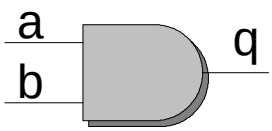
Chapter 3

Event Driven Simulation

3.1 Motivation

The preceding chapter presented the concept of levelized simulation, in which gates are sorted into levelized order and each is simulated once per input vector. Regardless of whether the interpreted or the compiled technique is used, when the simulation of a vector terminates, the value of each net is stored in a permanent location. For interpreted simulation these values are contained in the net table, while for compiled simulation the values are stored in global variables. Furthermore, if the inputs of a gate do not change, then the outputs will not change. (Edge triggering must be assumed for some gates.) Now suppose a circuit is simulated for two consecutive identical input vectors. There is no need to simulate the second vector, because nothing will change after the first vector.

The same idea can be extended to individual gates. If the inputs of a gate do not change there is no need to simulate it. This suggests a method for improving the speed of simulation, by reducing the number of gates simulated per input vector. Unfortunately, a naive approach will not result in any performance improvement. 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, it is not possible to test the inputs of every gate for changes. Figure 3-1 illustrates the difficulty. The enhanced code executes at least six instructions, and may execute as many as nine instructions, whereas the unenhanced code executes only three instructions.



Unenhanced		Enhanced	
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.

If the inputs of a gate do not change, then the output of the gate will not change. In the example of Figure 3-1, if neither a nor b change, then q will not change. If q is an input to another gate, it is not necessary to test the input q as long as neither a nor b has changed. To further illustrate this concept, consider the circuit of Figure 3-2.

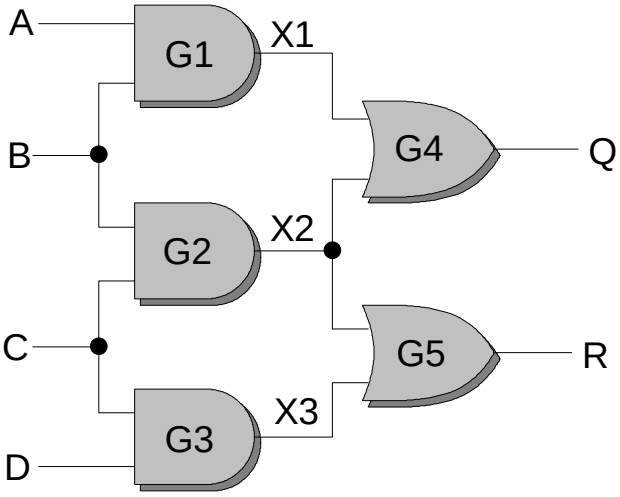


Figure 3-2. Change Propagation.

Suppose the circuit of Figure 3-2 is simulated with two consecutive input vectors, (0,0,0,0) and (0,0,0,1). Since A, B, and C have not changed, it is not necessary to simulate gates G1 and G2. Since neither G1 nor G2 have been simulated, it is not necessary to test X1 or X2 for changes. The simulation of G4 can be bypassed without testing X1 or X2. Unfortunately, the code for doing such a thing would be extraordinarily

complex, particularly considering the fanout of B, C, and X1. The problem lies with the static gate scheduling which is implicit in the code of Figure 3-1. Eliminating unnecessary gate simulations requires a fundamentally different approach to scheduling gate simulations.

3.2 Basic Principles

Event-Driven Simulation is designed to eliminate unnecessary gate simulations without introducing an unacceptable amount of additional testing. The concept of an event is central to this simulation technique. An event is defined to be a change in the value of a net. Internally, an event is represented as a data structure similiar 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.

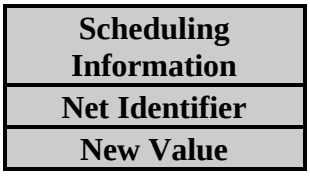


Figure 3-3. The Event Structure.

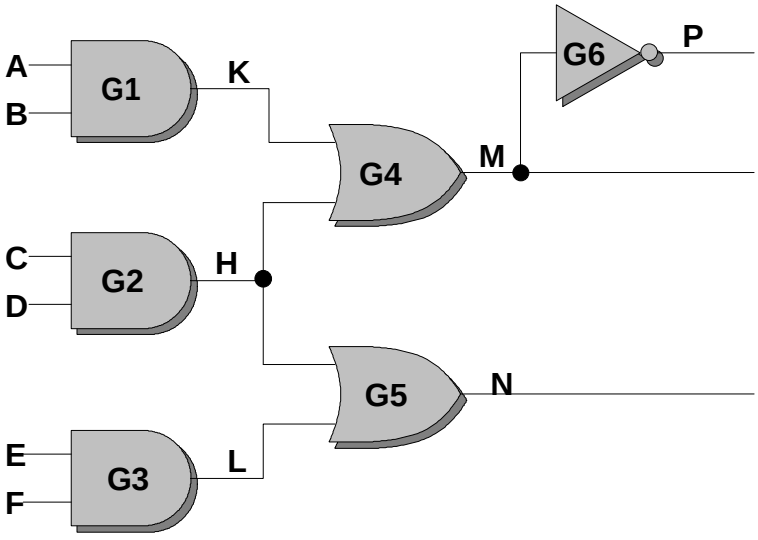
The main function of an Event Driven simulator is to detect events, and schedule gate simulations in response to them. If no events occur, implying that there are no net changes, then no gates will be simulated. Furthermore nets are tested for changes only when necessary. Unlike levelized simulation, event driven simulation is a dynamic scheduling technique which may schedule gate simulations in an order which cannot be predicted as the circuit is being parsed. Because of this, one or more dynamic scheduling queues will be used.

There are two times when it is necessary to test nets for changes, when a new input vector is read, and immediately after a gate is simulated. When an event is detected, an event structure must be created and stored for future processing. The Event Queue will be used to store any events that have been detected and not yet processed. Algorithm 3-1 illustrates the algorithm for processing a new input vector, which is the starting point for the simulation process.

```
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, events must be created for nets A, B, and F. These events are inserted into the event queue as illustrated in Figure 3-4.



Event Queue

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 Algorithm 3-1 inserts all new net values into the event structure not into permanent storage. The first part of event processing is to copy net values from the

event structures into permanent storage. The second part is to schedule all gates in the fanout of a net for simulation. This is done by placing the identification of the gate into another queue called the Gate Queue. The details of event processing are given in Algorithm 3-2. This algorithm assumes that the net table is being used for permanent storage of net values.

```
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 illustrates, event processing continues until all events have been processed and removed from the queue. At this point, there will usually be several gates in the gate queue. Figure 3-5 illustrates the gate queue after processing the events illustrated in Figure 3-4.

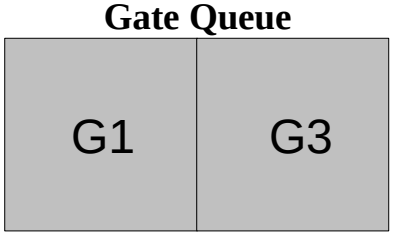


Figure 3-5. A Sample Gate Queue.

Once all events have been processed, it is necessary to simulate each gate in the gate queue. In addition to simulating each gate, it is also necessary to test the output of each simulated gate for changes and schedule events. The details of gate simulation process are illustrated in Algorithm 3-3. This algorithm assumes that a gate simulation subroutine is available for simulating an individual gate.

```
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.

As Algorithm 3-3 illustrates, gate simulation creates new events and places them in the Event Queue. The Event Processor must then be called to process these events. The gate processor and the event processor must be called repeatedly until there are no more events to process. Simulation of an input vector consists of a single call to the Input Processing Routine, and several calls to the gate and event processors. 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 the levelized simulation technique which provides a zero-delay model, the basic event-driven technique provides a unit-delay model, in which the delay of each gate is assumed to be one. To illustrate consider the circuit of Figure 3-6. Assume that the nets have the values shown in Figure 3-6, and that it is now being simulated with the new input A=1, B=1. The inputs of G2 will change from (0,1) to (1,0), so in levelized simulation the

output Q will remain at zero. However, this will not be true for event driven simulation, as illustrated in xxx.

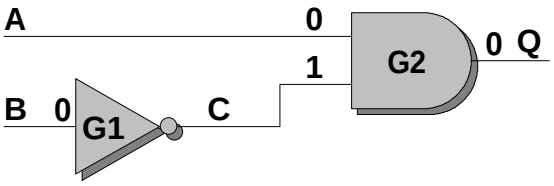


Figure 3-6. A Sample Circuit.

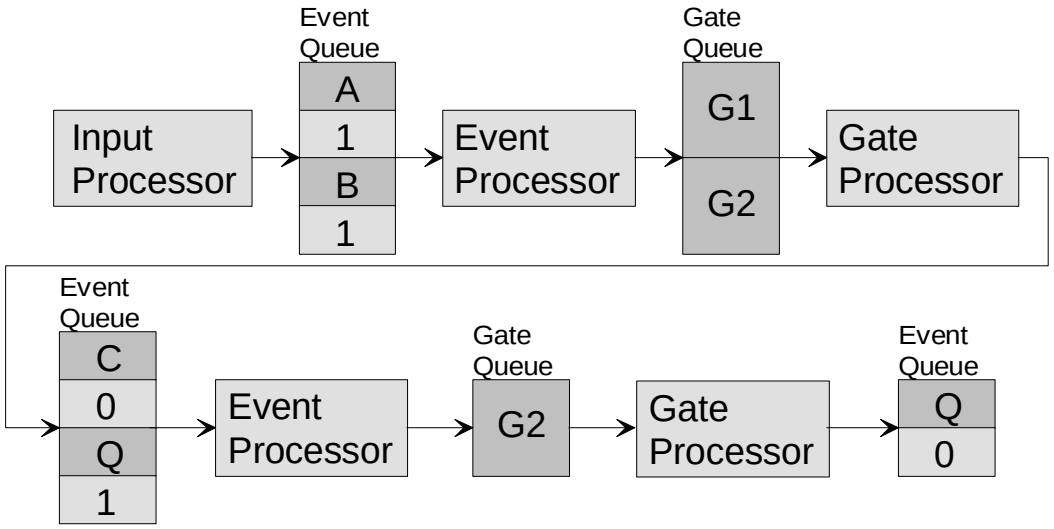


Figure 3-7. An Event-Driven Simulation.

Figure 3-7 illustrates several important points about event driven simulation. First, note that G2 is simulated immediately without waiting for the simulation of G2, as would be the case in levelized simulation. Also note that G2 is simulated twice, instead of once as in levelized simulation. Most importantly, there are two events for the net Q, one which changes the value from zero to one and the second which changes Q from one to zero. Thus the value of Q changes briefly from zero to one and back to zero. If Q were the input of a T flip-flop, the results of the simulation would be different for levelized simulation and event driven simulation.

The two-phase structure of the event-driven simulation algorithm imposes a time scale on the simulation. Each execution of the Event Processor subroutine defines one unit of simulated time. Intermediate outputs can be output 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 theoretically 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 three gates, as illustrated in Figure 3-8, and that net values are updated immediately after each gate simulation. Gate G2 will be simulated after G1 and before G3. If the simlation 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.

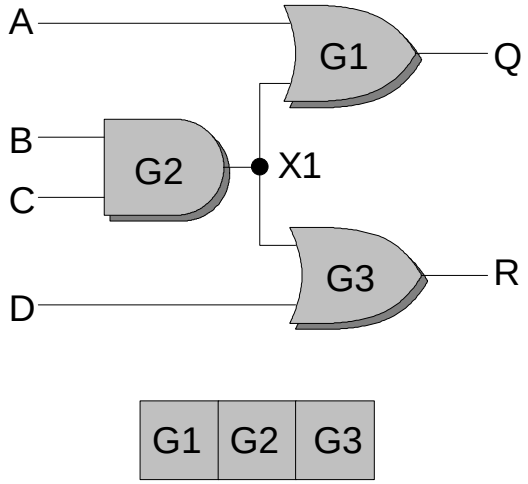


Figure 3-8. Net Update Conflicts.

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 which are illustrated in Figure 3-9 and dynamic hazards which are illustrated 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 stablizes 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.

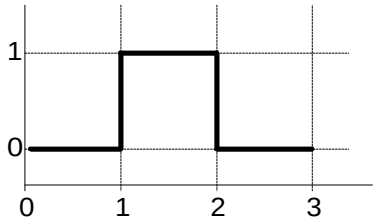


Figure 3-9. A Static Hazard.

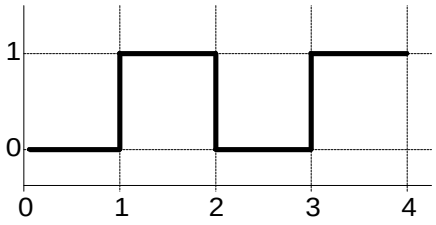


Figure 3-10. A Dynamic Hazard.

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.

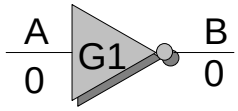


Figure 3-11. An Inconsistently Initialized Circuit.

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.

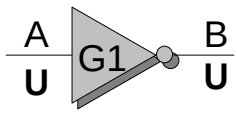


Figure 3-12. A Circuit initialized with Unknowns.

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.

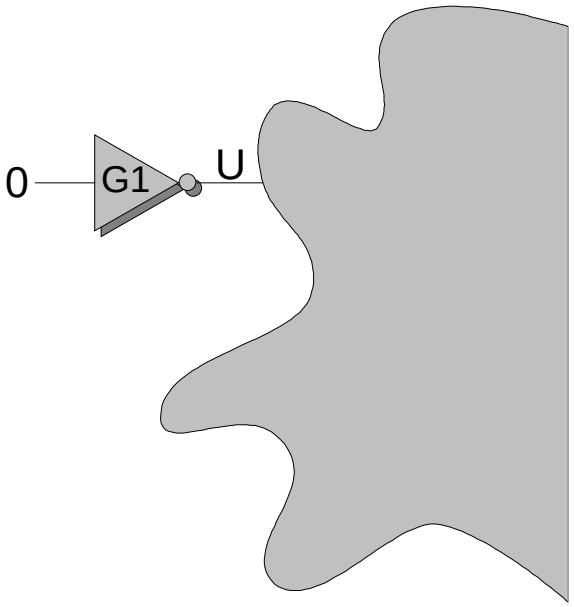


Figure 3-13. Constant Zero Signals.

In xxx 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 queueing an event for each constant-zero and constant-one signal for the simulation of the first input vector. The simplest approach is to pre-queue these events before the processing of the first input vector.

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.

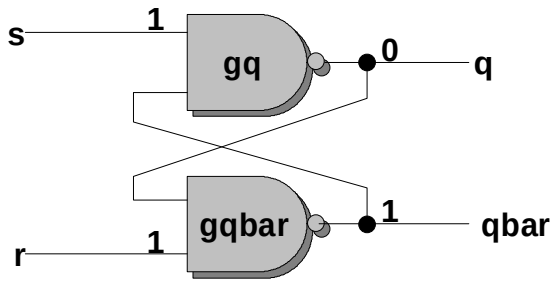


Figure 3-14. An Asynchronous Circuit.

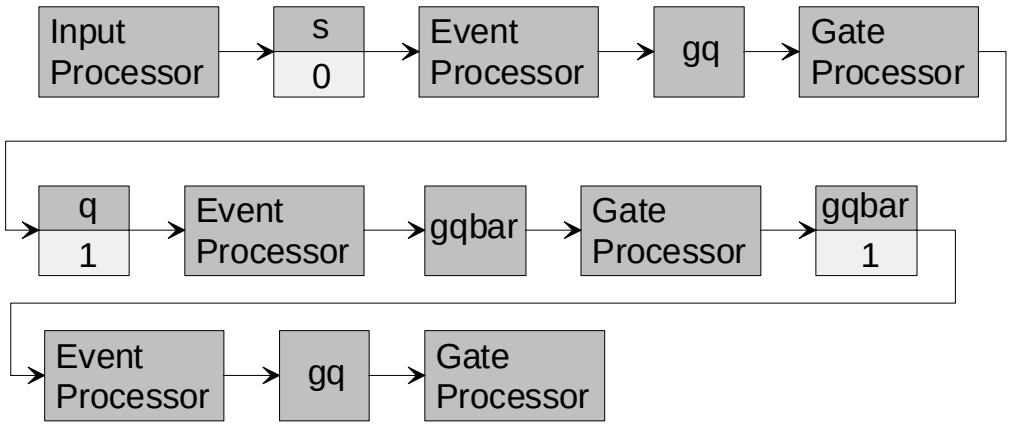


Figure 3-15. Simulating an Asynchronous Circuit.

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.

This procedure for setting nets to unknown values may fail if the cause of the excessive simulations is a synchronization failure in a supposedly synchronous circuit. In such a case, the circuit can cycle through a complex set of state transisions that involve a major portion of the circuit. In most cases the safest procedure is to detect the oscillation immediately before calling the Event Processor, and setting the values of all nets in the event queue to the unknown value.

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 simulation. 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 may occur in an asynchronous sequential circuit. The

primary reason for using event-driven simulation in preference to levelized compiled code 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 of a circuit is the number of gates simulated divided by the number of gates in the circuit. For a collection of input vectors, the activity rate is the total number of gates simulated for all input vectors, divided by the product of the number of gates in the circuit and the number of input vectors. The following is the mathematical expression for the activity rate.

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

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 at an activity rate of around 2-3%. Although this seems small, there are situations where activity rates significantly lower than 2% are to be expected. Suppose the simulator is being used to model the behavior of a microprocessor. In a microprocessor, most of the components are idle most of the time. In such a simulation, an activity rate lower than 2% is to be expected.

On the other hand, suppose the circuit under simulation is an array multiplier. Proper testing of such a circuit requires one to generate as much activity as possible in the circuit. For such a circuit, 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

Up to this point it has been assumed that event-driven simulation will be implemented interpretively. The tables produced by the parser will be used directly for simulation with no intermediate step. Because of the dynamic scheduling implied by event-driven simulation, 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 simulation for a circuit. As in levelized simulation, the motivation for creating a compiled simulator is to increase simulation efficiency. 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 sequence of instructions that pops the next address off the queue and branches to it. A segment is scheduled by placing its address on the queue. Segments are permitted to schedule one another. Figure 3-16 illustrates the structure of a threaded code program.

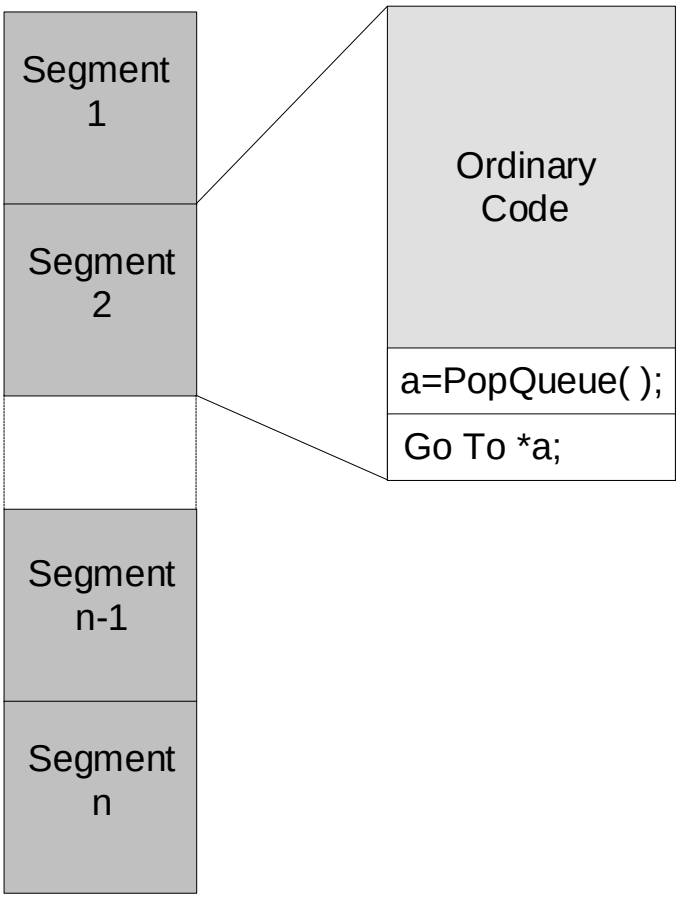


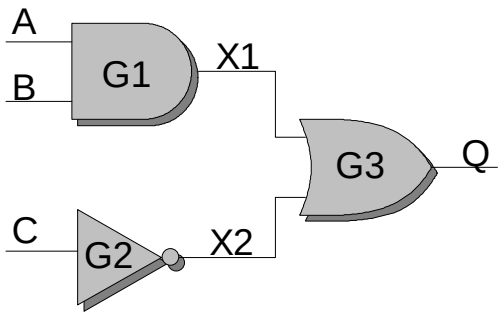
Figure 3-16. Threaded Code.

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. To illustrate,

```
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 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. A Sample Circuit with Generated Code.

The traditional idea of threaded code uses a queue of routine addresses 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 queue is used to queue event-processing routines, while the other is used to queue the gate simulation routines. This 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--;

Without some additional code, there will be problems with the code of Figure 3-17 when either stack becomes empty. 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 another. It is impractical to build all of this functionality into each routine. To solve the problem, each stack is initialized with a single address, which points to a stack-termination routine. Figure 3-18 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 3-18. A Stack Termination Routine.

Note that the first action of the stack termination routine is to push its own address onto the stack. Two different termination routines are required, one for each queue. Functionally, the event processors, gate simulators, and event processors act as a single simulation subroutine. The first step in this routine is to test the current input vector for changes, and schedule event processors for each changed input. The final action of the input test routine is to pop the first address off the event-processor stack and branch to the corresponding routine.

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

3.8 Eliminating Duplicate simulations.

As discussed above, event-driven simulation is most effective for simulations where the activity rate is low. Nevertheless, it is possible for two events to occur simultaneously on different inputs of the same gate. When this occurs, 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 rescheduled, 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 dammage 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.) xxx 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.

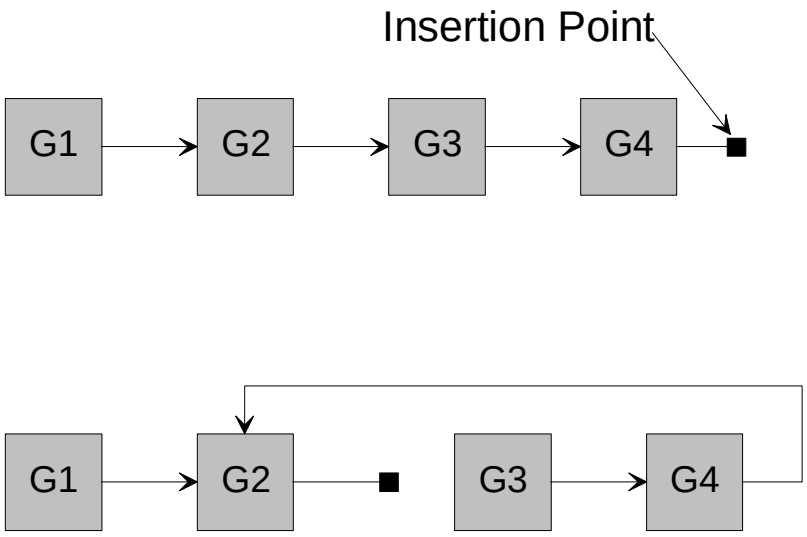


Figure 3-19. Rescheduling a Dedicated Gate Structure.

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

The event-driven algorithms described so far all use two scheduling queues, 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 the more conventional 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 two-list

simulation, there may be queued events for two distinct units of time. It is possible for two consecutive events to be queued for a single net. One method for keeping track of simulated time is to use a marker event as illustrated in Figure 3-20. 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 3-20. 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 3-21. Two simulaneous 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.

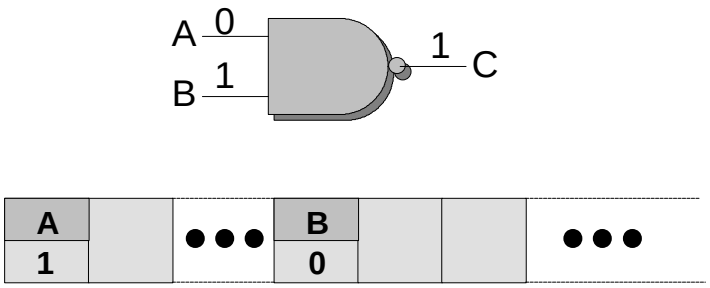


Figure 3-21. Single-List Scheduling Problems.

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 after 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 illustrated in Algorithm 3-5. The event structure is assumed to be the same as that illustrated 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 queueing 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 separates 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. The algorithm described in the next section is a gate-queue single-list algorithm that does not require temporary storage for net values.

3.10 Zero-Delay Event-Driven Simulation

Although the most straightforward implementations of event-driven simulation support the unit-delay timing model, it is possible to use virtually any timing model in conjunction with event-driven scheduling. One case that is particularly interesting is the zero-delay

model, which is normally associated with levelized simulation, especially levelized compiled code 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 simulations 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.

As with the algorithms of Chapter 2, the first step in zero-delay event-driven simulation is to levelize the circuit. This is done when the circuit description is parsed. Each gate is assigned a level number, which will not change during the progress of the simulation. 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 queue gates for simulation.

The first step in the simulation is to examine the latest input vector and compare the new values to the existing values of the primary inputs. Any gate attached to a primary input that has changed will be queue for simulation. The gate will be queued 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 a changed 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 that was force-levelized.

Unit-Delay Event-Driven simulation is illustrated in Figure 3-22. 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 illustrated in Figure 3-22 will have a queue termination element appended to the end of each queue.

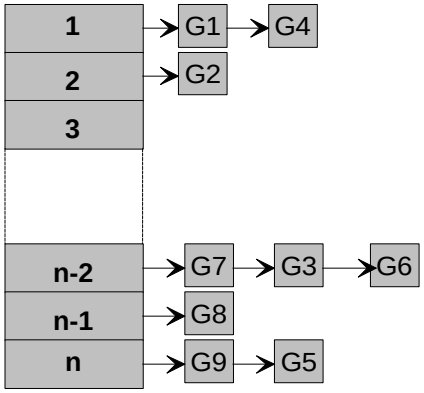


Figure 3-22. Unit-Delay Event-Driven Simulation.

3.11 Summary

3.12 Exercises

1. Make code for the circuit in xxx. Write the code in such a way that no gate is simulated unless one or more of its inputs changes value.
2.