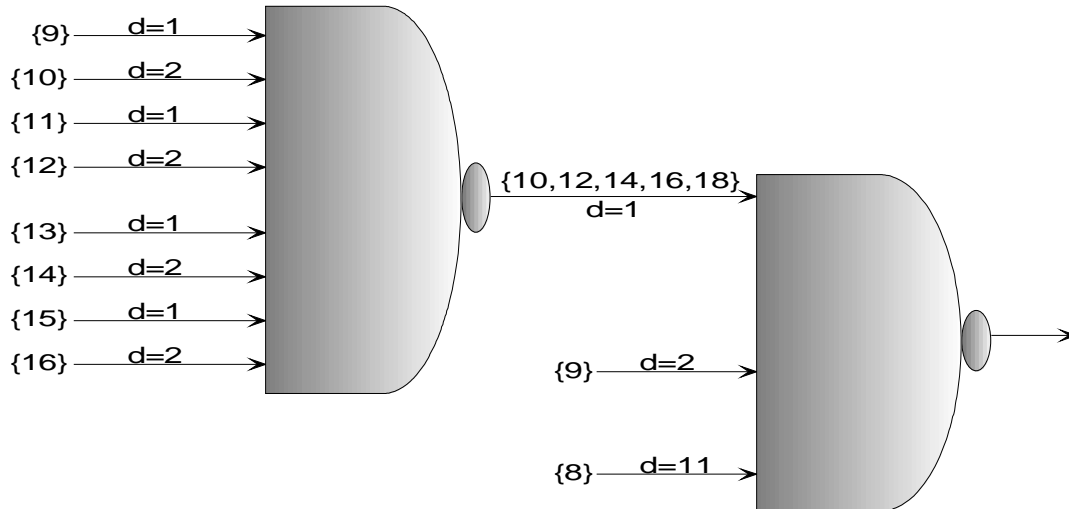


Figure 43. Scheduling Conflicts.

In Figure 43, there are only two queue density functions of interest, those for the outputs of the NAND gates. The maximum queue density for either of these nets is 2. However to avoid scheduling conflicts, it is necessary to use 3 colors to schedule the events of the first NAND gate. To illustrate how the conflict occurs, consider the computation of the queue density functions illustrated in Figure 44 and Figure 45.

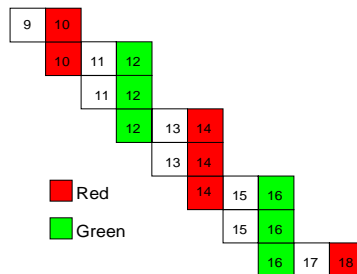


Figure 44. Computation of the First Queue Density Function.

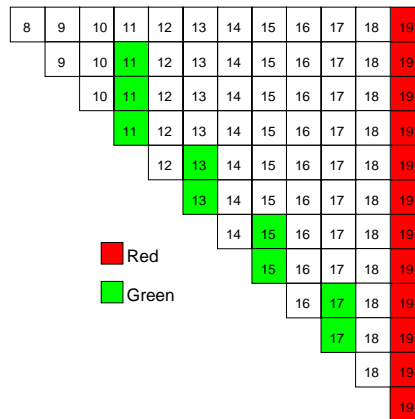


Figure 45. Computation of the Second Queue Density Function.

In Figure 45 slots 17 and 19 must be assigned different colors, since events can be queued in slot 17 and slot 19 simultaneously. Since events are propagated using static pointers, this requires two different colors to be assigned to slots 16 and 18 in Figure 44. If these two slots were assigned the same color, they could not queue events of two different colors. In Figure 45 slot 15 must have a different color from slot 19, which implies that in Figure 44, slot 14 must have a different color from slot 18. But in Figure 44 slot 14 and 16 must have different colors as well. Putting this all together, in Figure 44, slots 16 and 18 must be of different colors, slots 16 and 14 must be of different colors and slots 18 and 14 must be of different colors. This cannot be accomplished without using three colors. However, as the figures clearly show, the maximum queue density of either net is 2.

To avoid parent-child conflicts, it is necessary to propagate color information from the output of a gate to its inputs, but propagating this information may give rise to sibling conflicts. (Color propagation is the multi-color analog of Demand Coloring.) Sibling conflicts arise between two or more fanout branches of a single net. To understand how such conflicts arise, it is necessary to have a precise understanding of the scheduling mechanisms used in the Inversion Algorithm. The data structures used for scheduling are illustrated in Figure 46.

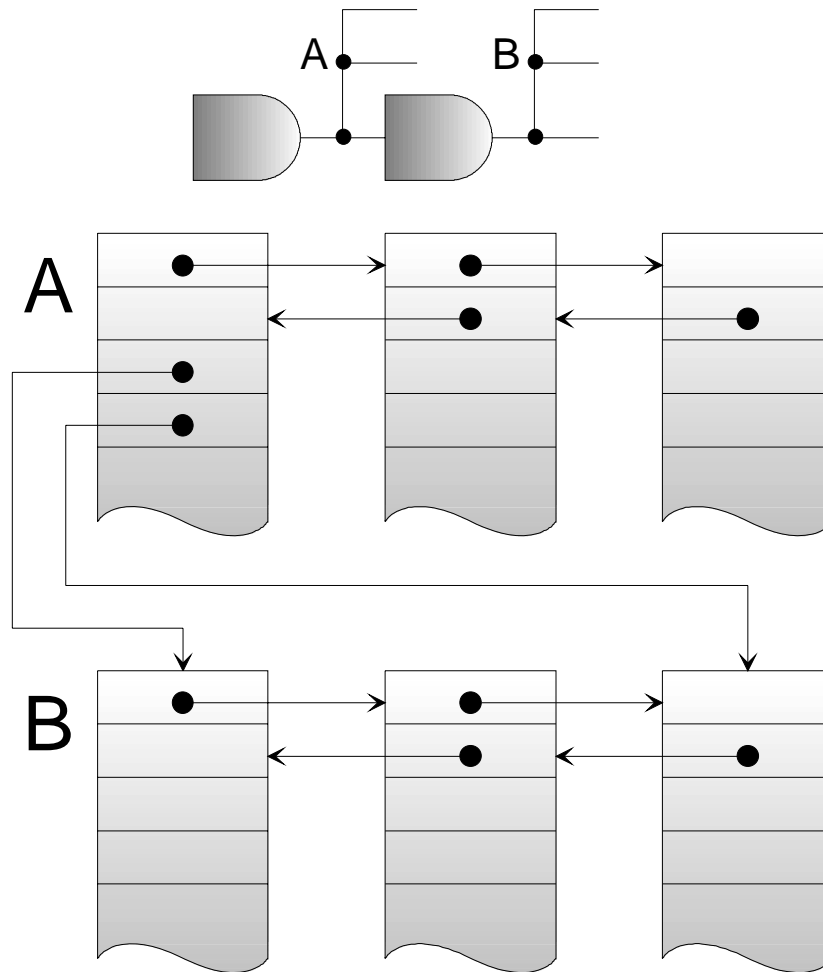


Figure 46. Scheduling Data Structures.

As Figure 46 illustrates, each fanout branch of a net is represented by a separate data structure. The set of data structures representing a net are chained together using static forward and back pointers. Each data structure contains static pointers to the chain that will be scheduled if an event propagates. Because static pointers are used, no data structure may appear in more than one chain. Figure 47 illustrates how sibling conflicts occur.

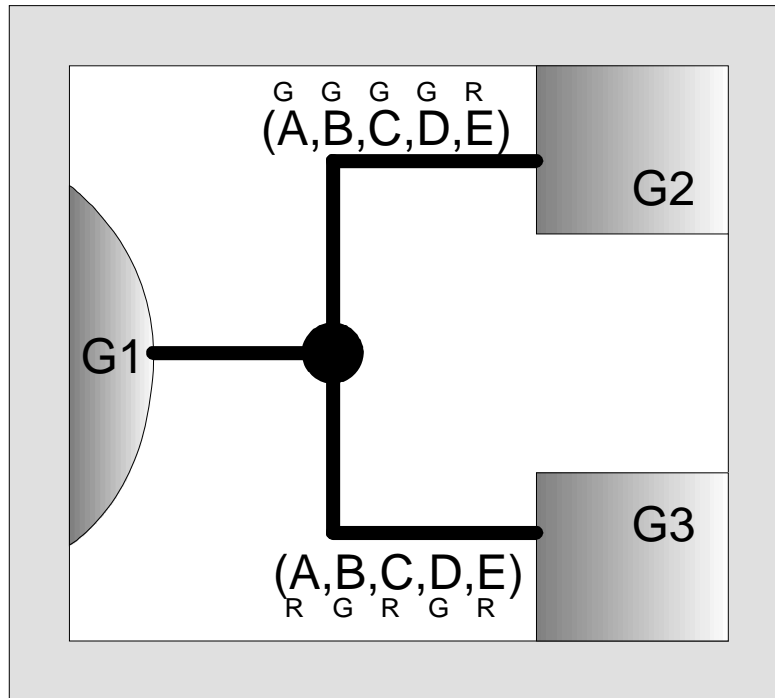


Figure 47. Sibling Conflicts.

In Figure 47, it is assumed that colors have been assigned to all time slots of the outputs of G2 and G3, and that color information has been propagated from outputs to inputs. Assuming that the net pictured in Figure 47 has the PC set {A,B,C,D,E}, it is necessary to determine the color of the structures that must be scheduled if an event propagates at any one of these times. Assuming further that the colors Red and Green are sufficient to schedule events for the outputs of G1 and G2, the indicators R and G indicate which data structures must be scheduled at each time slot. During time slot A, a Green event must be scheduled for gate G2, and a Red event must be queued for gate G3. During time slot B, a green event must be queued for both gates. Writing the combinations as a sequence of ordered pairs, the complete list is (G,R), (G,G), (G,R), (G,G), and (R,R). There are three distinct pairs, (G,G), (G,R), and (R,R). Because static pointers are used, it is necessary to create three distinct chains of data structures. In the first chain, the data structures point to the green structures for both nets, in the second, the data structures point to the red structures, while in the third one points to the red structures, and the other points to the green structures. The data structures are illustrated in Figure 48. Note that in this figure, it has been necessary to use three different colors to construct the data structures for the net. The information provided by the queue population function may require additional data structures to be created.

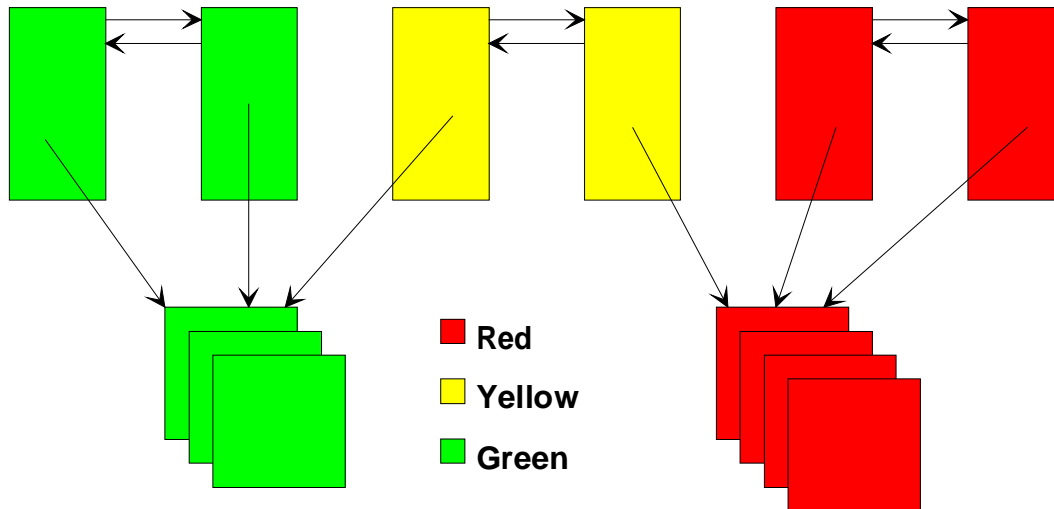


Figure 48. Sibling Conflict Resolution.

We have developed a coloring function which can color the time-slots of all nets in such a way as to avoid both parent-child conflicts and sibling conflicts. Before applying the coloring function to a network, it is necessary to break any cycles and levelize the resultant acyclic network. For synchronous sequential circuits, this can be done in straightforward way by breaking each synchronous flip-flop[44]. For asynchronous sequential circuits, a method such as the convergence algorithm outlined in [40] can be used. Colorizing starts with all fanout-free primary outputs of the network. Because no gates follow a primary output, the delay associated with the output will be one. Therefore, by Lemma One, fanout-free primary outputs need at most two colors. Algorithm 2 is used to assign colors to these nets. (This form of the algorithm is for illustrative purposes only. In practice, the obvious changes will be made to improve performance.)

```

For Each Fanout Free output  $x$  do
  For Each element  $p$  of the PC set of  $x$  do
    If this is the first element of the PC set
      Color time-slot  $p$  with 0;
    Else
      If there exists element  $(p - 1)$ 
        Color time-slot  $p$  with Complement (time-slot  $(p - 1)$ );
      Else
        Color time-slot  $p$  with 0;
      EndIf
    EndIf
  EndFor
EndFor;
  
```

Algorithm 2. Assign Colors to Primary Outputs.

Once all primary outputs have been colored, the main coloring algorithm, Algorithm 3, is used to color the remainder of the network. This algorithm is based on two queues, a

queue of gates, GQ , whose input nets require coloring, and a queue of nets, NQ , whose fanout branches must be combined to resolve sibling conflicts. In Algorithm 3, the modified PC-Sets of the fanout branches of a net are considered to be distinct from the modified PC-Set of the net itself. The **Propagate** function propagates colors to fanout branches, the **Combine** function propagates colors from the fanout branches to the net itself, and the **Recolor** function refines the coloring generated by the **Combine** function.

```

For Each gate  $G$  whose output is a fanout-free Primary Output do
    Add  $G$  to  $GQ$ ;
EndFor
While  $GQ$  is not empty do
    For Each gate  $G$  with output  $N$  in  $GQ$  do
        Propagate the coloring of  $N$  to each input  $I$  of  $G$ ;
        Delete  $G$  from  $GQ$ ;
        If all fanout branches of the input  $I$  have been colored Then
            Add  $I$  to  $NQ$ ;
        EndIf
    EndFor;
    For Each Net  $N$  in  $NQ$  do
        Combine the color information of the fanout branches of  $N$ ;
        Recolor slots based on the Queue Population Function of  $N$ ;
        Delete  $N$  from  $NQ$ ;
        If  $N$  is the output of a gate  $G$  Then
            Add  $G$  to  $GQ$ ;
        EndIf
    EndFor
EndWhile

```

Algorithm 3. The Main Coloring Algorithm.

Algorithm 4 is the **Propagate** function used in the main coloring algorithm. This algorithm propagates colors to the fanout branches of a net, not to the net itself. No conflicts can arise during this process.

```

Propagate(N:OutputNet,G:Gate)
begin
  For Each input I of G do
    d := DelayOf(I);
    For Each time-slot t in the Modified PC-Set of N do
      c := ColorOf(t);
      If the PC Set of I contains the element t-d Then
        Color the element t-d with color c,
          in the PC Set of the fanout branch of I
          leading to G;
      EndIf
    EndFor
  EndFor
End Propagate;

```

Algorithm 4. The Propagate function.

Algorithm 5 is the **Combine** function used to prevent sibling conflicts. This algorithm also illustrates how forward scheduling information is obtained for each data structure.

```

Combine(N:Net)
Var S:Set of Tuples, k:Integer;
begin
  S := The Empty Set;
  k := The number of Fanout branches of N;
  For Each element t in the modified PC Set of N do
    If the k-tuple of propagated colors for time slot t
      is not already contained in S Then
      Add the k-tuple of propagated colors to S;
    EndIf;
  EndFor;
  Assign the colors 0 through  $|S|-1$  to the elements of S,
    and to the corresponding time-slots;
  /* Retain information for the Data-Structure Generator */
  Retain the set S, and links between the elements of S and
    time slots;
End Combine;

```

Algorithm 5. The Combine Function.

The **Combine** function not only performs an initial color assignment, it also creates a vital piece of scheduling data, the *k*-tuple of colors associated with each time-slot. Effectively, this algorithm assigns a two-dimensional color to each time slot. The first component is the color created by the **Assign** statement, while the second component is the *k*-tuple of propagated colors. Both the color and the *k*-tuple will be used to create and link the final data structures.

Finally, Algorithm 6 is the **Recolor** function used to assign the final colors to the time-slots of a net. Although it does not appear explicitly in the algorithm, the association between time-slots and k -tuples is maintained throughout the recoloring process.

```

Recolor( $N$ :Net)
Var  $K$ : $k$ -tuple,  $c$ :color;
begin
  Repeat
    For Each element  $t$  in the modified PC-Set of  $N$  do
      For Each element  $s$  in  $P_N(t)$  do
        If  $s$  has the same color as a previous element of  $P_N(t)$  Then
           $K :=$  the  $k$ -tuple associated with  $t$ ;
          If there is a time-slot  $u$  of color  $c$ ,
            and elements of the Modified PC-Set pointed to by the time slot
               $u$  do not appear in  $P_N(t)$ ,
            and  $K$  is equal to and in the same order as the  $k$ -tuple
associated
              with time-slot  $u$ ,
            and  $c$  is greater than the current color of  $s$  Then
              Assign  $c$  to  $s$ ;
          Else
             $c :=$  The smallest color not used to color any element of
              the Modified PC-Set of  $N$ ;
            Assign  $c$  to  $s$ ;
          EndIf
        EndIf
      EndFor
    EndFor
  Until No Slots are Recolored;
End Recolor;

```

Algorithm 6. The Recolor Function.

Note that when a time-slot is recolored by the **Recolor** function, the new color will always be numerically larger than the existing color. This eliminates any circularity problem that may occur should it be necessary to recolor a time-slot more than once. Since it is possible to recolor events more than once, it is possible that multiple recolorings of several nets will leave the net in a state where scheduling conflicts are still possible. Hence it is necessary to repeat the recoloring process until no more nets can be recolored.

Once the main coloring algorithm has completed, it is possible to create scheduling data-structures for each of the nets. First, a set of data structures is created for each net. Suppose a net N has fanout k and that c colors have been used to color the time-slots of the net. In this case c chains of data structures will be created, each one of which has k elements. The forward scheduling information for each data structure will be taken from the k -tuple associated with the color of the data structure. Although, strictly speaking, k -tuples are associated with slots rather than colors, the process of assigning colors to slots

Design Automation: Logic Simulation

guarantees that if two slots have the same color, then they are associated with the same k -tuple.

The simulation code is quite simple, and essentially identical to that used by the zero delay algorithm. For the sake of completeness, this code is replicated in Figure 49. In this code, the scheduling data structures are referred to as *Shadows*[38]. This code is written in C, the Inversion Algorithm implementation language. The computed goto at the end is actually implemented in assembly language.

```
INCREMENTX:
    /* Alternate the INC & DEC processors */
    *Current_Shadow->subroutine = &DECREMENTX;
    (*Current_Shadow->Lock)++;
    /* If a change in the output will occur */
    if ((*Current_Shadow->Lock) == 1)
    {
        /* If the gate is not already queued */
        if (Current_Shadow->last_fanout->next ==
            Current_Shadow->last_fanout)
        {
            /* queue the gate for simulation */
            if (Queue_Tail != NULL)
            {
                Queue_Tail->next =
                    Current_Shadow->first_fanout;
                Current_Shadow->first_fanout->previous =
                    Queue_Tail;
            }
            else
            {
                Queue_Head =
                    Current_Shadow->first_fanout;
                Current_Shadow->first_fanout->previous =
                    NULL;
            }
            Queue_Tail = Current_Shadow->last_fanout;
            Current_Shadow->last_fanout->next = NULL;
        }
        else
        {
            /* dequeue the gate */
            Current_Shadow->last_fanout->next->previous =
                Current_Shadow->first_fanout->previous;
            Current_Shadow->first_fanout->previous->next =
                Current_Shadow->last_fanout->next;
            Current_Shadow->last_fanout->next = NULL;
            Current_Shadow->first_fanout->previous =
                Current_Shadow->first_fanout;
        }
    }
    Temp = Current_Shadow->next;
    Current_Shadow->next = Current_Shadow;
    Current_Shadow = Temp;
    if (Current_Shadow == NULL) return;
    Goto **Current_Shadow->subroutine;
```

Figure 49. Inversion Algorithm Code.

The scheduling code differs significantly from the zero-delay algorithm, in that, instead of using several queues, each of which represents a PC-Set level in the circuit, this algorithm uses a single queue. Inside the queue are structures called Sentinels which serve as timing markers. When a set of Shadows is queued, each shadow is moved forward in the queue past a number of Sentinels equal to the delay of the shadow. When a Sentinel structure is dequeued for processing, it causes the algorithm to output the current state of the primary outputs. Each Sentinel acts as a single gate-delay marker. As noted in reference [35] the run-time code for the Inversion Algorithm consists of several (less than ten) slightly different versions of the routine pictured in Figure 49.

8.33 Experimental Data.

We have implemented the unit-delay algorithm. We have compared this algorithm to .. using the ISCAS 85 combinational benchmarks[15]. 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 50.

The same data is presented graphically in Figure 51. 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 50 and Figure 51. Also included in the experimental results is the percentage of data structures that were not created due to the color analysis algorithm.

The Unit-Delay Inversion algorithm shows a significant speed-up even in unrealistic activity rates. Although circuit c6288 did not show improvement over all the other algorithms, it should be noted that this circuit has a very high number of static hazards causing an extreme number of events to be queued. As such, c6288 is not included in the graph of the data. The reduction in the number of data structures created is very significant. On average, a third of the data structures were eliminated from the run time code, making a significant reduction in the amount of memory required for processing.

Circuit	Interp.	Gateway	C-Shad	Unit-Delay Inversion	% Savings in Data Struc
c432	23.4	3.6	3.9	3.1	31.5
c499	26.1	4.2	4.5	3.0	74.5
c880	46.3	14.0	8.3	7.6	49.0
c1355	93.8	30.9	18.0	13.1	26.3
c1908	172.9	61.1	32.9	26.4	35.1
c2670	192.1	81.1	43.8	39.1	38.3
c3540	277.1	112.7	63.9	48.6	29.6
c5315	519.1	228.3	126.9	97.6	38.1
c6288	5108.6	2602.5	1245.6	1348.8	22.4
c7552	795.1	372.7	201.1	155.0	25.1

Figure 50. Raw Experimental Data

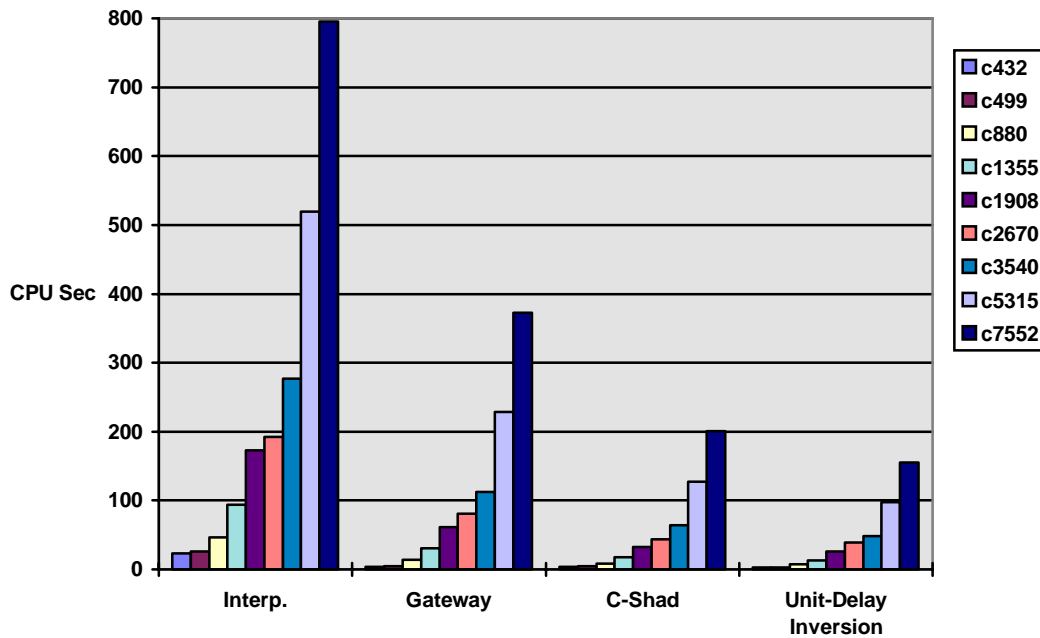


Figure 51. Graph of Experimental Data

8.34 Conclusion.

Although the Inversion Algorithm has been proven to be effective for zero-delay simulation, there has been some question about its application to more complex timing models. This paper shows that the algorithm can be adapted to the Unit-Delay model and out performs other unit-delay algorithms. The two most significant problems in this adaptation, are providing the ability to eliminate gates and connections the same way this is done in the zero delay model, and reducing the number of duplicate data structures required to prevent scheduling errors. This paper has also shown, in a preliminary way, how the more complex multi-delay model could be used with the Inversion Algorithm. However, in the Unit-Delay model, even after deletion of NOT gates and collapsing connections, the number of non-unit delays should be relatively small. In the multi-delay model, non-unit delays tend to be the norm, which may impose more stringent demands on the elimination of duplicate data structures. The multi-delay model also supports several types of delay including both transport and inertial delay. The current paper discusses only the transport delay model. Support for the inertial delay model, with event canceling, will be significantly different.

This paper serves to demonstrate the versatility and adaptability of the Inversion Algorithm. It must be noted that the run-time code required by the Unit-Delay model is virtually identical to the run-time code for the Zero-Delay model. Due to the extremely small size of the code, the Unit-Delay model should be just as adaptable as the Zero-Delay model. Finally, this paper demonstrates that the Inversion Algorithm is a widely applicable technique that will prove to be an effective design automation tool in the future.

8.35 Interpreted Simulation

8.36 Introduction.

Simulation is an integral part of virtually every integrated circuit design methodology. Few of today's complex integrated circuits could have been designed without some form of simulation, and for many integrated circuits, simulation consumes more time than any other design activity. Because of this, there have been many recent improvements in the speed of simulation algorithms[**Error! Reference source not found.-Error! Reference source not found.**], and there have been several implementations of simulation algorithms in hardware[**Error! Reference source not found.-Error! Reference source not found.**]. Logic simulation is popular because it is reasonably efficient, and provides a reasonably accurate model of the circuit being designed. The components of the logic model can be compiled directly into silicon. For full custom designs, netlist comparison tools can be used to verify the layout against the logic model. Because simulation is a critical part of VLSI design, and because integrated circuits are growing ever larger and more complex, there has been a constant search for new and more efficient simulation algorithms.

Oddly enough, the search for more efficient algorithms has lead back to an algorithm that was widely used before the advent of event driven simulation, namely Levelized Compiled Code (LCC) simulation[**Error! Reference source not found.**]. The LCC algorithm enjoys the reputation of being the fastest available method for performing logic simulation. This reputation is somewhat undeserved, but under certain conditions it is accurate. LCC simulation belongs to a class of simulation algorithms known as "Oblivious," because the number of gates simulated is not dependent on circuit activity. Such algorithms tend to minimize the amount of time required to simulate a single gate while maximizing the number of gates simulated per input vector. As expected, oblivious algorithms out-perform event-driven techniques when circuit activity is high, but are less advantageous when circuit activity is low. LCC simulation is attractive because the break-even activity rate between LCC simulation and traditional event-driven simulation is quite low, typically from 1 to 3 percent. This low break-even point is crucial to the success of the LCC algorithm. If the break-even point were 40 or 50 percent, the LCC algorithm would be considerably less attractive.

One disadvantage of LCC simulation is the time required to build the simulator. Because code must be generated and compiled, a significant amount of time must elapse between making a change to a circuit and running the first simulation. This can slow the debugging process and increase development time. Nevertheless, many VLSI designers are willing to put up with long compile times to obtain increased simulation performance.

Recently, a new algorithm, the Inversion Algorithm, has challenged LCC simulation for its position as "the fastest known type of simulation." The Inversion Algorithm is event driven, and has been shown to be capable of exceeding the speed of LCC simulation in some cases, even when activity rates exceed 50%[**Error! Reference source not found.**]. When activity rates decrease, the speed of the Inversion Algorithm increases, while the speed of LCC simulation remains constant. In addition to its speed, the Inversion Algorithm has a number of other intriguing advantages. It can be

implemented either as an interpreted or as a compiled algorithm using virtually the same run-time code, and it can be implemented in less than ten pages of code. This suggests that an interpreted implementation of the Inversion Algorithm ought to run at about the same speed as a compiled version. The purpose of this paper is to describe the differences in the compiled and interpreted implementations of the Inversion Algorithm, and to present experimental data to compare the two approaches.

8.37 An Overview of the Inversion Algorithm.

Although the Inversion Algorithm is described in detail in reference **[Error! Reference source not found.]**, a short description is necessary here to make this paper self-contained. The Inversion Algorithm is an event-driven simulation algorithm which differs from other event-driven algorithms in one important respect. In ordinary event-driven simulation, no gate is scheduled for simulation unless one of its *inputs* changes value. In the Inversion Algorithm no gate is scheduled for simulation unless its *output* is guaranteed to change value. When an event occurs on a gate input, the event processor performs tests to determine whether the event will propagate through the gate. If the gate simulation will not cause event propagation, no simulation is performed. These tests are relatively simple compared to the amount of processing required to simulate a gate. Since no gate is simulated unless its output changes value, a gate can be simulated by inverting its output value. The inversion can be done using a single unary operation.

For simplicity, the current implementation of the Inversion Algorithm is limited to eight gate types: AND, OR, NAND, NOR, XOR, XNOR, BUFFER, and NOT. (This *is not* a limitation of the algorithm, only of the current implementation.) Different event processors must be used for the inputs of different gates. Since any change in an input of an XOR, XNOR, BUFFER, or NOT gate implies a change in the output, no special tests are required for these gates. All events propagate unconditionally through the gate. Because an XOR or an XNOR gate has multiple inputs, it is possible to schedule one of these gates several times during the processing of a single input vector. However, successive changes in an output imply successive inversions of the output value. Since two successive inversions cancel one another, the inversion algorithm combines successive events and eliminates them.

Processing for AND, OR, NAND, and NOR gates is more complex, because not all events propagate through the gate. To handle gates of this type, the Inversion Algorithm maintains a count of dominant values for the gate. For OR gates, this is a count of the number of inputs with the value one, while for AND gates this is the count of the number of inputs with value zero. When the count changes from one to zero or from zero to one, a change is generated in the output. (As noted in references **[Error! Reference source not found.]** and **[Error! Reference source not found.]**, a similar counting technique can be used in conventional event-driven simulation.)

Every change in an input causes the count of dominant values to be incremented or decremented. Two event handlers are used for each net, one that increments the count and one that decrements it. These routines are forced to alternate with one another in strict fashion. As long as the identity of the next event processor is known, there is no need to maintain the value of the input net. Furthermore, since every change in the inputs of XOR, XNOR, BUFFER, and NOT gates generates a change in the output, it is not

necessary to maintain values for the input nets of these gates. This implies that, except for primary inputs and monitored nets, no net values are ever required.

The key element which enables the Inversion Algorithm to be implemented in either compiled or interpreted form is the underlying implementation using the shadow algorithm[**Error! Reference source not found.**]. The general structure used to represent an event is pictured in Figure 52.

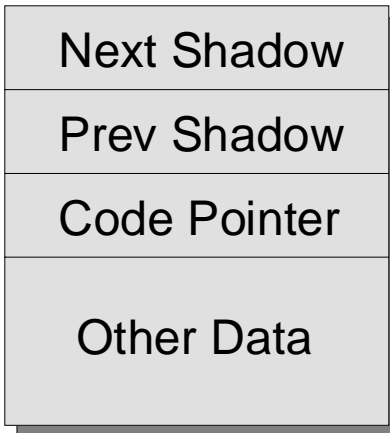


Figure 52. The Structure of a Shadow.

In Figure 52, the Previous and Next-Shadow pointers are used to queue and dequeue the event, while the Code Pointer contains a pointer to the event-processing routine. This structure differs from that of the “pure” Shadow Algorithm in that it contains both forward and backward pointers, and the value of the Code Pointer changes during simulation. In the Inversion Algorithm, one event structure is generated per fanout branch, and the number of different event-processing routines is fixed. In the “pure” Shadow Algorithm, one structure is generated per net and the number of different event and gate processing routines varies from circuit to circuit. The Inversion Algorithm has no gate processing routines.

8.38 Changing to an Interpreted Implementation.

The changes required to adapt the Inversion Algorithm from a compiled implementation to an interpreted implementation are surprisingly simple. To illustrate, the steps performed by the compiled implementation are listed in Figure 53.

1. **Create the global data structures and the queues.**
2. **Write a data-structure definition to the output file for each fanout branch in the circuit.**
3. **Write a copy of the event-handling routines to the output file.**
4. **Generate the input-vector processing routine, and write it to the output file.**
5. **Generate the output-vector print routine and write it to the output.**
6. **Generate the main simulation loop and write it to the output file.**
7. **Compile the output file.**
8. **Run the compiled program.**

Figure 53. Outline of the Compiled Implementation.

Design Automation: Logic Simulation

The primary change between the compiled and interpreted implementation is dynamically allocating data structures, and writing data to memory rather than to an output file. The circuit compiler must contain the event processors, and must be supplied with generic input and output routines as well as a generic main loop. The generic input and output routines are somewhat slower than the customized versions created for the compiled implementation, because in the customized routines all loops are completely unrolled. Otherwise the compiled and interpreted routines are identical. Because the Inversion Algorithm uses a fixed set of event handlers, no customization of the code is required in the compiled implementation, and no generalization of the code is required for the interpreted implementation. Figure 54 illustrates the steps performed by the interpreted algorithm.

1. **Allocate the global data structures and the queues.**
2. **Allocate space for the event-descriptors.**
3. **Write a data-structure definition into memory for each fanout branch in the circuit.**
4. **Enter the main simulation routine.**

Figure 54. The Interpreted Implementation.

The next section describes several performance comparisons that were made between the two implementations.

8.39 Experimental Results.

Because the interpreted and compiled simulations result in almost precisely the same sequence of instructions being executed during simulation, intuition would suggest that the simulation times for the two implementation techniques should be nearly identical. This intuition is very nearly correct, however the interpreted implementation runs in a larger partition, and places the event-descriptors in different portions of the virtual address space. As the results of Figure 55 show, these factors have some effect on the simulation time. The times reported here are in seconds of CPU execution time, and exclude both the time required to read and print vectors, and the time required to compile the circuit. The times were obtained using 5000 randomly generated test vectors. The tests were run on a SUN IPC with a dedicated disk drive and twelve megabytes of memory.

Circuit	LCC Sim.	Unoptimized			NOT Elimination		
		Comp.	Interp.	%Incr.	Comp.	Interp.	%Incr.
c432	0.5	1.7	2.7	240.00	1.6	2.3	- 99.33
c499	0.6	2.0	2.6	233.33	1.9	2.5	- 99.19
c880	1.2	3.8	4.7	216.67	3.5	4.3	- 98.38
c1355	1.9	6.5	6.5	242.11	5.4	5.7	- 97.77
c1908	4.4	8.1	8.5	84.09	5.8	5.6	- 93.10
c2670	5.3	17.7	19.7	233.96	13.2	14.8	- 94.36
c3540	8.4	16.5	18.2	96.43	11.6	14.0	- 87.97
c5315	21.7	36.9	39.5	70.05	28.8	30.2	- 58.89
c6288	30.1	40.4	43.5	34.22	40.0	43.8	16.89
c7552	40.7	52.6	56.4	29.24	40.6	42.5	38.85

Figure 55. Interpreted Implementation Performance.

The results of Figure 55 show that the performance penalty for using interpreted simulation ranges from a high of about 60% for the smallest circuit, to a low of less than 5%. During a debugging session where relatively few test vectors are used this difference in speed would be barely noticeable. The results of Figure 55 show two different versions of the Inversion Algorithm, one which performs no optimizations, and one which eliminates NOT gates before performing simulation. (As explained in reference [Error! Reference source not found.], it is possible to eliminate all NOT and BUFFER gates from the Inversion Algorithm simulation without altering the results of the simulation.) The difference in compilation time between the two implementations is given in Figure 56. The circuits listed are the ISCAS-85 combinational benchmarks that have been used to evaluate the performance of many new simulation techniques, as well as algorithms for automatically generating test vectors[Error! Reference source not found.]. The times reported here are the amount of real time that must elapse before the first vector can be simulated. Real times are reported here instead of CPU times, because this represents the amount of time that the user must wait before the first meaningful result can be obtained. These times were obtained using a dedicated system, and depending on other activity in the system, may vary by a few seconds from run to run. The system configuration for these tests was the same as that for the data reported in Figure 55.

Compile Time			
Circuit	Gates	Compiled	Interpreted
c432	160	12.1	0.6
c499	202	14.2	0.7
c880	383	21.9	1.3
c1355	546	30.8	2.1
c1908	880	37.0	2.6
c2670	1269	64.7	4.2
c3540	1669	73.5	5.7
c5315	2307	128.1	8.7
c6288	2416	132.9	9.9
c7552	3513	193.8	14.4

Figure 56. Compilation Time Differences.

As Figure 56 illustrates, the difference in compilation time for the two techniques is more than a factor of 10. However, for the circuits studied, the absolute amount of time required for the compiled implementation is small enough to be tolerable: slightly more than 3 minutes for the largest circuit. The real benefit of using the interpreted implementation comes when debugging larger circuits. To determine the benefit of the interpreted implementation for a large circuit, let us assume that the relationship between compile time and circuit size is linear, and that circuit c7552 is representative of this relationship. (As Figure 57 and Figure 58 show, these assumptions are not unreasonable.)

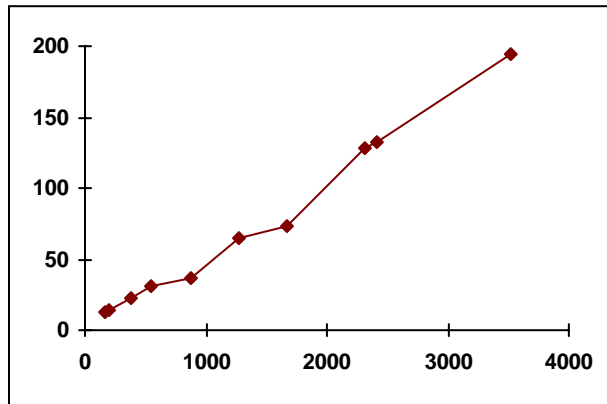


Figure 57. Compiled Compilation Time vs. Circuit Size.

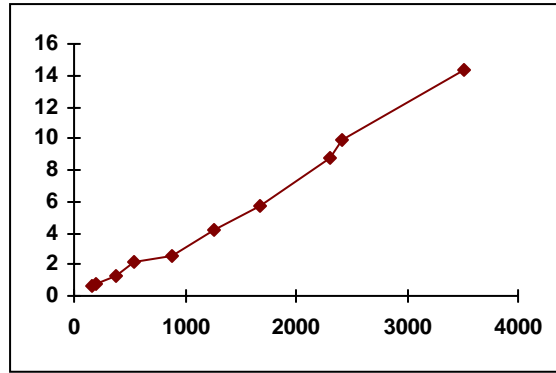


Figure 58. Interpreted Compilation Time vs. Circuit Size.

For circuit *c7552*, the compiled simulator compiles an average of 18.13 gates per second, while the interpreted simulator compiles an average of 243.96 gates per second. Extrapolating these numbers to a quarter-million gate circuit, the respective compilation times are 3 hours and 48 minutes for the compiled implementation and 17 minutes for the interpreted implementation.

8.40 Conclusion

This paper has discussed the differences between compiled and interpreted versions of the Inversion Algorithm. The Interpreted implementation has been shown to be almost as fast, in terms of simulation time, than the compiled technique, especially for larger circuits. At the same time it has also been shown that the interpreted compilation time is significantly smaller than that of the compiled implementation.

The numbers reported here suggest directions for future research. Although the interpreted implementation provides for much faster compilation times, 17 minutes for a complete recompilation is nearly intolerable for intense debugging sessions. While it is true that a circuit of this size would normally be partitioned into smaller pieces for debugging, there might be advantages to doing at least a portion of the debugging on the entire circuit. For such a technique to be feasible for extremely large circuits, some form of incremental compilation will be necessary, regardless of what type of implementation is used. Because the executable portions of the interpreted Inversion Algorithm are fixed, it is an ideal vehicle for performing such compilations. Precompiled portions of the circuit will consist only of data structures which can be loaded or replaced as needed. The internal links used by these data structures, and the multi-level queue structure of the simulation will complicate matters, but these problems have well-known straightforward solutions.

Regardless of future research, the Inversion Algorithm has been shown to be effective technique for debugging medium to large circuits, and will undoubtedly become an indispensable tool in the logic designer's repertoire.

8.41 Performance

8.42 Summary

8.43 Exercises