Chapter 4

Multi-Delay Simulation

4.1 Differing Gate Delays

When the timing accuracy of unit-delay simulation is insufficient to debug a circuit, it is necessary to move to a more detailed timing model. There are many different models, and there is no common agreement on the terminology used to describe them. However, to clarify the discussion, we will define our terms as follows.

Multi-Delay Simulation Gates may have arbitrary delays, but the delays must be integers. Acceptable delays are 1, 5, 10, and 5000. A delay of 2.5 is not allowed. The unit of time is arbitrary. Delays are usually fixed, but may depend on gate input and output values.

Nominal Delay Simulation Gate delays are arbitrary, but not necessarily integers. A delay of 2.5 is acceptable. Time units are usually specified.

Timing Simulation Gate delays are not fixed, but dependent on conditions that exist in the circuit, including those external to the gate.

Circuit Simulation The circuit is simulated as a set of differential equations. Net values are represented as voltages rather than ones and zeros.

This chapter will focus on Multi-Delay simulation, with some discussion of Nominal Delay simulation.

Assigning differing delays to each gate complicates the process of simulation. To illustrate the problem, consider the circuit pictured in Figure 4-1.

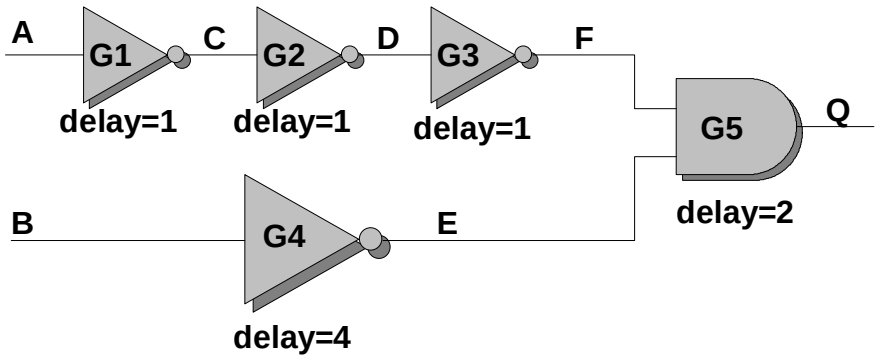

Figure 4-1. A Circuit with Differing Delays.

Suppose that there are changes in inputs A and B at time zero. Because of the delays in G1, G2, and G3, the change will not reach net F until time 3. The change in net E will arrive one time unit later, at time 4. This presents a problem for the event-driven simulation algorithm, because changes at time zero will cause both G1 and G4 to be scheduled for simulation at time 0. This will cause events to be scheduled for nets C and E at time 1. To prevent the change in E from happening too soon, it is necessary to prevent the event for E from being processed at time 1. This could be done by delaying the simulation of G4 until time 3, but if there were additional changes in Net B at times 1, 2, or 3, the simulation of G4 would be performed with incorrect input values. For correct simulation, it is necessary to method is to simulate G4 at time 0, but avoid processing the event on E until time 4. There are two simulated times associated with each event, the creation time and the execution time. These times must differ by the delay of the gate.

Although delaying an event seems simple, using the wrong algorithm can significantly degrade simulation performance. Suppose we decide to use a simple event queue, such as that used in unit-delay simulation, and add events as they are created. When it comes time to process events for time T, the simulator would need to search the queue for all events with the proper time stamp. Although it may not be apparent, this technique is functionally equivalent to the **Repeated Minimum Sort** algorithm, which is known to be an $O(n^2)$ algorithm. It is possible to modify the algorithm to keep the queue in ascending order by execution time. As each event is created, the queue is scanned for the proper insertion point. This approach is functionally equivalent to the **Insertion Sort** algorithm, which is also an $O(n^2)$ algorithm.

The primary difficulty is that events are not created in execution-time order. Before events can be processed they must be sorted into ascending order by execution time. The most successful algorithms are based on efficient sorting algorithms. Two techniques used in practice are the *Timing Wheel*, and the *Priority Queue*. The timing wheel is a variation of **Bucket Sort**, an $O(n)$ algorithm, while the priority queue is a variation of **Heap Sort**, an $O(n \lg n)$ algorithm. Even though the Timing Wheel is more efficient, it is not applicable to all simulations. The Priority Queue, on the other hand, has few restrictions on its use.

4.2 The Timing Wheel

The Timing Wheel is used when delays are expressed as integers, when the largest delay is reasonably small. The term "Reasonably Small" means different things in different situations, but for the purposes of this discussion we will assume that the maximum gate delay is less than 100. The idea behind the timing wheel is that the event queue will be replaced by several event queues, one for each unit of simulated time. The timing wheel is simply a zero-based array of queues, as illustrated in Figure 4-2.

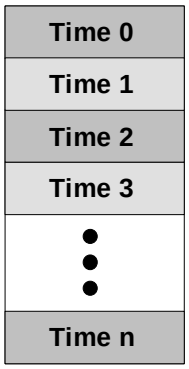

Figure 4-2. Timing Wheel Structure.

The most simplistic implementation would create one queue for each instant of simulated time, but this is not necessary. Suppose that the maximum delay of any gate is D. When gates are being simulated at time T, the latest time for which I could queue an event is time T+D. There can be no events queued at time T+D+1, or at time T+D+I, for any I > 0. Because gates are not simulated until all events are processed, there can be no events queued for time T, or for any time earlier than T. In particular, the event queue for time T is guaranteed to be empty. The empty queue can be used to store events for time T+D. Because of this, the maximum number of queues required is D, where D is the maximum delay of any gate in the circuit. The queues are reused in a circular fashion as illustrated in Figure 4-3. The term Timing Wheel comes from the circular reuse of queues.

Figure 4-3. The Reuse of Timing Wheel Slots.

As Figure 4-3 illustrates, each queue is reused as many times as necessary. For a simulation time T, the index of the associated queue is computed by the statement Q = T % D where % is the notation for the remainder operation.

It is necessary for the simulator to keep track of the current time, or at least the current queue index. The event processing routines are much the same as those for unit delay simulation with the appropriate modifications. The new main routine is illustrated in Algorithm 4-1 and the new event processor is illustrated in Algorithm 4-2, with the changes from the unit-delay algorithm double-underlined.

```
SimulateVector:
{
    GetInput( );
    CurrentTime := -1
    While Event_Queue is not empty do
        CurrentTime := CurrentTime + 1
        CurrentTime := CurrentTime % MaxDelay
        Event_Processor( );
        If Gate_Queue is not empty Then
            PrintIntermediateOutput( );
            Gate_Processor( );
        EndIf
    EndWhile;
    PrintFinalOutput( );
}
```

Algorithm 4-1. The Main Simulation Routine.

```
Process_Events:
{
    For each Event E in Event_Queue[CurrentTime] do
        N := Net Identifier of E
        Copy New Value from E to the Net Table entry for N.
        For each Gate G in the fanout of N do
            If G is not already in Gate Queue Then
                Add G to Gate Queue
            End If
            Remove E from Event Queue[CurrentTime]
        End For
    End For
}
```

Algorithm 4-2. The Multi-Delay Event Processor.

The new gate processor is illustrated in Algorithm 4-3. As in Algorithm 4-2, the changes from the unit-delay algorithm are double-underlined.

```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        D := The Delay of G
        Q := (CurrentTime+D) % MaxDelay
        Simulate G, put result in New_N
        If New_N is different from The current value of N Then
            Create a new event E
            Net Identifier of E := N
            New Value of E := New_N
            Add E to Event_Queue[Q]
        End If
        Remove G from Gate_Queue
    End For
}
```

Algorithm 4-3. The Multi-Delay Gate Processor.

The changes illustrated in Algorithm 4-2 and Algorithm 4-3 will produce an accurate Multi-Delay simulation only if there are never two events queued for the same net at the same time. (The two events would have to be queued in two different queues.) If two events are queued for the same net at the same time, the modified algorithms can produce erroneous results. In Figure 4-4, G is assumed to be contained in a much larger circuit. The input and output values are assumed to be those that exist at time 100. At time 101 an event is processed that changes the value of **a** from 1 to 0. This causes an event to be scheduled for b at time 105, which will change the value of b from 0 to 1. At time 102, another event is processed that changes the value of a from 0 back to 1. The event for time 105 is still queued, so the value of b has not yet been changed. G is simulated at time 102 with the input value 1, but because the output of G is already 0, no new event is created. Suppose no further events occur on **a**. At time 105 the event is processed for b changing its value to 1. At the end of the simulation, both **a** and b will be 1, an impossible situation.

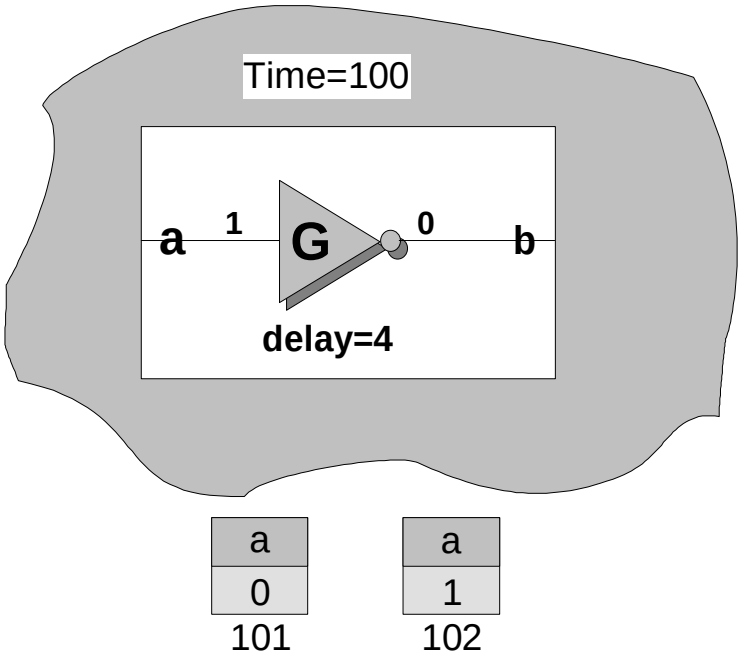

Figure 4-4. A Multi-Delay Scheduling Error.

The error occurs when the result of the gate simulation at time 102 is compared to the existing value of the net b. This is not the value b will have at time 106, so the comparison produces an erroneous result. This problem can be handled in several ways. One technique is to keep track of the events that are queued for net b, and compare new simulation values against the value contained in the event. Another technique is to delay the comparison of new and old values until the event is processed. The second technique is simpler to implement, and is illustrated in Algorithm 4-4 and Algorithm 4-5. The main routine is illustrated in Algorithm 4-1. The corrections are double underlined. It's important to note that this technique will produce many more events than testing for changes during gate processing.

```
Process_Events:
{
    For each Event E in Event_Queue[CurrentQueue] do
        N := Net Identifier of E
        If New Value from E <>  Net Table value for N Then
            Copy New Value from E to the Net Table entry for N.
            For each Gate G in the fanout of N do
                If G is not already in Gate Queue Then
                    Add G to Gate Queue
                End If
            End For
        End If
        Remove E from Event Queue[CurrentQueue]
    End For
}
```

Algorithm 4-4. The Corrected Multi-Delay Event Processor.

```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        D := The Delay of G
        Q := (CurrentTime+D) % MaxDelay
        Simulate G, put result in New_N
        /* Create Event Unconditionally */
        Create a new event E
        Net Identifier of E := N
        New Value of E := New_N
        Add E to Event_Queue[Q]
        Remove G from Gate_Queue
    End For
}
```

Algorithm 4-5. The Corrected Multi-Delay Gate Processor.

4.3 Transport Delay Vs. Inertial Delay

There is another problem with this approach. Figure 4-5 illustrates the wave-forms for the input and output of the NOT gate of Figure 4-4 during the simulation described above. Unfortunately, Figure 4-5 is not an accurate picture of how the circuit will really behave. The real behavior of the circuit is illustrated in Figure 4-6. In reality, the delay of the gate represents the time required for the output to change from zero to one. The change is neither discrete nor instantaneous. The output of the gate will change continuously from a low voltage to a high voltage. If the process is interrupted, the output will never achieve

the high voltage. It is unlikely that the intermediate voltage would be recognized by a one by any subsequent gates.

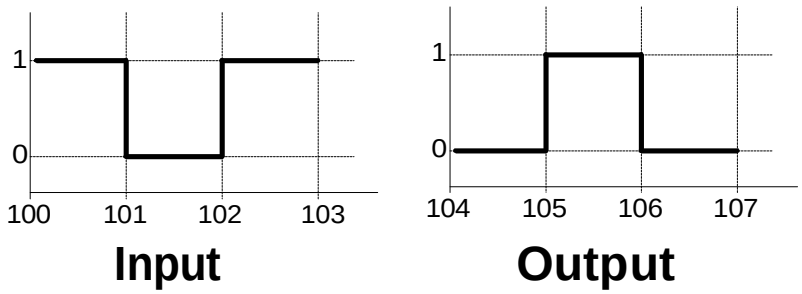

Figure 4-5. Input and Output Wave Forms.

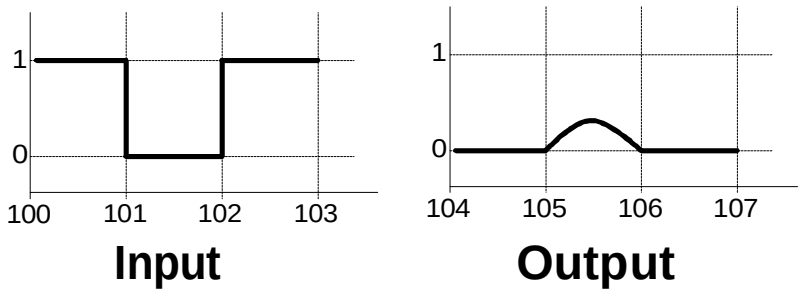

Figure 4-6. The Real Behavior of the Gate.

The real question raised by Figure 4-5 and Figure 4-6, is how to handle input pulses that are shorter than the delay of the gate. There are two different methods of handling such pulses, which result in two different delay models. In the Transport Delay model, the gate is viewed as a binary device which will eventually pass all changes in its inputs through to its outputs. Any pulse can be transported through the gate, regardless of its length. Algorithm 4-4 and Algorithm 4-5 implement the transport delay model.

The alternative to the Transport Delay model is the Inertial Delay model. The Inertial Delay model views a gate as a physical device which is not capable of changing state instantaneously. The device exhibits a resistance to change, which manifests itself as a delay between changes in the input and output. In Inertial Delay simulation, no gate is permitted to transmit a pulse shorter than its gate delay. Testing for changes occurs during gate processing, just as in unit-delay simulation. When a change is detected on the output of a gate, the event queue is checked for existing events. If an event already exists for the same net, both events are canceled, and no change will occur in the output of the gate.

Although the Inertial Model is a more accurate reflection of reality than the Transport Delay model, it is possible to make a case for using the Transport Delay model for simulation. Strictly speaking, the purpose of digital simulation is not to create an accurate model of the real world. The purpose of digital simulation is to detect and correct bugs in a circuit. A short pulse which is only slightly shorter than the gate delay, may cause unpredictable results in the real circuit. The short pulse may be recognized as a change by some gates and not by others. The short pulse may behave differently in different copies

of the same circuit. In such a case, it is better to transmit the pulse than to filter it out. On the other hand, a circuit designer may decide to use a large slow gate as a filter for hazards that occur in preceding stages of the circuit. If such hazards cause the circuit to malfunction, the Inertial Model must be used to simulate the correct behavior of the circuit.

4.4 The Sources of Delay

To fully understand the delay models that are used in digital simulation, it is necessary to understand the physical mechanisms through which delays occur. The basic element of most VLSI circuits is the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) illustrated in Figure 4-7.

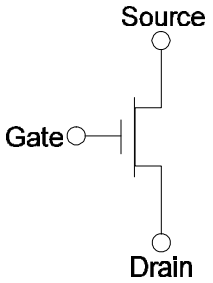

Figure 4-7. The MOSFET.

In one sense, the MOSFET behaves as a switch. In its default state the switch is open, and no current will flow between the source and the drain. However, placing a large enough charge on the gate will cause the switch to close, allowing a current to flow between the source and the drain. The time required to charge the gate is the most important source of delay in MOSFET circuits. Electrically, the gate of a MOSFET acts as if it were a capacitor in series with a resistor, as illustrated in Figure 4-8.

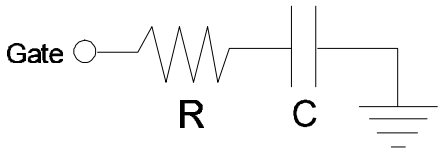

Figure 4-8. An RC Circuit.

When a voltage is applied to the gate, a charge will accumulate on the capacitor, but this will not happen instantaneously. The amount of time required to accumulate a specific charge **Q** depends on the applied voltage, the resistance **R**, and the capacitance **C**. In most digital circuits, the voltage is constant, so the determining factors are the resistance and capacitance. A MOSFET switch will close when the capacitor reaches a certain percentage of its maximum charge. The time required to reach this point increases with an increase in the resistance **R** or the capacitance **C**. When the percentage of maximum charge is graphed with respect to time, the result is an exponential curve such as that illustrated in Figure 4-9.

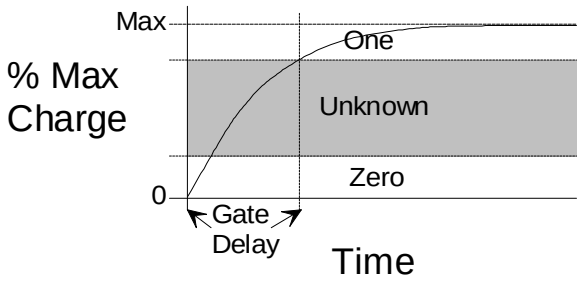

Figure 4-9. RC Circuit Timing.

As Figure 4-9 illustrates, the charge on the gate falls into three regions. To understand these three regions, it is necessary to visualize how the MOSFET will be used in a circuit. As Figure 4-10 illustrates, the source of the MOSFET will be attached to the gate of another transistor. The three regions in Figure 4-9 are determined by the effect that the MOSFET-A has on MOSFET-B. In the Zero region, the MOSFET-A is clearly in the non-conducting state and no current can flow to the gate of MOSFET-B. In the One region the MOSFET-A is clearly in the conducting state allowing current to flow to the gate of MOSFET-B. In the unknown region, the effect of MOSFET-A on MOSFET-B cannot be predicted with certainty.

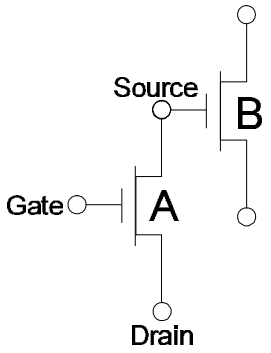

Figure 4-10. Two Connected MOSFETS.

Suppose the capacitor of Figure 4-8 is already charged to its maximum capacity. Suppose further that the voltage is removed from the gate terminal, and the terminal is connected to ground. The charge on the capacitor will dissipate as illustrated in Figure 4-11. This curve illustrates that gate delays are due to both the charging and discharging of capacitors in the circuit.

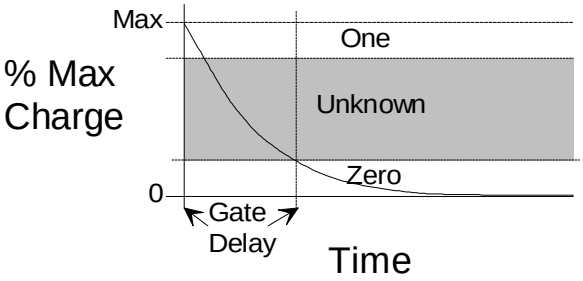

Figure 4-11. Capacitor Discharge Rate.

Today, the most common type of MOSFET circuit is the CMOS circuit. CMOS circuits contain two different types of MOSFETs, P-Channel transistors and N-Channel transistors. An N-Channel transistor requires a positive charge on its gate to become conducting, while the P-Channel transistor requires a negative charge. The transistors pictured in Figure 4-10 are N-Channel transistors. A P-Channel transistor is distinguished by a small circle on the gate, as illustrated in Figure 4-12.

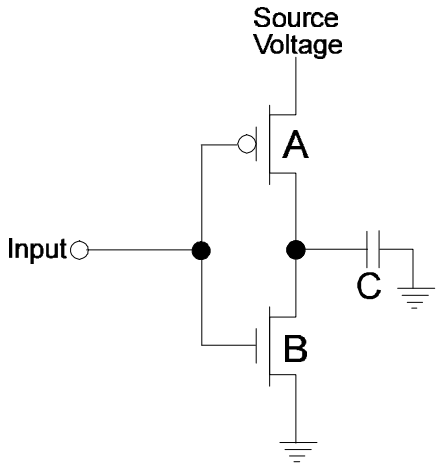

Figure 4-12. A CMOS Not Gate.

Figure 4-12 illustrates a CMOS Not gate. When the input of this gate is set to one, Transistor B will be conducting, and Transistor B will be non-conducting. The capacitor C will discharge through transistor B. When the input is zero, Transistor A will be conducting while Transistor B will be non-conducting. This will cause the capacitor C to charge through Transistor A. In any CMOS circuit, the components with the highest resistance are the transistors, even when they are in the conducting state. The Not gate of Figure 4-12 consists of two RC circuits, the charging (or pull-up) circuit consists of transistor A and capacitor C, while the discharging (or Pull-Down) circuit consists of transistor B and capacitor C. In most cases the two transistors will not have equal resistance. This implies that the charging rate of the capacitor will be different from the discharge rate. This in turn implies that the 0-1 output transition will not take the same amount of time as the 1-0 transition.

This section has covered only a few of the many factors that can affect the delay of a gate. Gate delays are not constant, but depend on the dynamic conditions that exist in the circuit as it is performing its function. Regardless of the complexity of the timing model, a discrete simulation that models net values as ones and zeros is not accurate enough to model all factors that could affect the speed of a gate. Nevertheless, the timing models used by discrete simulation can be made to give a reasonable approximation of circuit performance in all but the most demanding situations. One can increase the accuracy of discrete simulation by taking into account some of the issues discussed in this section.

4.5 Other Delay Models

In addition to the Inertial Delay and Transport Delay models, there are two other delay models that are commonly used in multi-delay and nominal-delay simulation. The first of these is the Ambiguous Delay model, which is based on the transition diagrams illustrated in Figure 4-9 and Figure 4-11. Rather than modeling a 0-1 (or 1-0) transition as a single change, each transition is modeled as two consecutive changes, a 0-Unknown change followed by an Unknown-1 change. This sequence of changes reflects the fact that in a real 0-1 change the signal will have an indeterminate value for a short period of time. An Ambiguous-Delay transition is illustrated in Figure 4-13. The Ambiguous Delay model is seldom used in practice.

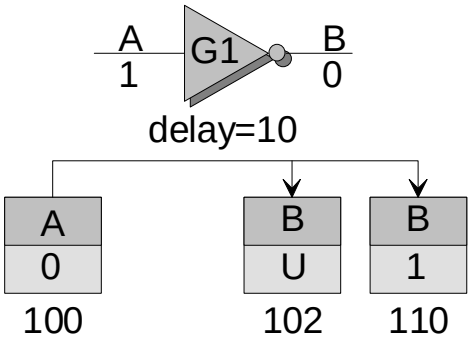

Figure 4-13. Ambiguous Delay

The second model is known as Rise/Fall Delay Model. This model allows different delays to be used for the 0-1 and 1-0 transitions of a gate. When an event occurs on the output of a gate, the delay depends on whether the new value is zero or one. When used in conjunction with the Transport Delay model, it is necessary to test new net values twice. During gate processing when the event is created, it is necessary to test the new value to determine the delay that should be associated with the event. During event processing it is necessary to test the net value again to determine if a change has actually occurred.

It is more common to use the Rise/Fall Delay Model with the Inertial Delay Model. The differing rise and fall delays cause no difficulties with event cancellation, since regardless of the difference in delay between an event and the complementary event that cancels it, the canceled event still represents an activity that was not allowed to complete normally. Figure 4-14 illustrates two transitions for the same net, with differing rise and fall delays.

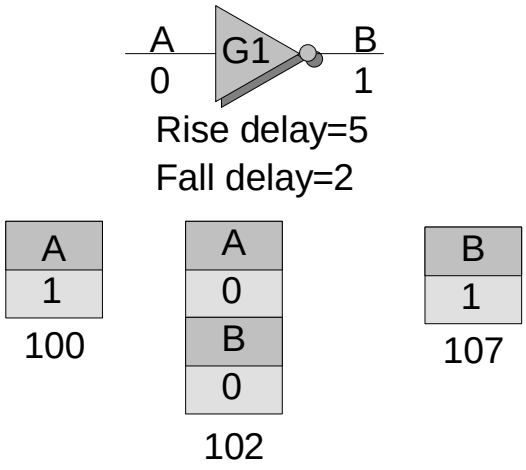

Figure 4-14. Rise/Fall Delays.

4.6 Timing Wheel Size

In section 4.2 it was shown that the timing wheel need be no larger than the maximum delay of any gate. Of course, one can use a larger timing wheel if there is a compelling reason to do so. For example, one might use a timing wheel size that was a power of 2 to simplify the remainder operation. In some cases, it is advantageous to use a timing wheel whose size is smaller than the maximum gate delay. This practice was common in the early days of simulation when memory sizes were small, but there are situations today in which a smaller timing wheel is desirable. Consider a circuit that contains several different types of elements, most of which are very fast but with a few that are very slow. Suppose the fast elements have delays of 32 or less, while the slow elements have delays exceeding 1000. If one were to use a full-sized timing wheel, most of the event queues would be empty most of the time. During simulation, each of these event queues must be examined for events. Each event queue requires one iteration of the main routine, even if it is empty, which can have a detrimental effect on performance.

An alternative is to ignore the slow gates, and create a timing wheel based only on the fast elements. A special Overflow Queue will be used to store events that won't fit in the timing wheel. The majority of the events will fit in the timing wheel and can be handled in the usual manner. The main procedure now consists of three phases, Event Processing, Gate Simulation, and Overflow Queue processing. During Overflow Queue processing, any events in the overflow queue that will now fit into the timing wheel are removed from the Overflow Queue and placed in their proper positions in the timing wheel. If the timing wheel becomes empty, and the overflow queue still contains some events, the current time is advanced to the earliest time in the overflow queue. To facilitate Overflow Queue processing it is necessary to add timing information to the event structure as illustrated in Figure 4-15.

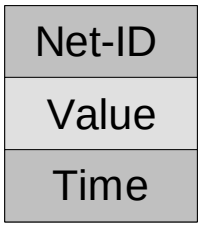

Figure 4-15. An Event Structure with Timing Information.

Adding the Overflow Queue does not affect the code of the Event Processor, but changes are required in both the Main Routine and the Gate Processor. Algorithm 4-6 and Algorithm 4-7 show the updated routines, with the changes from Algorithm 4-1 and Algorithm 4-5 double-underlined.

```
SimulateVector:
{
    GetInput( );
    RealTime := -1
    While Event_Queue is not empty
            and Overflow_Queue not empty do
        If Event_Queue is empty Then
            RealTime := Earliest time in Overflow_Queue
        Else
            RealTime := RealTime + 1
        EndIf
        CurrentTime := RealTime % MaxDelay
        For Each event E in Overflow_Queue do
            If Time of E = RealTime Then
                Add E to Event_Queue[CurrentTime]
            EndIf
        EndFor
        Event_Processor( );
        If Gate_Queue is not empty Then
            PrintIntermediateOutput( );
            Gate_Processor( );
        EndIf
    EndWhile;
    PrintFinalOutput( );
}
```

Algorithm 4-6. The Main Routine for Overflow Processing.

```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        D := The Delay of G
        Simulate G, put result in New_N
        /* Create Event Unconditionally */
        Create a new event E
        Net Identifier of E := N
        New Value of E := New_N
        If D ≤ MaxDelay Then
            Q := (CurrentTime+D) % MaxDelay
            Add E to Event_Queue[Q]
        Else
            Time of E = RealTime+D
            Add E to Overflow_Queue
        EndIf
        Remove G from Gate_Queue
    End For
}
```

Algorithm 4-7. Overflow Queue Gate Processor.

4.7 Nominal Delay Issues

In Nominal Delay simulation gate delays are fixed, but not necessarily integral. Although this seems to be a trivial distinction, there are hidden subtleties. When fractional delays are allowed, the range of different delay values is generally much larger than for integral delays. Even if the resolution of delays is limited, say to four digits, the number of different delays is larger by a factor of 10,000 over the same basic range. Even if a sufficiently large timing wheel can be used, most of the individual queues will be empty most of the time. This can lead to inefficient simulations.

When delays are specified as integers, the intent is usually to provide a model of the *relative* timing of various gates. When delays are expressed as real numbers, the intent is usually to create a model of the *actual* timing of the circuit components. In an actual circuit, the delay of a gate will vary over a range of values. The Nominal delay of the gate is the average of the delays exhibited by the gate. Since the units of delay represent real time units, it may be necessary to synchronize event and gate processing with the reading of new input vectors. In other words, it may be necessary to read a new input vector after a fixed number of time units, even if there are events remaining to be processed.

4.8 Priority Queues

Various techniques have been suggested for reducing the size of the timing wheel for Nominal Delay simulation. One technique is to provide a timing wheel slot for integral values only, and dynamically sort events as they are added to the individual queues. Each of the individual event queues is guaranteed to be in ascending order by time. Another approach is to use a wheels-within-wheels approach, where each slot in the timing wheel is a smaller timing wheel. The first technique is known to be an $O(n^2)$ algorithm, although the partitioning of events into different queues may enhance the performance to the point where it is acceptable. If there are k slots in the timing wheel, the performance will be $O((n/k)^2)$ which is still $O(n^2)$, but much more efficient that sorting the entire set of events. The second approach rearranges the timing wheel, but may not reduce the total size.

The priority queue is a dynamic sorting mechanism that does not suffer from the limitations of the timing wheel. The priority queue works equally well for decimal values and integer values. It is not as efficient as the timing wheel, because its performance is O(n lg n) while that of the timing wheel is O(n). The Priority Queue is modeled on a "nearly complete" binary tree, as illustrated in Figure 4-16.

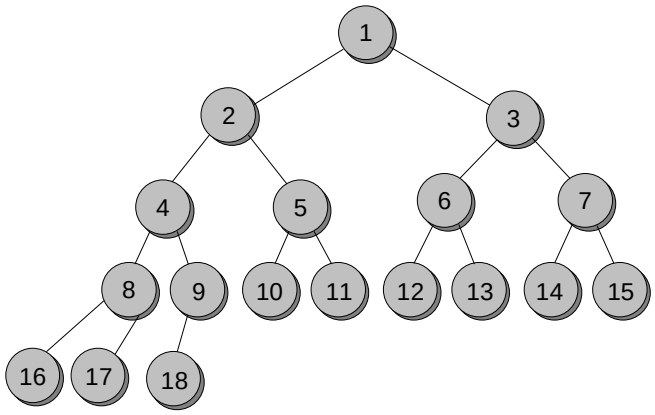

Figure 4-16. Priority Queue Structure.

The priority queue grows and shrinks during simulation. Vertices are added to the tree in the order indicated in Figure 4-16. Vertices are added level by level from left to right. When the tree shrinks, the most recently added vertex is removed first. Except for the lowest level, the tree will be a complete binary tree. If the lowest level is not complete, all vertices will be as far to the left as possible. Each of the vertices in the binary tree represents an event structure. Each of these structures will contain the usual information, plus the time at which it is to be processed. For every vertex, the time of the event in the vertex must be the same or earlier than the times in either of its children. This constraint is illustrated in Figure 4-17.

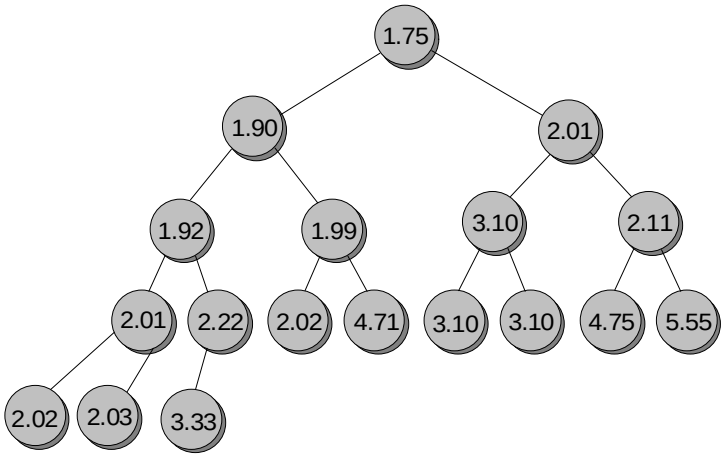

Figure 4-17. A Priority Queue with Timing Information.

It is obvious from Figure 4-17 that the event with the earliest time will be in the root vertex of the tree. As long as the priority queue structure is maintained, events can be processed in the correct order by repeatedly processing the event in the root vertex. The process has three steps, processing the root event, deleting a vertex, and rearranging the tree to preserve its properties. Figure 4-18 illustrates one cycle of event processing.

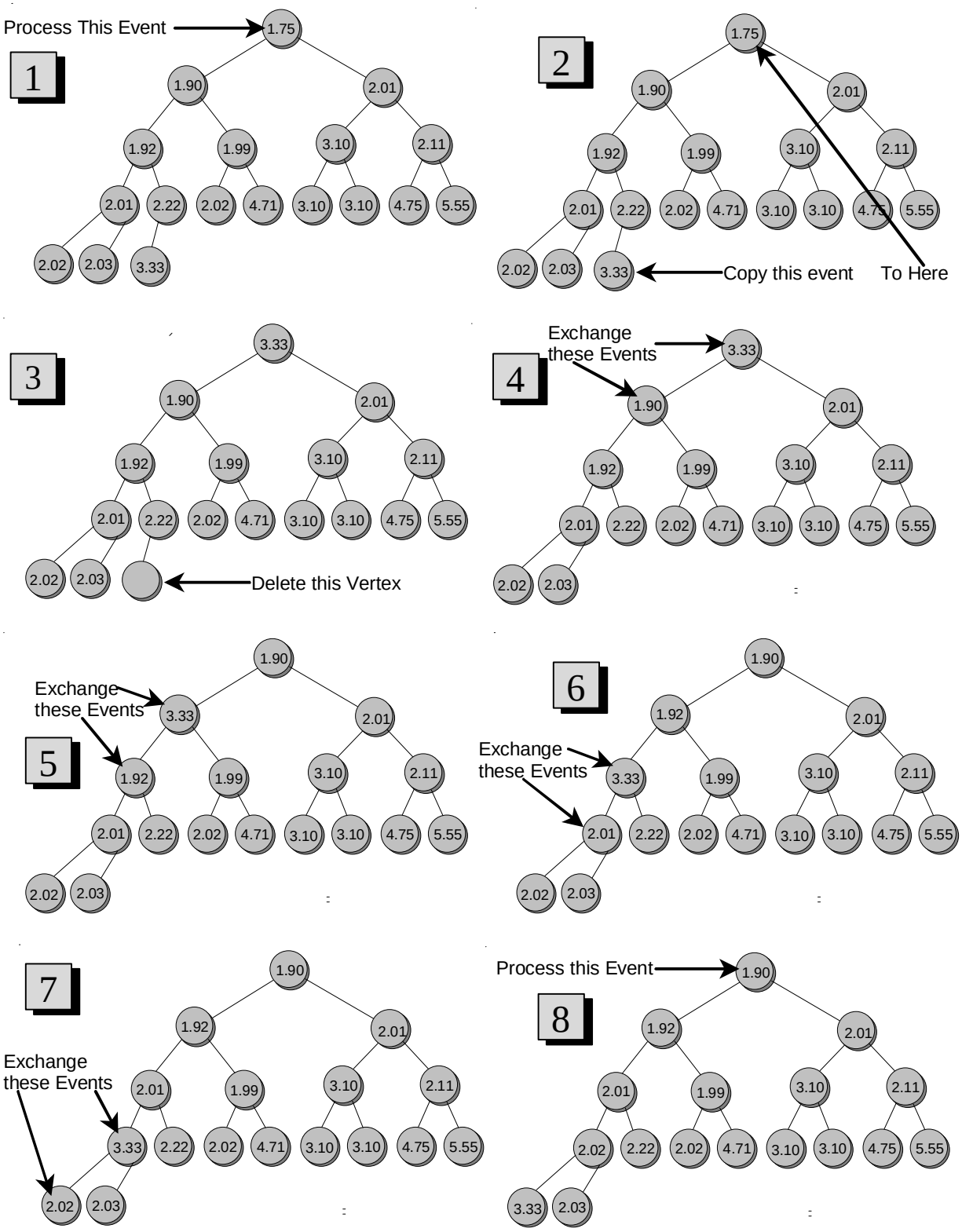

Figure 4-18. Deleting an Event From a Priority Queue.

To add a new event, it is necessary to add a new vertex to the tree, insert the new event into the new vertex, and then rearrange the tree so the times have the correct relationship. Figure 4-19 illustrates one cycle of this process.

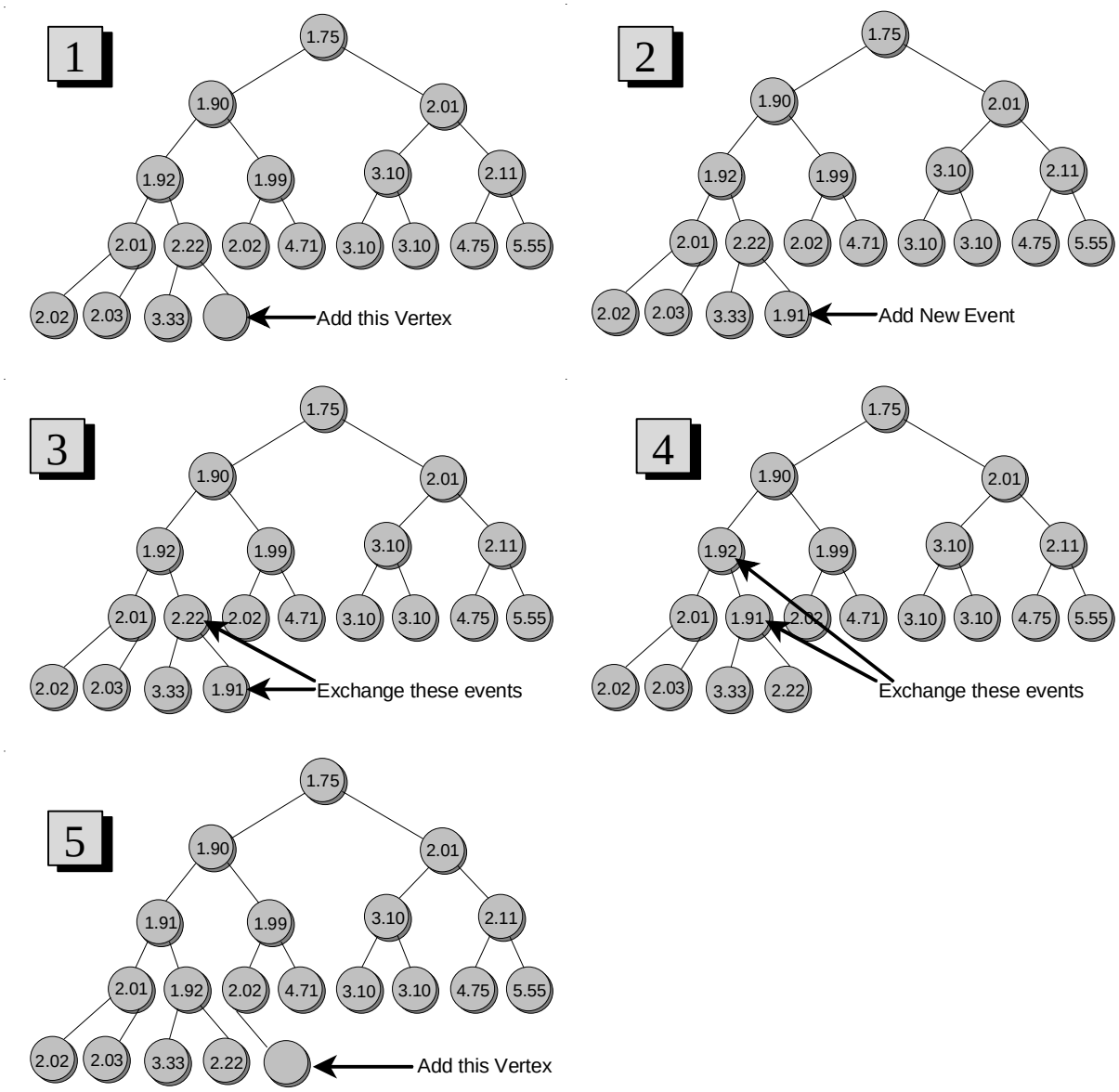

Figure 4-19. Adding an Event to a Priority Queue.

It should be obvious from these examples why the priority queue is not limited to integer values. However, without an good implementation, the priority queue could be so inefficient that any advantage would be negated by the additional overhead. Fortunately there is a simple, efficient implementation of the underlying binary tree that greatly simplifies the required processing. The binary tree is implemented as a simple array as illustrated in Figure 4-20.

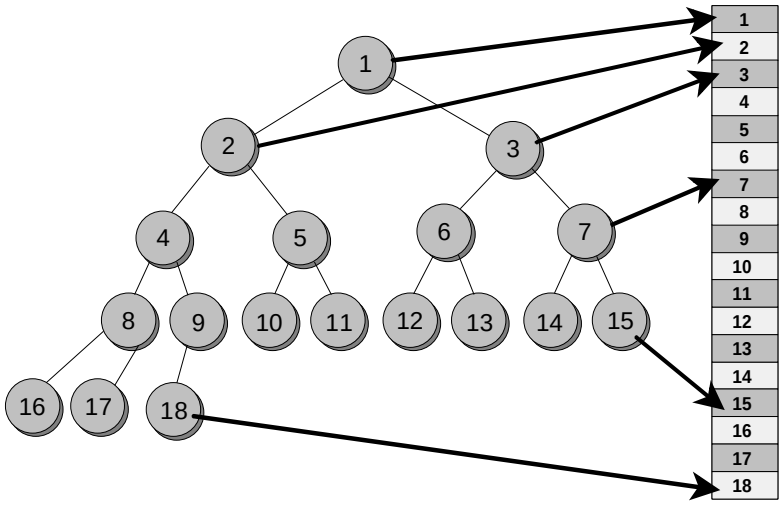

Figure 4-20. Priority Queue Implementation.

The underlying array is generally significantly larger than the average size of the binary tree. A *Size* variable is used to indicate the amount of the array that is currently occupied by the binary tree. Vertices are added and removed from the tree by changing the value of the *Size* variable. Moving through the tree is accomplished by computing new indices from the index of the current vertex. To find the left child of a vertex, the index of the current vertex is multiplied by two. The index of the right child is one larger than the index of the left child. To obtain the index of the parent, the index of the current vertex is divided by 2, and the remainder is discarded. Figure 4-21 illustrates these computations.

```
LeftChildIndex := 2*CurrentIndex
RightChildIndex := 2*CurrentIndex + 1
ParentIndex := CurrentIndex/2   /* integer division */
```

Figure 4-21. Priority Queue Indexing Operations.

The process of deleting an event from the queue, and rearranging the queue, is illustrated in Algorithm 4-1. The priority queue is implemented as an array of structures in the following format.

```
DeleteEvent:
{
    If Size = 3 Then
        Size := 0
    Else
        Queue[1] := Queue[Size]
        Size := Size - 1
        Done := False
        Current := 1
        While not Done do
            Left := 2*Current
            Right := Left+1
            If Left > Size Then
                Done := True
            Else
                SmallChild := Left
                If Right ≤ Size Then
                    If Queue[Right].Time < Queue[Left].Time Then
                        SmallChild := Right
                    EndIf
                EndIf
                If Queue[SmallChild].Time < Queue[Current].Time Then
                    TempEvent := Queue[SmallChild]
                    Queue[SmallChild] := Queue[Current]
                    Queue[Current] := TempEvent
                    Current := SmallChild
                Else
                    Done := True
                EndIf
            EndIf
        EndWhile
    EndIf
}
```

Algorithm 4-8. Deleting the Root Event from a Priority Queue.

Algorithm 4-8 illustrates the removal of an event from the root of the priority queue. The only node that can legitimately be removed from the tree is the vertex the last vertex, that is, the vertex with the highest index. Therefore, instead of removing the root vertex, the event in the last vertex is copied into the root vertex, overwriting its previous contents, and the last vertex is removed. This leaves the priority queue in an invalid state, since the event times of the children of the root are no longer earlier than the time of the event in the root. To correct this situation, the event in the root is pushed down into the tree until it reaches the bottom, or until the event times have the correct relationship. To push the event down the tree, it is necessary to exchange the event in a vertex with an event in one of its children. When choosing a child for exchange, it is necessary to choose the child

with the earliest event time. The SmallestChild variable in Algorithm 4-8 is used to select the child with the earliest event time. Once the smallest child has been determined, it is necessary to compare the event time in the vertex with the event time of the smallest child. If the event time in the vertex is earlier, then no exchange is done, because the tree structure is already valid. Otherwise the event in the vertex is exchanged with the smallest child, and the process continues at the smallest child.

An event is added to the event queue by adding a vertex to the underlying tree, and placing the new event into the new vertex. The only place a vertex can be added is at the end, so a new vertex is added to the end, and the new event is copied into it. Unfortunately, the event time of the parent of this new vertex is not guaranteed to be earlier than the event time of the new event. To correct the relationship between event times, the new event must be pushed up the tree until it reaches the root, or until the relationship of the event times is correct. To do this, the event-time of the vertex is compared to the event-time of the parent. If the parent event-time is later, then the two events are exchanged, and the process is repeated with the parent. Algorithm 4-9 shows the procedure for adding a new event to the priority queue.

```
AddEvent(NewEvent):
{
    Size := Size + 1
    Queue[Size] := NewEvent
    Current := Size
    Done := False
    While not Done do
        Parent := Current / 2
        If Parent = 0 Then
            Done := True
        Else
            If Queue[Parent].Time > Queue[Current].Time Then
                TempEvent := Queue[Parent]
                Queue[Parent] := Queue[Current]
                Queue[Current] := TempEvent
                Current := Parent
            Else
                Done := True
            EndIf
        EndIf
    EndWhile
}
```

Algorithm 4-9. Adding an Event to a Priority Queue.

In a timing-wheel implementation, there is an independent clock that is used to drive the processing of the timing wheel. In a priority-queue implementation there is no independent clock. Instead, the priority queue itself determines the current simulation time. The current simulation time is always equal to the event time of the root event in the

priority queue. Substantial changes are required in the main simulation routine, as illustrated in Algorithm 4-10. Changes are also required in the Gate Processor and Event Processor routines, which are illustrated in Algorithm 4-11 and Algorithm 4-12.

```
SimulateVector:
{
    ReadInputVector
    CreateInitialEventSet
    CurentTime := 0
    While Size > 0 do
        ProcessEvent(Queue[1])
        RemoveEvent
        If Size = 0 or Queue[1].Time ≠ CurrentTime Then
            GateProcessor
            CurrentTime := Queue[1].Time
        EndIf
    EndWhile
}
```

Algorithm 4-10. The Main Routine for Priority Queues.

```
GateProcessor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        D := The Delay of G
        Simulate G, put result in New_N
        Net Identifier of TempEvent := N
        Value of TempEvent := New_N
        Time of TempEvent := CurrentTime+D
        AddEvent(TempEvent)
        Remove G from Gate_Queue
    End For
}
```

Algorithm 4-11. The Gate Processor for Priority Queues.

```
ProcessEvent(E):
{
    N := Net Identifier of E
    If New Value from E <>  Net Table value for N Then
        Copy New Value from E to the Net Table entry for N.
        For each Gate G in the fanout of N do
            If G is not already in Gate Queue Then
                Add G to Gate Queue
            End If
        End For
    End If
}
```

Algorithm 4-12. Event Processor for Priority Queues.

The Priority Queue algorithms given in this section implement the transport delay timing model. Implementation of the inertial delay model introduces the additional complication of deleting events from the middle of the priority queue. Deleting an event from the middle of the queue requires a combination of Algorithm 4-8 and Algorithm 4-9. The first steps are to copy the last event in the queue into the location occupied by the deleted event, and to reduce the size of the queue by one. It is then necessary to rearrange the queue so the event times have the correct relationship. Three mutually exclusive conditions may exist between the event that has been moved and its neighbors. If the event is earlier than its parent it must be moved up the tree as in Algorithm 4-9. If the event is later than one of its children it must be moved down the tree as in Algorithm 4-8. If neither condition holds, then the event is already in the correct position, and no corrective action is necessary.

4.9 Model Accuracy

When the delays modeled by the simulator are intended to represent real timings, it is necessary to consider the synchronization of input vectors with the processing of internal events. All of the algorithms considered so far treat each vector independently. The events generated by one input vector are completely processed before the processing of the next input vector begins. However, if gate delays represent real time intervals, it may take longer than one clock phase to complete the processing of events generated by an input vector. In such cases, events from one vector may interact with events from a subsequent vector. To model this phenomenon, it is necessary to guarantee that input vectors are processed at the correct instants of simulated time.

There are several methods for synchronizing input vector processing with event processing. One straightforward method is to treat input vector processing as a special type of gate simulation, and schedule these “gate simulations” with a series of dummy events. To facilitate the scheduling of dummy events, it is necessary to create a dummy net which is the input to a dummy gate of type “ReadNewVector.” It is necessary to bootstrap the simulation by reading and processing the first input vector before the normal simulation process begins. Note that since comparison of net values takes place when an

event is processed rather than when it is created, each value in an input vector translates into one event. Before simulation begins, the event queue is bootstrapped with the events from the first input vector, and a dummy event to trigger the reading of the second input vector. Dummy Event appears to be the input to a gate of type "ReadNewVector." Algorithm 4-13 illustrates the main routine for this type of simulator.

```
MainRoutine:
{
    Read InputVector
    If not EOF detected Then
    Convert InputVector to InputEvents
    Queue InputEvents at time Zero
    DummyNet.Value := 0
    DummyEvent.Value := 1
    Queue DummyEvent at time Zero
    While Size > 0 do
        ProcessEvent(Queue[1])
        RemoveEvent
        If Size = 0 or Queue[1].Time ≠ CurrentTime Then
            GateProcessor
            CurrentTime := Queue[1].Time
        EndIf
    EndWhile
}
```

Algorithm 4-13. Main Routine for Input Synchronization.

Algorithm 4-13 assumes that a priority queue us used for the event queue. The GateProcessor routine will be identical to that shown in Algorithm 4-11. The "**Simulate** *G*" step, which will normally be a subroutine containing a large Case statement to decode the gate type, must be enhanced with simulation code for the gate type "ReadNewVector." This code is shown in Algorithm 4-14.

```
Case ReadNewVector:
{
    Read InputVector
    If not EOF detected Then
        Convert InputVector to InputEvents
        Queue InputEvents at time CurrentTime+PhaseLength
        DummyNet.Value := 0
        DummyEvent.Value := 1
        Queue DummyEvent at time CurrentTime+PhaseLength
    EndIf
}
```

Algorithm 4-14. Reading Successive Input Vectors.

4.10 Zero Delay Elements

All algorithms discussed so far in this chapter assume that every gate has a delay greater than zero. At times, it is advantageous to assign a delay of zero to some gates. Zero delay elements cannot be handled in the same way as elements with non-zero delay, because the input and output events of the elements must be processed at the same instant of simulated time, with an intervening gate simulation. For simulators that use a timing-wheel event queue, a different method must be used to advance from one queue to the next. To avoid scheduling conflicts, it is necessary to add one additional slot to the timing wheel. Events generated by zero-delay elements must be scheduled in the current event queue. The simulator must not advance to the next event queue until the current event queue is empty. If a priority queue is used for scheduling events, no special changes are required in event handling, but it is necessary to simulate any zero delay elements before allowing simulation time to advance.

The biggest problem with zero delay elements is illustrated in Figure 4-22. Figure 4-22 illustrates three zero-delay elements of a larger circuit. If Nets A and B change simultaneously from zero to one as indicated, both G1 gate and G2 will be simulated at the same instant of time. No events have yet been generated for nets C and D, so the simulation of G2 gate will produce an output of one. The simulation of G1 and G2 will produce events on nets C and D. This will cause simulations to be performed for G2 and G3. D will return to its original value of zero, but not before the gate G3 is simulated. Assuming the clock is active, E will toggle, even though no change has occurred in D. The two changes in net D do not represent a hazard, because they occur simultaneously. This type of change is known as a Delta Pulse. A delta-pulse differs from a hazard because a delta-pulse has zero length, while a hazard has a length greater than zero. A Delta-Pulse does not represent a real change in the circuit, and must not be allowed to affect the permanent state of the circuit.

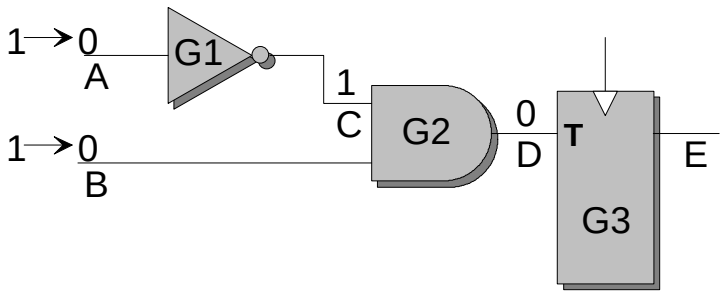

Figure 4-22. The Delta Pulse Phenomenon.

The real problem with simulating zero-delay elements using the standard event-handling mechanisms, is that the result will be an algorithm that uses the two-phase scheduling mechanism of unit-delay event-driven simulation. Although the current simulation time is held constant, the simulation of the zero delay elements will be simulated using a unit-delay model. (Strictly speaking, the model will either be pure Unit-Delay, or a mixture of the Unit-Delay and Zero-Delay models, depending on the scheduling of gate simulations and the handling of net values.) To handle zero-delay elements, it is necessary to use a true zero-delay simulation for the zero delay elements.

As in zero-delay simulation, the circuit must first be levelized. To avoid problems with asynchronous sequential circuits, it is possible to do a piecewise levelization. In piecewise levelization, all circuit elements with non-zero delay are eliminated from the levelization process. This leaves isolated clumps of zero delay elements that can be levelized independently of one another.

The main scheduling queue can be either a timing wheel or a priority queue. Two gate queues must be used, one for zero-delay elements and one for gates with non-zero delay. The zero-delay gate queue is a levelized queue such as that described in Chapter 2, while the other queue is identical to that typically used in multi-delay simulation. Zero delay elements are scheduled in the levelized gate queue, in the position corresponding to the level of the gate. Gates with non-zero delay are scheduled in the usual manner. During the gate simulation phase of the simulation, the levelized queue is processed first. Only when the levelized queue is empty, will gates in the multi-delay queue be simulated.

When a zero-delay element is simulated, no new event will be created. Instead, the values of the output nets will be updated immediately, and any gates in the fanouts will be scheduled for simulation. As with other gate simulations, zero-delay elements will be scheduled in the levelized queue, and gates with non-zero delay will be scheduled in the multi-delay queue. After the levelized queue becomes empty, the multi-delay queue will be processed. When the multi-delay queue becomes empty, event-processing resumes.

4.11 Fanout Delays

In some cases the gate-delay model may not provide an accurate enough model of how delays actually occur in a circuit. To illustrate, consider the circuit of Figure 4-23. As mentioned in Section 4.4, the state of net X is not determined not by a certain voltage, but by the effect of the net X on gates G2, G3, and G4. Some gates may be more sensitive to input changes than others, and some gates may be more sensitive to certain inputs than to others. For very high speed circuits, the propagation delay of the electric signal along the conductor may contribute significantly to the delay of the circuit. Because of these factors, it may be desirable to assign different delays to the different fanout branches of net X.

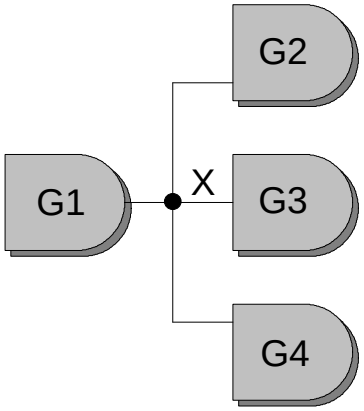

Figure 4-23. Fanout Delays.

Differing fanout delays can be modeled in several different ways. One method is to schedule an event for each fanout-branch of a net, rather than for each net. This allows each fanout branch to have a different delay. It is also possible to model such delays without altering the simulator. Instead, the circuit will be enhanced with additional gates. Assume that in Figure 4-23, the G2 fanout branch has delay 3, the G3 branch has delay 2, and the G1 branch has delay 1. The gate G1 is assigned the minimum delay 1, and the remainder of the circuit is enhanced with non-inverting buffers as illustrated in Figure 4-24.

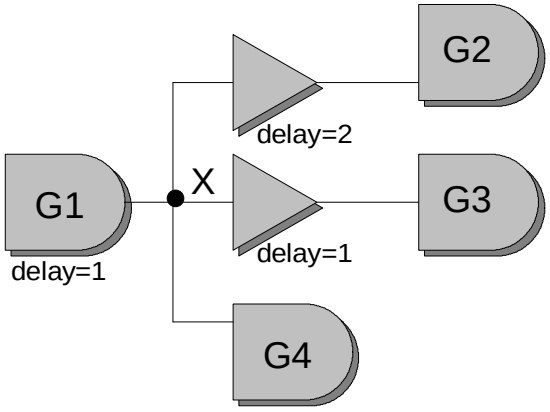

Figure 4-24. Simulating Fanout Delays.

4.12 Summary

4.13 Exercises