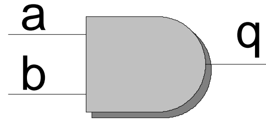
Chapter 3

Event Driven Simulation

3.1 Introduction

The simulation techniques described in Chapter 2 simulate every gate for every input vector. This is wasteful because if the inputs of a gate don't change, then the output doesn't change. When a gate is simulated twice with the same inputs there is no need to perform the second simulation, because the output of the gate will not change after the first simulation. By the same token, if a circuit is simulated using two consecutive identical input vectors, there is no need to simulate the second vector, because nothing will change after the first vector.

If we can avoid simulating those gates whose inputs do not change, we can improve speed. However, a naive approach will not help, because the amount of work required to test the inputs of a gate for changes exceeds the amount of work needed to simulate the gate. Figure 3-1 illustrates the difficulty. The code with tests will execute at least six instructions and may execute as many as nine instructions, whereas the code without tests executes only three instructions.



Without Tests		With Tests	
High-Level	**Assembly**	**High-Level**	**Assembly**

q = a And b;	load r1,a
	and r1,b
	store r1,q
		if a ≠ Old_a Or
		b ≠ Old_b then
		q = a And b;
		endif
			load r1,a
			cmp r1,Old_a
			jne simgate
			load r1,b
			cmp r1,Old_b
			je nosim
			simgate:
			load r1,a
			and r1,b
			store r1,q
			nosim:

Figure 3-1. Naive Testing for Changes.

Fortunately, it is not always necessary to test the inputs of every gate for changes. Consider the circuit of Figure 3-2. If the inputs A, B, and C don't change, the outputs X1 and X2 will not change. It is not necessary to test X1 unless A or B changes. By the same token, it is not necessary to test X2 unless B or C changes. By carefully organizing our tests we can eliminate not just gate simulations, but other input tests as well. If we are to do this successfully we must plan our approach carefully to avoid undue complexity.

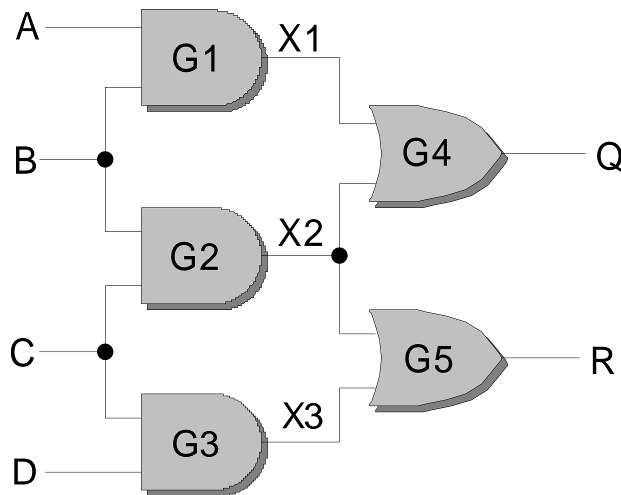


Figure 3-2. Change Propagation.

3.2 Basic Principles

Event-Driven Simulation is designed to eliminate unnecessary gate simulations without introducing an unacceptable amount of additional testing. It is based on the idea of an *event*, which is a change in the value of a net. In Figure 3-2, an event for net A or B may cause an event on net X1, however if there is no event on either A or B, there can be no event on X1. In event driven simulation, we represent each event as a data structure, and use these data structures to trigger the simulation of gates and the creation of other events.

An event is represented as a data structure similar to that of Figure 3-3. The scheduling information is used to link the data structure into queues, and for other behind-the-scenes purposes. In most cases, we will omit the discussion of this portion of the data structure.

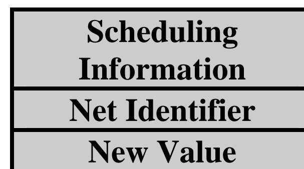


Figure 3-3. The Event Structure.

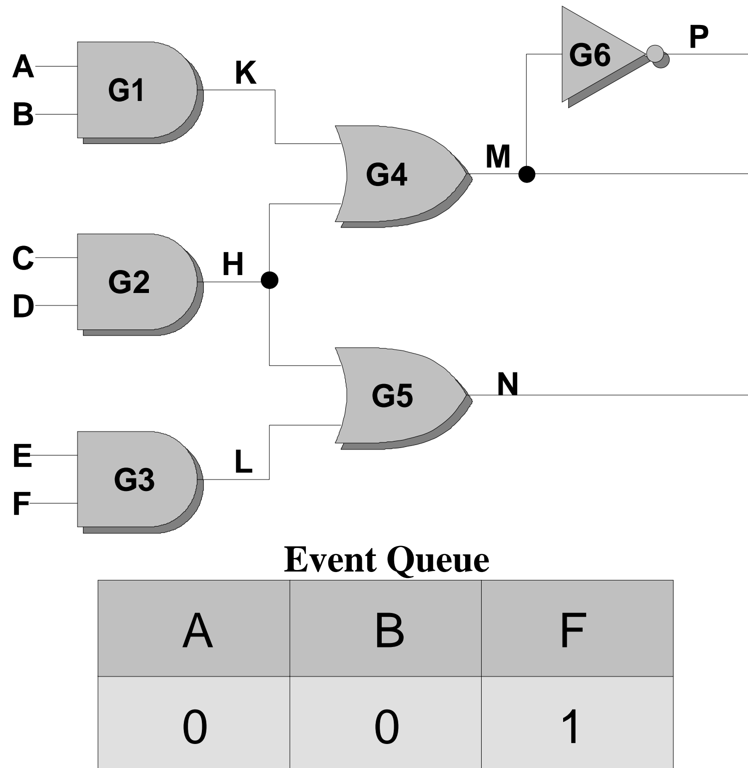
During simulation, events trigger gate simulations and gate simulations produce events. If no events occur then no gates will be simulated. The initial set of events is created by comparing each bit of an input vector with the corresponding bit in the previous input vector. An event is created for each pair of bits that is different. Thereafter, nets are tested for changes only after a gate simulation. When an event is processed, any gates that use the net as an input will be scheduled for simulation. The order in which gate simulations are performed cannot be predicted ahead of time, so dynamic queues are used to schedule both event processing and gate simulation.

When an event is detected, an event structure will be created and stored in the *Event Queue* for future processing. Algorithm 3-1 illustrates the algorithm for creating the first set of events.

```
Get_Input:
{
    For I = 1 to NumberOfInputs Do
        Temp = ReadPrimaryInputValue(I)
        If Temp ≠ PrimaryInput(I) Then
            EventPointer = NewEvent( );
            EventPointer->NetID = I;
            EventPointer->Value = Temp;
            QueueEvent(EventPointer);
        EndIf
    EndFor
}
```

Algorithm 3-1. Input Vector Processing.

To illustrate the action of Algorithm 3-1, suppose the circuit of Figure 3-4 were simulated with two consecutive input vectors, (1,1,0,0,1,0), and (0,0,0,0,1,1). These two vectors differ in three positions, therefore, three events must be created: one for A, one for B, and one for F. These events will be inserted into the event queue as illustrated in Figure 3-4.



A	B	F
0	0	1

Figure 3-4. A Circuit with an Event Queue.

Once a set of events has been accumulated by Algorithm 3-1, it is necessary to process each event. Note that all new net values are inserted into the event structure not into permanent storage. This allows gate-simulations to see a consistent set of net values but adds an additional step to event processing. The first part of event processing is to copy net values from the event structures into permanent storage. The second part is to schedule the simulation of all gates that use the net as an input. A gate is scheduled placing it (or a pointer to it) into the *Gate Queue*. The event processing algorithm is given in Algorithm 3-2. This algorithm assumes that two tables, called the *Net Table* and the *Gate Table*, are being used to store the details of the circuit. The Net Table entry is assumed to have space for permanent storage of the net value.

```
Event_Processor:
{
    For each Event E in Event_Queue 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
        End For
        Remove E from Event Queue
    End For
}
```

Algorithm 3-2. Event Processing.

As Algorithm 3-2 shows, event processing continues until all events have been processed and removed from the Event Queue. At this point, there will usually be several gates in the gate queue. Figure 3-5 shows the gate queue after processing the events of Figure 3-4.

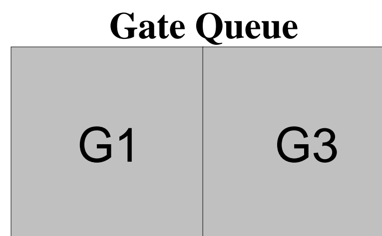


Figure 3-5. A Sample Gate Queue.

Once all events have been processed, it is necessary to simulate each gate in the gate queue, test outputs for changes, and schedule events. The details of gate simulation are illustrated in Algorithm 3-3. It is assumed that a gate simulation routine is available for simulating individual gates.

```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        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
        End If
        Remove G from Gate_Queue
    End For
}
```

Algorithm 3-3. Gate Processing.

Algorithm 3-3 creates new events and places them in the Event Queue. Algorithm 3-2 must then be called to process these events. Algorithm 3-2 and Algorithm 3-3 must be called repeatedly until there are no more events to process. Simulation of an input vector consists of a single call to the Algorithm 3-1, and several calls to Algorithm 3-2 and Algorithm 3-3. Algorithm 3-4 gives the structure of the main routine.

```
SimulateVector:
{
    GetInput( );
    While Event_Queue is not empty do
        Event_Processor( );
        If Gate_Queue is not empty Then
            PrintIntermediateOutput( );
            Gate_Processor( );
        EndIf
    EndWhile;
    PrintFinalOutput( );
}
```

Algorithm 3-4. The Main Simulation Routine.

3.3 Timing Models

Unlike levelized simulation which provides a zero-delay model, the event-driven technique provides a unit-delay model, in which all gates are assumed to have a delay of one. To illustrate, consider the circuit of Figure 3-6. Assume that the nets have the values shown and that it is now being simulated with the input A=1, B=1. In levelized simulation the output Q will remain at zero. However, this will not be true in event driven simulation, as Figure 3-7 shows.

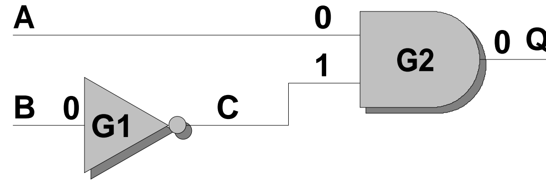


Figure 3-6. A Sample Circuit.

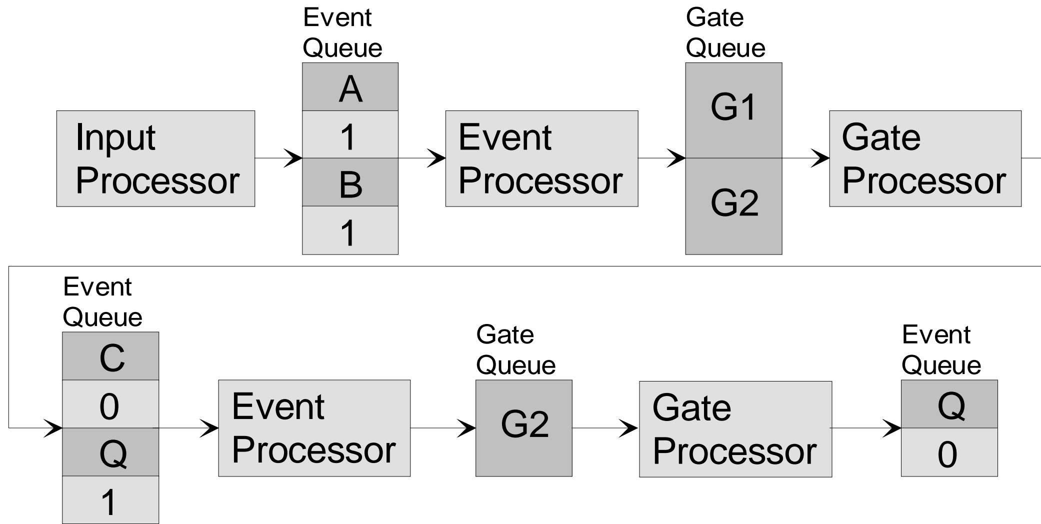


Figure 3-7. An Event-Driven Simulation.

Figure 3-7 demonstrates several important points about event driven simulation. First, note that G2 is simulated immediately without waiting for the simulation of G1, as would be the case in levelized simulation. Also note that G2 is simulated twice, instead of once. Most importantly, there are two events for the net Q, one which changes the value from zero to one and the second which changes the value from one to zero. The value of Q changes briefly from zero to one and back to zero. This sort of change is called a *static hazard*. If Q were the input of a T flip-flop, the hazard could result in unwanted state changes. Levelized simulation is incapable of detecting this change.

It is the two-phase structure of the event-driven simulation algorithm that imposes the time scale on the simulation. Each execution of the Event Processor subroutine defines one unit of simulated time. Intermediate output vectors can be produced after each execution of this routine.

The first execution of the Event Processor is designated as simulated time zero. Simulated time increases by one for each subsequent execution of the Event Processor. It is usually assumed that an arbitrary number of simulated time units can occur between successive input vectors. It is possible to impose stricter timing on input-vector processing by reading and processing a new vector after a fixed number of simulated time units, but this is seldom done in the unit-delay timing model. More detailed timing models may impose such a restriction. (See Chapter 4.)

The unit-delay timing model is also the motivation for storing the intermediate value of a net in the event structure rather than immediately updating the net value. Suppose the gate simulation queue contains the three gates shown in Figure 3-8. Suppose further that

net values are updated immediately after each gate simulation, and that all four primary inputs change value. All three gates are simulated at time zero and let us assume they are simulated in lexicographic order. Gate G2 will be simulated after G1 and before G3. If the simulation of G2 causes a change in the net X1, the simulations of G1 and G3 will use two different values of X1 for their simulations. G2 will appear to have a delay of one for the simulation of G1, and a delay of zero for the simulation of G3. Correct unit-delay simulation requires the value of X1 to be held constant until after the simulation of G3. Storing the new net value in the event structure accomplishes this.

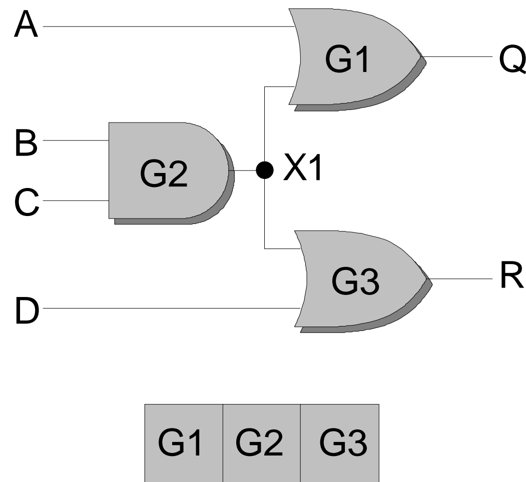


Figure 3-8. Net Update Conflicts.

State changes such as that exhibited by the net Q in Figure 3-7 are known as *hazards*. There are two varieties of hazards, static hazards such as those shown in Figure 3-9 and dynamic hazards which are given in Figure 3-10. In a static hazard, a net changes value more than once, but eventually returns to its original value. In a dynamic hazard, a net changes more than once, and the net eventually stabilizes at the opposite value from which it started. Hazards can cause problems in the construction of asynchronous sequential circuits. Because levelized simulation cannot detect hazards, event driven techniques are preferred for simulating asynchronous circuits.

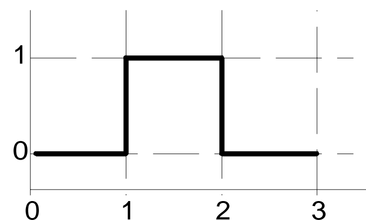


Figure 3-9. A Static Hazard.

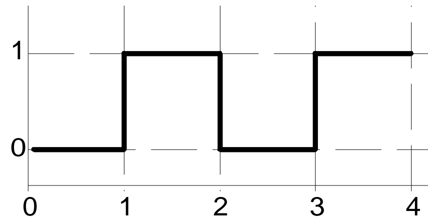


Figure 3-10. A Dynamic Hazard.

3.4 Initialization

All examples discussed so far have assumed that two consecutive vectors have been simulated, and have detailed process of simulating the second of these vectors. This begs the question of what to do for the first input vector. Before the first vector is simulated, it is necessary to initialize each net to some value. Suppose all nets are uniformly initialized to zero. This can cause errors in simulation, as Figure 3-11 illustrates.

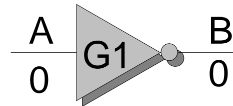


Figure 3-11. An Inconsistently Initialized Circuit.

Suppose the circuit of Figure 3-11 is simulated with a series of zero inputs. Because the net A does not change value, no events will be scheduled. Because no events are scheduled, gate G1 will never be simulated. Because gate G1 is never simulated, the output B never changes from zero to one. The output of the simulator will be incorrect for these inputs. The problem is that initializing all nets to zero causes the net values to be inconsistent with the logic of the circuit. The only reasonable way to obtain a set of consistent initialization values is through a simulation of the circuit.

There are two ways to solve this problem. The first is to change to a three-valued logic model that includes the value U (Unknown). All nets are initialized to the unknown value as illustrated in Figure 3-12. Regardless of whether the first input is one or zero, it will be different from U, so an event will be queued for net A. This will cause G1 to be simulated, and a consistent value to be calculated for net B.

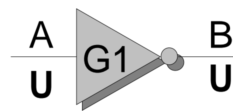


Figure 3-12. A Circuit initialized with Unknowns.

The disadvantage of three-valued simulation is that the simulation code is much more complicated than that for two-valued logic. A second method for achieving the same goal is to mark each net at the beginning of the simulation, and modify the criteria for creating events. In place of the simple test for changes, an event will be created if the net value has changed or if the net is marked. When an event is queued for the net, the mark is removed. From a scheduling point of view, this method is identical to using unknowns, since the mark can be considered to be an “Unknown indicator.” However, this technique

uses the simpler two-valued simulation code. If it is necessary to process Unknown input values, three-valued logic must be used. Otherwise, net-marking will be more efficient.

Regardless of which initialization method is used, it is necessary to take special care with constant-one and constant-zero signals. Figure 3-13 illustrates why this is so.

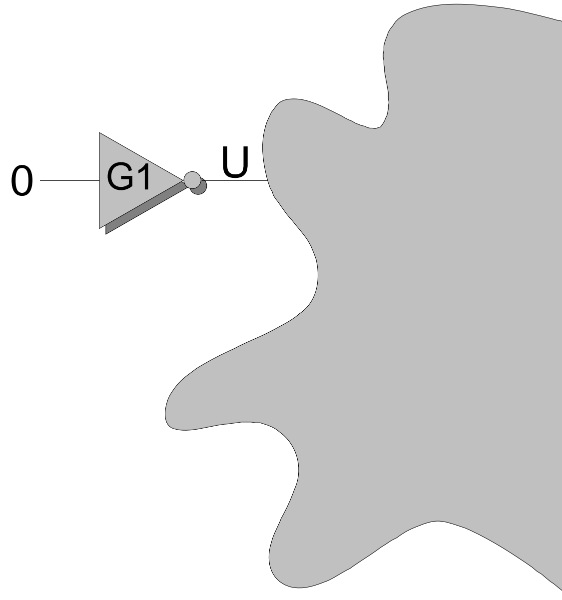


Figure 3-13. Constant Zero Signals.

In Figure 3-13 the input of G1 is a constant-zero signal, and will never change. Because the input of G1 never changes, G1 will never be simulated. Because G1 is never simulated, the output will remain unknown. It is necessary to force the simulation of G1 at least once. This is done by queuing an event for each constant-zero and constant-one on the first input vector. The simplest approach is to pre-queue these events before the first input vector is processed.

3.5 Sequential Circuits

Event driven simulation is capable of handling sequential circuits of all types with no special provisions other than testing for oscillations. To illustrate, consider the circuit of Figure 3-14. If the circuit is in the state pictured in Figure 3-14, the input vector s=0,r=1 will be simulated as illustrated in Figure 3-15.

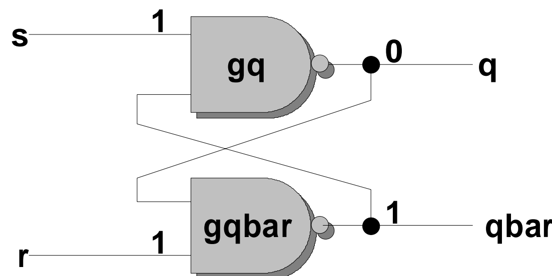


Figure 3-14. An Asynchronous Circuit.

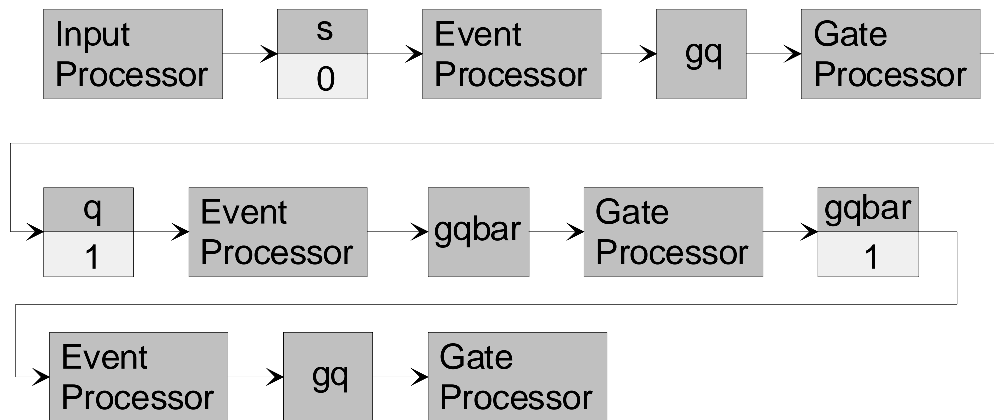


Figure 3-15. Simulating an Asynchronous Circuit.

Synchronous sequential circuits can also be simulated with no modifications to the basic procedure. Furthermore, it is possible to verify the synchronous nature of a circuit, which cannot be done using levelized simulation. The primary difficulty in the simulation of sequential circuits is the detection of oscillations. Although a careful analysis of the gates being simulated would allow the simulator to detect an oscillation, it is more efficient to place an ad-hoc limit on the number of gates simulated per input vector. In a well-designed circuit, no gate should be simulated more than two or three times. By placing a limit on the number of gates simulated, such as 10 times the number of gates in a circuit, an oscillation can be safely reported when an excessive number of gates have been simulated.

Breaking the simulation at this point will leave the simulation in an inconsistent state. If it is desired to continue the simulation beyond the point at which an oscillation is detected, it is safest to substitute unknown values for all nets whose values are oscillating. One way to do this is to continue the simulation beyond the state where the oscillation was detected. The simulation should continue until the simulator enters precisely the same state as it was in when the oscillation was detected. Any net for which an event was processed should be set to the unknown value. A faster technique is to cancel all events that are in the event queue when the oscillation is detected, and set the values of the corresponding nets to unknown.

3.6 Activity and Efficiency

Because basic event-driven simulation is capable of detecting hazards, it is viewed as being more accurate than levelized compiled code. Although the unit-delay model is indeed a more accurate representation of reality than the zero-delay model, it is too crude to detect all problems that might occur in an asynchronous sequential circuit. The primary reason for using event-driven simulation is efficiency. Unfortunately, it is difficult to compare the efficiency of the two algorithms, because the performance of event-driven simulation depends very heavily on the input vectors used to simulate the circuit, while the performance of levelized compiled code is independent of the input.

One measure that used to characterize the performance of event driven simulation is activity rate. For a single input vector, the activity rate is the number of gates simulated

divided by the number of gates in the circuit. This is extended in the natural way to collections of vectors, giving the following formula.

$$ActivityRate = \frac{GatesSimulated}{VectorCount * GatesInCircuit}$$

It is obvious from the algorithms presented in this chapter, that the amount of code executed per gate simulation is much larger for event-driven simulation than for levelized compiled code simulation. If the activity rate is close to 100%, one would expect levelized compiled code simulation to significantly outperform event-driven simulation. On the other hand, if the activity rate is zero, one would expect event-driven simulation to significantly outperform levelized compiled code simulation.

It is obvious that there must be some break-even point where levelized compiled code and event-driven simulation perform equally well. Experimentation has shown that the break-even point occurs around 2-3%. Although this seems small, there are situations where activity rates significantly lower than 2% are to be expected.

In a microprocessor, most of the components are idle most of the time and an activity rate lower than 2% is to be expected. On the other hand, in an array multiplier proper testing requires one to generate as much activity as possible and an activity rate significantly higher than 2% is to be expected. The “moral” of this discussion is that selection of a simulator depends on the task to be performed. There is no one technique which is inherently superior to another under all circumstances.

3.7 Threaded Code

The most common way to implement event-driven simulation is interpretively using the tables produced by the parser directly with no intermediate step. With the dynamic scheduling it is not clear that any other implementation is possible. However, using a concept called *threaded code*, it is possible to create a compiled event-driven simulator which is much more efficient than an interpreted simulator.

Threaded code is a method of organizing a computer program into segments that can be scheduled and executed independently of one another. Segments are scheduled using a queue of addresses, each of which points to an executable segment. Each segment is terminated by a couple of instructions that remove the next address from the queue and branch to it. Segments schedule one another by placing their addresses on the queue. Figure 3-16 shows the structure of a threaded code program.

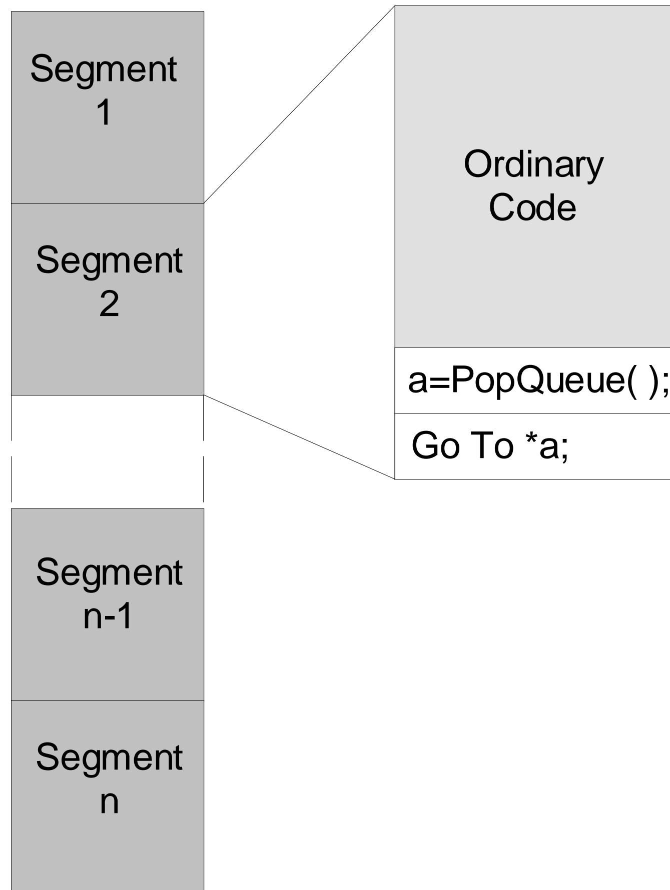


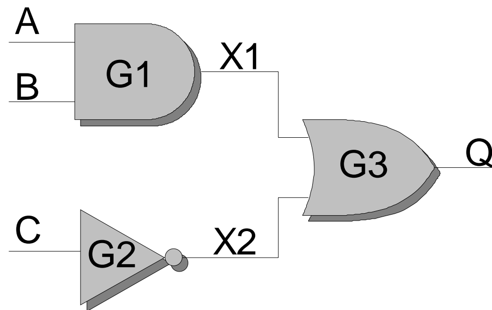
Figure 3-16. Threaded Code.

The first step in generating a threaded code simulation is to create a code segment for each gate and each net in the circuit. The gate-segments are used to schedule gate simulations, while the net-segments are used to schedule events. The segment for a gate contains specialized simulation code for the gate, while the segment for a net contains code for processing events that occur on the net. Figure 3-17 gives some sample segments for gates and nets, while Figure 3-18 contains a sample circuit and some of the code that is generated from it.

```
G1: NewX1 = A And B;
    If NewX1 <> X1 Then
        Queue(NetX1);
    EndIf
    Address = PopQueue( );
    Go To *Address;
G2: NewX2 = Not C;
    If NewX2 <> X2 Then
        Queue(NetX2);
    EndIf
    Address = PopQueue( );
      Go To *Address;
```

```
NetA:  A = NewA;
       Queue(G1);
       Address  =  PopQueue(
);
       GoTo *Address;
NetX1:     X1 = NewX1;
       Queue(G3);
       Address  =  PopQueue(
);
       GoTo *Address;
NetQ:  Q = NewQ;
       Address  =  PopQueue(
);
       GoTo *Address;
```

Figure 3-17. Sample Code Segments.




```
G1: NewX1 = A And B;
    If NewX1 <> X1 Then
        Queue(NetX1);
    EndIf
    Address = PopQueue( );
    Go To *Address;
G2: NewX2 = Not C;
    If NewX2 <> X2 Then
        Queue(NetX2);
    EndIf
    Address = PopQueue( );
    Go To *Address;
```

```
NetA:  A = NewA;
       Queue(G1);
       Address = PopQueue( );
       GoTo *Address;
NetX1:     X1 = NewX1;
       Queue(G3);
       Address = PopQueue( );
       GoTo *Address;
NetQ:  Q = NewQ;
       Address = PopQueue( );
       GoTo *Address;
```

Figure 3-18. A Sample Circuit with Generated Code.

Traditionally, threaded code uses a queue to guarantee that code segments are executed in the order in which they were scheduled. However, in event-driven simulation stacks can be used instead of queues as long as two separate stacks are used. One stack is used for event-processing routines, while the other is used for gate simulation routines. Using two stacks can improve simulation performance because stack management is

somewhat less complicated than queue management. Using stacks, the statements Queue(X), and X=PopQueue( ) translate into the following sequences of statements.

Queue(X)	***X=PopQueue( )***
TopOfStack++;	X = *TopOfStack;
*TopOfStack = &X;	TopOfStack--;

When a stack becomes empty, three things must happen. First, the stack must no longer be used as a source for dynamic jump addresses. Second, it is necessary to test for termination and for oscillations. Third, if simulation is to continue, it is necessary to switch from one stack to the other. Because it is impractical to build all of this functionality into each routine, each stack is initialized with a single address that points to a stack-termination routine. Figure 3-19 illustrates the termination routine for the event-processor stack.

```
EventTermination:
    *EventTopOfStack = &EventTermination;
    EventTopOfStack++;
    If *GateTopOfStack = &GateTermination Then
        /* Terminate simulation of current vector */
        Return;
    EndIf
    Address = *GateTopOfStack;
    GateTopOfStack--;
    Go To *Address;
```

Figure 3-19. A Stack Termination Routine.

The first action of the stack termination routine is to push its own address onto the stack, thereby maintaining its presence at the bottom of the stack. Different termination routines are required for each stack.

When an input vector is processed, it is tested for changes and the initial event processors are scheduled. The input-test routine then pops the first address off the event stack and branches to the corresponding event processing routine. If there are no events in the stack, this will cause immediate termination of the simulation and reading of the next input vector.

Experimental results have shown that threaded code event-driven simulators significantly outperform interpreted simulators. The most significant performance increases have been realized by assembly-language based simulators.

3.8 Eliminating Duplicate simulations.

When two events to occur simultaneously on different inputs of the same gate, the gate will be scheduled for simulation twice, unless special precautions are taken. The scheduling of duplicate simulations can also cause duplicate scheduling of events, which can, at least in theory, cause the number of events to explode. One method for preventing

duplicate gate simulations is to flag a gate when it is queued for simulation, and deflag it when the simulation actually occurs. If a gate is flagged, it will not be scheduled, regardless of the events that occur on its inputs.

Although flagging eliminates duplicate gate simulations, it adds additional tests during event processing and gate simulation. In practice, it is debatable whether there will be sufficient savings to offset the additional processing. In fact, some studies have shown that it is more efficient to simply allow the duplicate simulations to occur. In some cases it is necessary to avoid duplicate scheduling of gate simulations to prevent damage to the scheduling queues. For example if gates are scheduled using linked lists, and a single data structure is used for each gate, duplicate scheduling of a gate will destroy a portion of the gate queue. (Note that interpreted simulators often use such a scheme.) Figure 3-20 illustrates how this can happen. In this example, gate G2, which is already on the queue, is inserted at the end of the queue. This causes gates G3 and G4 to disappear from the end of the queue, since they were accessible only through the link pointer of G2.

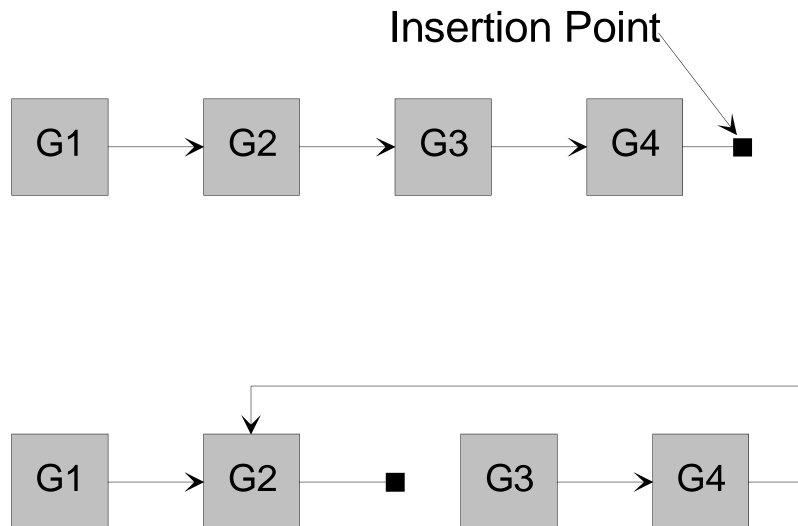


Figure 3-20. Rescheduling a Dedicated Gate Structure.

3.9 Two-List V.S. One-List Scheduling

The algorithms described so far all use two scheduling structures, one for events and one for gates. An alternative approach is to eliminate the gate queue or the event queue and use a single scheduling queue. Such an approach is called *Single-List* or *One-List scheduling* to contrast it with Two-List scheduling.

In most cases, it is the gate queue which is eliminated. Gate simulation is performed when an event is processed, and any new events are scheduled immediately. To guarantee that events are processed in the correct order, the event queue must be a true first-in first-out queue, and some mechanism for keeping track of simulated time must be used. In two-list simulation, all queued events are for the same unit of simulated time. In one-list simulation, there are normally events queued for two consecutive units of time, and it is possible for two consecutive events to be queued for a single net. Because there is no switching back and forth between gate simulation and event processing, there is no clear

boundary between time units. One method for keeping track of time units is to use a marker event as shown in Figure 3-21. When the marker event is processed, an intermediate output vector is created, and a new marker event is queued at the end of the event queue. An alternative method is to use a time stamp in each event. Events created during the processing of an input vector are stamped with time zero. When a new event is created during event processing, the time stamp of the new event will be one larger than the time stamp of the event being processed. Intermediate output vectors are created when the time stamp of the event currently being processed is larger than the time stamp of the preceding event.

A	B	C	M	D	C	E
1	0	1		0	0	1

Figure 3-21. A Single-List Event Queue.

Management of net values is tricky because gates may be evaluated before all input values are available. For example, consider the gate, and the event queue pictured in Figure 3-22. Two simultaneous events are scheduled for the inputs of a single gate. The effect of these events will be to leave the output unchanged, and because the events are simultaneous, no hazard exists. Single-List scheduling will simulate the gate twice. The first simulation will produce an output of zero the second will produce an output of one. Since each simulation represents a change, one would expect that an event would be scheduled after each simulation. However, if this is done, it is important to protect against incorrect triggering of edge-triggered flip-flops. The safest approach is to schedule an event for net C after the first simulation, and remove the event after the second simulation. This is known as *Event Cancellation*.

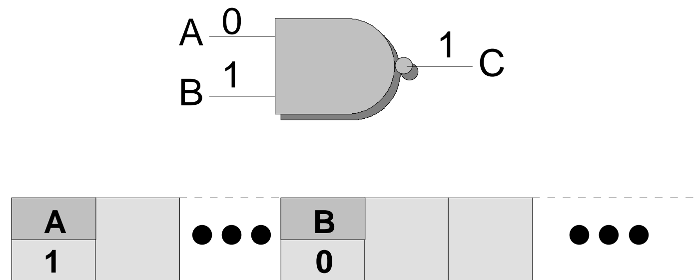


Figure 3-22. Single-List Scheduling Problems.

Even with Event Cancellation it is necessary to exercise care when processing events. Correct simulation demands that the value of C not change until one time unit later than the changes in A and B. Since the changes in A and B are simultaneous, this implies that the value of C must not change during the processing of the event for A. When the event for B is processed, the result of the simulation must be compared against the value contained in the event for C, not against the value of C. If the result of the simulation is compared against the value of C, no event will be created, resulting in an incorrect output value. The comparison is complicated by the fact that there may be a second event queued for C, which is simultaneous with the events for A and B. This event must not be canceled when the event for A is processed, but its value must be used for comparison

purposes since it represents the value that C will have at the end of the current simulated time unit. Single-List scheduling with an event queue is given in Algorithm 3-5. The event structure is assumed to be the same as that given in Figure 3-3.

```
Event_Processor:
{
    For each Event E in Event_Queue do
        If E is a marker event Then
            Output intermediate vector
            Queue new marker event
        Else
            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
                Simulate G
                For each output K of G do
                    If an event is queued for K for Next Time Period Then
                        If New Value of K ≠ Event Value Then
                            Delete Event
                        EndIf
                    Else If an event is queued for K at for Current Time Then
                        If New Value of K ≠ Event Value Then
                            Queue Event with New Value of K
                        EndIf
                    Else
                        If New Value of K ≠ Old Value of K Then
                            Queue Event with New Value of K
                        EndIf
                    EndIf
                EndFor
            EndFor
        EndIf
        Remove E from Event Queue
    EndFor
}
```

Algorithm 3-5. Single-List Event Processing.

A second method of single-list scheduling is to eliminate the event queue and use only a gate queue. In this technique, a gate is simulated, and the output is tested for changes. instead of queuing an event, all gates in the fanout of the gate are immediately placed in the gate queue. The problems that this technique must address, are preventing nets from changing value until the end of a simulated time period, and keeping track of simulated time periods. Status flags are usually used to prevent duplicate gate scheduling, but it is possible for a gate to appear in the queue twice as long as it is queued for two different time periods. One way to solve these problems is to use a queue marker similar to that used in Algorithm 3-5. The results of each gate simulation are saved in temporary

storage until the queue marker is processed. During the processing of the queue marker, all net values are copied from temporary storage to permanent storage. Queue marker processing also creates intermediate output vectors and places a new queue marker at the end of the gate queue. The queue markers separate the queue into sections, each of which contains gates for one and only one simulated time period. Algorithm 3-6 gives the gate simulation algorithm for single-list gate-oriented scheduling.

```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        If G is a Queue Marker Then
            Copy Temporary net values to Permanent Storage
            Create output vector
            Queue new Queue Marker
        Else
            N := The Output of G
            Simulate G, put result in New_N
            If New_N is different from The current value of N Then
                For each gate H in the fanout of N Do
                    If H is not queued for next time slot Then
                        Add H to queue
                    EndIf
                EndFor
            EndIf
        EndIf
        Remove G from Gate_Queue
    End For
}
```

Algorithm 3-6. Gate Processing.

The process of copying net values from temporary to permanent storage is similar to event processing in two-list scheduling. Because of this there is little difference between this type of single-list scheduling and two-list scheduling. However, if it were possible to ignore the requirement that nets not change value until the next simulated time period, gate-queue single-list scheduling would be preferred because of its simplicity.

3.10 Zero-Delay Event-Driven Simulation

The unit-delay timing model exhibited by the two-list and one-list algorithms is a consequence of the structure of the queues used to schedule events. By using a more complex queue structure it is possible to create event-driven simulators for other timing models. One case that is particularly interesting is the zero-delay model, which is normally associated with levelized simulation. Although the unit-delay model is considered to be more realistic than the zero-delay model, the zero-delay model has some advantages. In particular, in the zero delay model no gate of a combinational circuit will be simulated more than once per input vector. In the unit-delay model, multiple

simulation of a single gate is a common occurrence. Elimination of duplicate gate simulations can significantly speed up simulation. At the same time, one must sacrifice the detection of hazards, because for a hazard to be detected, a gate must be simulated at least twice.

The first step in zero-delay event-driven simulation is to levelize the circuit and assign each gate a level number. Instead of one or two scheduling queues, *n* queues are used, where *n* is the number of levels in the circuit. Each gate is permanently associated with the queue corresponding to its level number. No event queue is used. The scheduling queues are used to store the gates that have been scheduled for simulation.

As with other event-driven algorithms, the first step is to examine the latest input vector for changes and queue any gate that is attached to a primary input that has changed value. Any time a gate is scheduled, it will be placed in the queue corresponding to its level number.

Queues are processed in ascending order by level number. As queued gates are encountered, they are removed from the queue and simulated. The outputs of the gate are then examined for changes. If a change occurs, all gates in the fanout of the net are queued for simulation. As before, each of these gates will be queued in the queue that corresponds to its level number. The simulator keeps track of the level number corresponding to the current gate queue. If any gate is inserted into a queue with a lower level number than the current queue, then an iteration flag is set, which will cause a second pass to be made through the scheduling queues. The only thing that can cause the iteration flag to be set is a change in a feedback arc in an asynchronous sequential circuit. Depending on the method used to break cycles and force-levelize the circuit, the simulation of asynchronous circuits may not be as accurate as the corresponding unit-delay simulation.

Unit-Delay Event-Driven simulation is illustrated in Figure 3-23. This algorithm can be implemented as an interpretive algorithm, or as a compiled algorithm using threaded code. When a threaded code implementation is used, each of the queues will have a queue termination element appended to it. The termination element advances the simulation to the next queue.

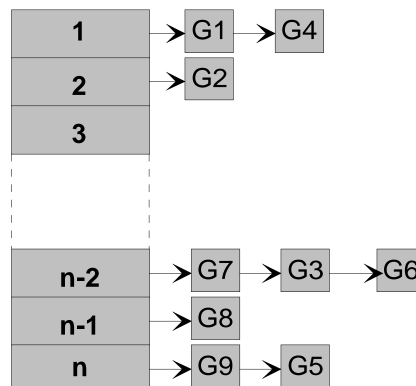


Figure 3-23. Unit-Delay Event-Driven Simulation.

Zero-delay event-driven simulation is considerably more efficient than unit-delay event-driven simulation. The break-even point occurs at an activity rate of around 20% rather than the 2-3% level exhibited by unit-delay simulation. Certain speed-up

techniques, such as simulating fanout-free networks as single gates, have boosted the break-even point to as high as 40% for some circuits.

3.11 Cache Efficiency

The code generated by the threaded code technique is not particularly cache-friendly. Because each net and each gate has its own subroutine, locality of reference during the simulation of a single input vector is minimal. In most cases, there will be too much code for the entire simulator to fit in the cache. Combined with the poor locality of reference, this virtually eliminates any beneficial effect of the cache.

Let us review why such a large amount of code is necessary. Figure 3-24 shows two gates and the code that is generated for each.

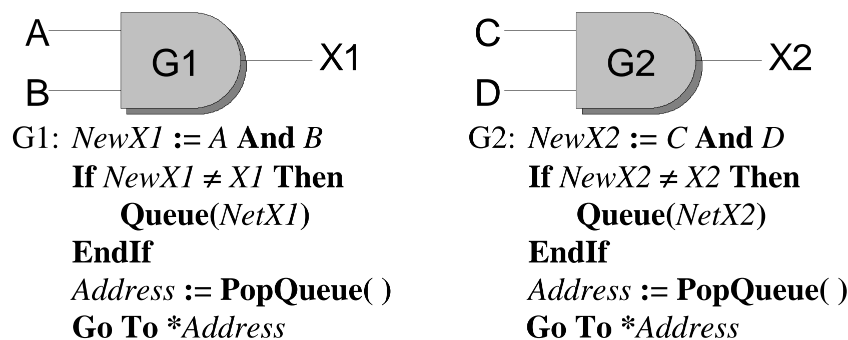


Figure 3-24. Gate Simulation Routines.

The two generated routines shown in Figure 3-24 are identical except for the variable names. These routines are generated in this way to eliminate the decoding of gate types, and to eliminate the input-net processing loop. One way to reduce the amount of generated code would be convert these two routines into a common subroutine, such as that illustrated in Figure 3-25.

```
TwoInputAnd(Op1,Op2,Out,NetID)
{
   NewVal := Op1 And Op2
   If NewVal ≠ Out Then
       Queue(NetID,NewVal)
   EndIf
   Address := PopQueue( )
   Go To *Address
}
```

Figure 3-25. A Gate-Simulation Subroutine.

The *Shadow Algorithm* is a mechanism for replacing the individual routines of Figure 3-24 with generic routines like those of Figure 3-25.

3.12 The Shadow Algorithm

To combine the use of generic routines with the concept of threaded code, it is necessary to have a parameter-passing mechanism that is different from the usual stack-based mechanisms. The basis of this mechanism is a data structure called a *shadow*, which is used to provide a portion of the environment for the gate simulation routines. In a typical program, the stack is used to provide the parameters and local variables to a subroutine. The Stack Pointer provides access to these variables and parameters. The shadow technique uses a second register, called the Shadow Pointer, to provide additional parameters and local variables to the threaded code segments of the program. There is one standard environment for all code segments but each invocation of a code segment will use a different shadow.

A shadow is a statically allocated structure containing the parameters and variables necessary to simulate a single gate or to process a single event. To avoid the overhead of type decoding, each shadow contains a pointer to the a processing routine, either a gate simulation routine or an event processor. Unlike the threaded code techniques which schedule code segments, the shadow technique schedules shadows. A shadow is processed by removing it from the queue and branching to its processing routine pointer. The final step in each processing routine is to dequeue the next shadow, load its pointer into the shadow pointer register, and branch to the new processing routine. Figure 3-26 gives the structure of a gate shadow, and Figure 3-27 shows a shadow-based gate processing routine.

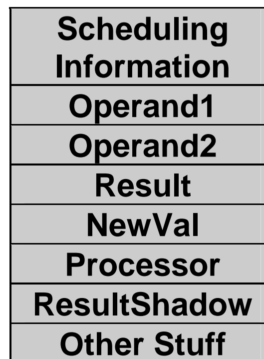


Figure 3-26. The Structure of a Gate Shadow.

```
TwoInputAnd:
    *NewVal := *Operand1 And *Operand2
    If *NewVal ≠ *Result Then
        Queue(ResultShadow)
    EndIf
    ShadowPointer := PopQueue( )
    Go To *Processor
```

Figure 3-27. Gate Processor for the Shadow Technique.

In Figure 3-27, shadow variables are indicated using an asterisk indicating that the variable must be dereferenced through the shadow pointer. Each gate and each net must

have its own shadow. The shadows are created at compile time, and for the most part, will not be altered during run time. In most cases, net values must be accessible from several different shadows, so the shadow must contain a pointer to the value rather than the value itself. Net shadows, which are illustrated in Figure 3-28, are used to schedule the processing of events. An example of an event processing routine is given in Figure 3-29. There must be one event processing routine for each different fanout found in the circuit.

Scheduling Information
TempVal
RealVal
Fanout1
Fanout2
Processor
Other Stuff

Figure 3-28. The Structure of a Net Shadow.

```
FanoutOfTwo:
    *RealVal := *TempVal
    Queue(Fanout1)
    Queue(Fanout2)
    ShadowPointer := PopQueue( )
    Go To *Processor
```

Figure 3-29. Net Processor for the Shadow Technique.

3.13 The Interpretive Shadow Algorithm.

Because neither the gate-simulation routines nor the event processors depend on the structure of the circuit, these routines could be pre-compiled and loaded from a library rather than being generated and compiled. Since the number of different routines, even in the worst case, is quite small, it is possible to use a standard simulation engine for all circuits, and generate only the data structures. This procedure eliminates the need for compilation without sacrificing performance.

3.14 Conclusion

Event-driven simulation is a versatile simulation technique that can be used to accurately simulate virtually any kind of digital circuit, including asynchronous sequential circuits. It is more efficient than levelized simulation when the activity rate is sufficiently low, but significantly less efficient when activity rates are higher. The use of different queuing structures can lead to different timing models. This chapter shows how to implement the unit-delay and zero-delay models. More complex timing models will be

discussed in later chapters. Many commercial simulators are based on the algorithms presented in this chapter.

3.15 Exercises

1.