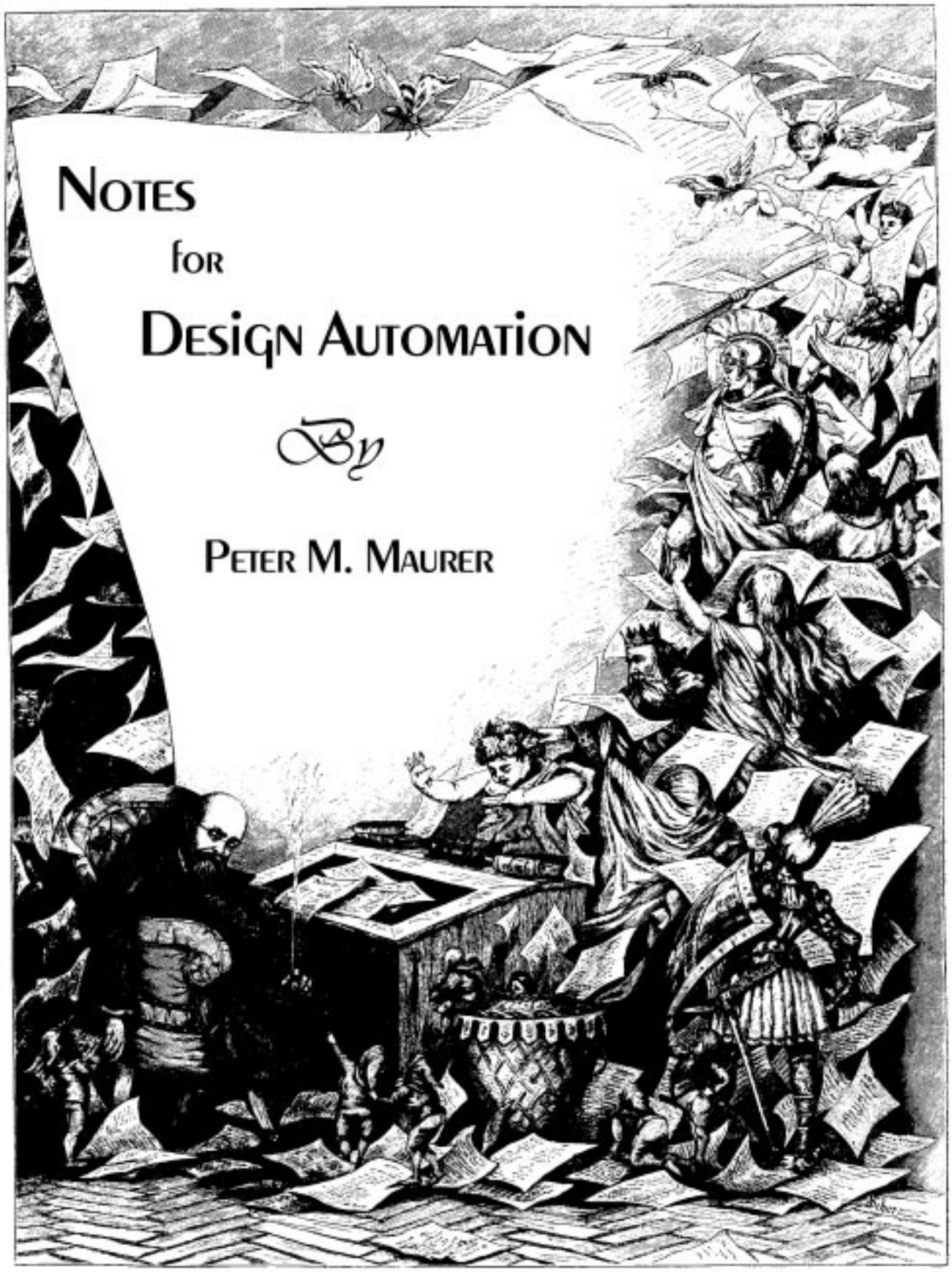
Notes for Design Automation

By

Peter M. Maurer



Chapter 1
Review of logic design.

1. Logic Gates.

Logic gates are the fundamental building blocks of most digital designs. A logic gate is a simple digital circuit that performs an elementary Boolean function. Because its electrical properties are well-known and predictable, its underlying structure can be ignored. In most cases, a gate can be treated as if it were a pure function, although there are times when one must focus on the electrical properties of the underlying circuit. Although a typical digital design methodology may use hundreds of gates, we will confine the discussion to four types of gates, the **And** gate, the **Or** gate, the **Not** gate, and the **Xor** gate. The **Xor** gate is also known as **Exclusive Or**. Many of the more complex types of gates can be constructed by combining several of these simpler gates. The four gates in question can be represented using the following schematic symbols.

And **Or** **Not** **Xor**

The functions performed by these gates are given by the following tables.

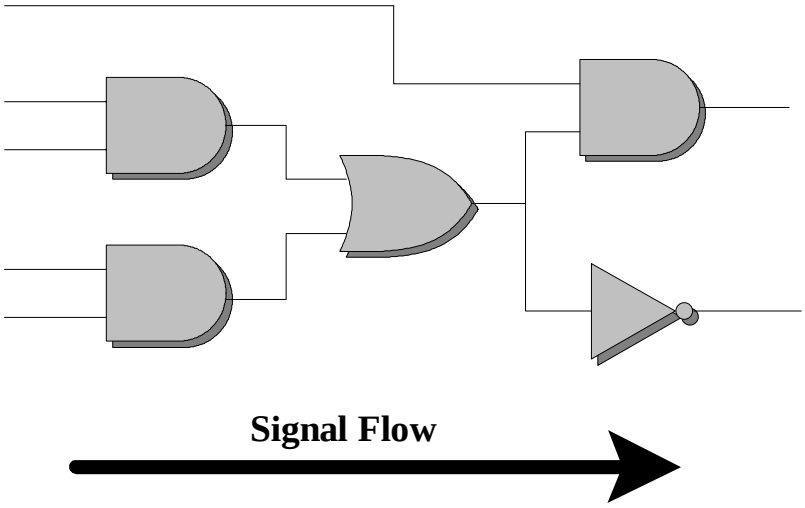
And	0	1
0	**0**	**0**
1	**0**	**1**

Or	0	1
0	**0**	**1**
1	**1**	**1**

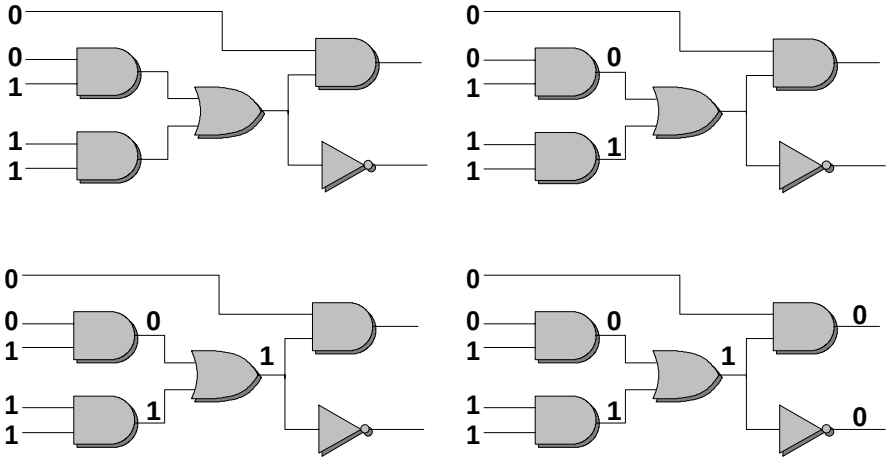
Xor	0	1
0	**0**	**1**
1	**1**	**0**

Not	
0	**1**
1	**0**

More complex functions can be constructed by connecting several gates together. A complete digital design may contain thousands or even millions of gates. Connections between gates are represented by drawing branching lines between them, as illustrated in the following picture. Signals flow in the direction indicated.



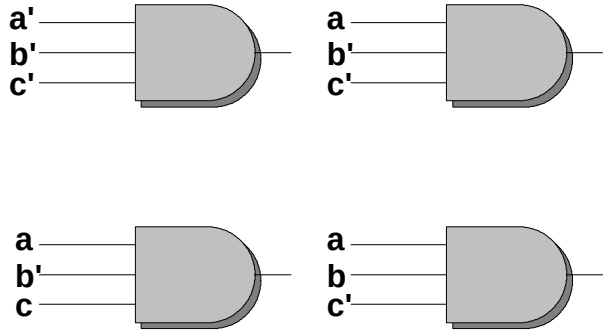
To simulate a circuit, such as that pictured above, one starts by assigning binary values to the primary inputs, and proceeds by propagating those values through the gates to the primary outputs. The following set of pictures illustrates how this is done.



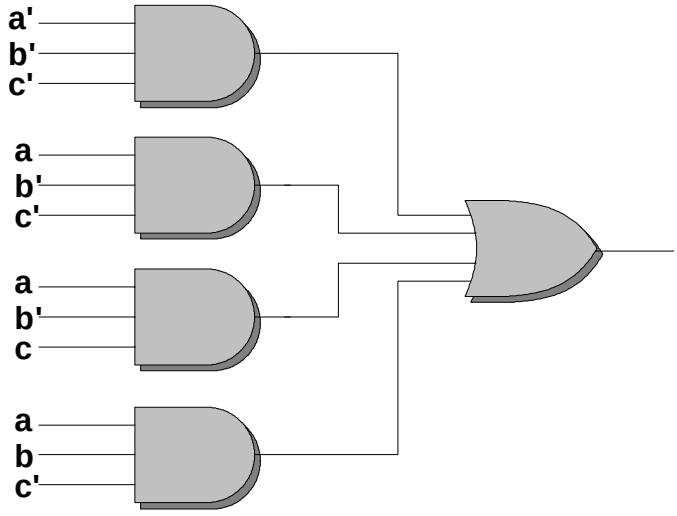
The *level* of a net is the number of gates that a signal must passes through to reach a the net. Only the longest path is considered when computing the level. It is well known that any binary function can be implemented with two levels of logic. (Three if one counts the **not** gates required to invert input signals.) Any binary function must be **one** for some input combinations, and **zero** for all others. To create a two-level circuit for a function, one can begin by listing the input combinations, and the required value for each, as has been done in the following table.

Sample Function			
Input	***Value***	***Input***	***Value***
0,0,0	1	1,0,0	1
0,0,1	0	1,0,1	1
0,1,0	0	1,1,0	1
0,1,1	0	0,1,1	0

Next, one makes a list of those input combinations for which the function must be one. In this case, the input combinations are 0,0,0; 1,0,0; 1,0,1; and 1,1,0. Assume that the input combinations represent the values of three input variables, **a**, **b**, and **c**, and assume that the complements of these variables, **a'**, **b'**, and **c'**, are also available. Using a three input **and** gate, it is possible to create a circuit whose output is equal to **one** for a particular input combination, and **zero** otherwise. The following diagram gives the **and** gates corresponding to the four listed input combinations.



Since each of these and gates is one for the desired input combination, and zero for everything else, it is possible to compute the desired function by feeding the outputs of these gates into an **or** gate, as illustrated below.



The Boolean formula that corresponds to this circuit is a'b'c'+ab'c'+ab'c+abc'. Although this method of implementing Boolean functions works, it usually creates circuits that are larger than necessary. Various techniques for minimizing Boolean functions have been studied, the most common of which is the Karnaugh map. The following Karnaugh map represents the preceding function.

	00	**01**	**11**	**10**
0	1			
1	1	1		1

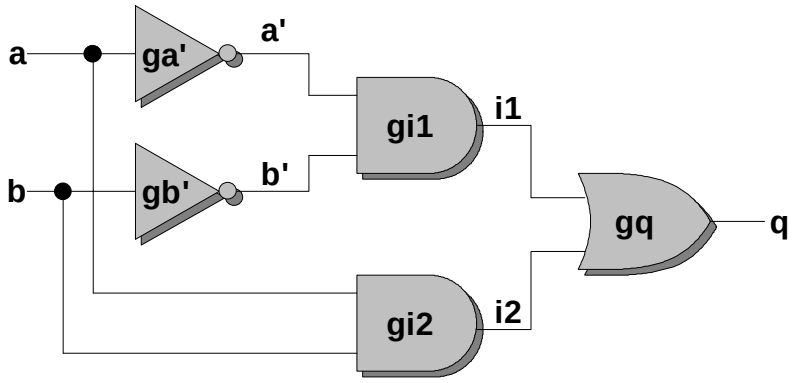
Based on this map, it is possible to reduce the function to the following formula: ab'+ac'+b'c'. Karnaugh maps do not work well for functions with more than four input variables. Although there are techniques for creating Karnaugh maps for five- and six-input functions, more complex techniques are generally used for these types of functions. One such technique is the Quine-McClusky technique, which is especially suited for implementation on a computer. Although the Quine-McClusky technique can be used manually, this is seldom done.

Before a circuit can be simulated, it must be converted to machine-readable form. Many companies have created tools that permit a circuit designer to draw a logic diagram on the computer screen, and feed that logic diagram to a simulation program. These tools are known as *schematic capture* tools. Although schematic capture tools are popular, the process of creating a schematic is slow. For complex circuits, the schematic may difficult to create correctly and may also be difficult to read. A number of textual languages have been created to simplify the process of designing circuits. Two popular textual languages are *VHDL* and *VERILOG*. VHDL was created by the U.S. Department of Defense, while VERILOG was created by Cadence Corporation. Although these two languages are comprehensive, they are also complex, and not really suitable for educational purposes. For this reason, we have invented a simpler language, *FHDL*, which will be used in this course and all other courses that make use of design automation. The following is an FHDL description of the circuit given above.

```
Sample:   circuit
          inputs     a,b,c
          outputs    q
gaprime:  not        a,aprime
gbprime:  not        b,bprime
gcprime:  not        c,cprime
gi1:      and        (aprime,bprime,cprime),i1
gi2:      and        (a,bprime,cprime),i2
gi3:      and        (a,bprime,c),i3
gi4:      and        (a,b,cprime),i4
gq:       or         (i1,i2,i3,i4),q
          endcircuit
```

FHDL permits the use of eight different gate types, **and**, **or**, **not**, **xor**, **nand**, **nor**, **buff**, and **xnor**. FHDL also provides the ability to specify ROMs, PLAs, State Machines, and various high level blocks such as Multiplexors and Registers. In this course we will be primarily concerned with the simulation of simple gates.

Before a circuit can be simulated, it must be translated from text into data-structures. Parsing a textual language such as FHDL is beyond the scope of these notes, but assuming that some mechanism exists for translating the textual description into data structures, two types of data structures are required, one that represents gates, and one that represents *nets*, the connections between gates. First, let us begin by specifying a simple circuit as follows.



This circuit can be translated into FHDL as follows.

```
Test1:      circuit
            inputs      a,b
            outputs     q
gq:         or          (i1,i2),q
gi2         and         (a,b),i2
gi1:        and         (aprime,bprime),i1
gaprime:    not         a,aprime
gbprime:    not         b,bprime
            endcircuit
```

This FHDL circuit could be translated into data structures as follows.

Gate Table

Index	*Name*	*Type*	*Inputs*	*Outputs*
0	gq	or	3,4	2
1	gi2	and	0,1	4
2	gi1	and	5,6	3
3	gaprime	not	0	5
4	gbprime	not	1	6

Net Table

Index	*Name*	*Type*	*Fanout*	*Value*
0	a	PI	1,3	0
1	b	PI	1,4	0
2	q	PO		0
3	i1		0	0
4	i2		0	0
5	aprime		2	0
6	bprime		2	0

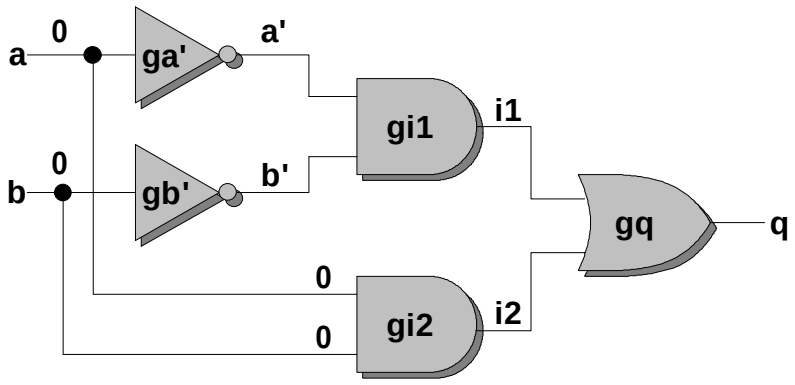
The Net Table will be used during the simulation to keep track of the value of each net. Because it is necessary to initialize the value of each net to *some* value, each net has been initialized to zero. These values are incorrect because the input and output of **gaprime** cannot both be zero, but the only way to assign consistent values to all nets is to simulate the circuit.

Although simulating a circuit seems straightforward, a naive algorithm can produce incorrect outputs. The first step in any simulation is to take input from the user and insert it into the net table. Since **a** and **b** are the primary inputs of the circuit, the user must supply values for **a**, and **b**. Suppose the user supplies the values **1,1**, and that these values are injected into the Net Table. A naive approach to simulating the circuit would be to

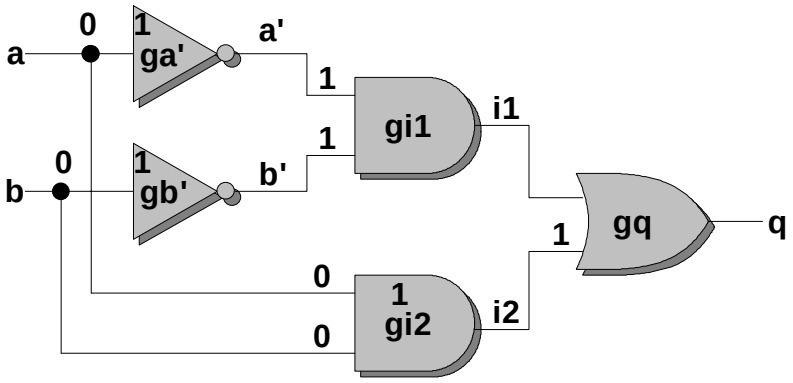
simulate all gates in the order that they appear in the gate table. This would result in the following values being computed for each net.

Net Table				
Index	***Name***	***Type***	***Fanout***	***Value***
0	**a**	**PI**	**1,3**	**1**
1	**b**	**PI**	**1,4**	**1**
2	**q**	**PO**		**0**
3	**i1**		**0**	**0**
4	**i2**		**0**	**1**
5	**aprime**		**2**	**0**
6	**bprime**		**2**	**0**

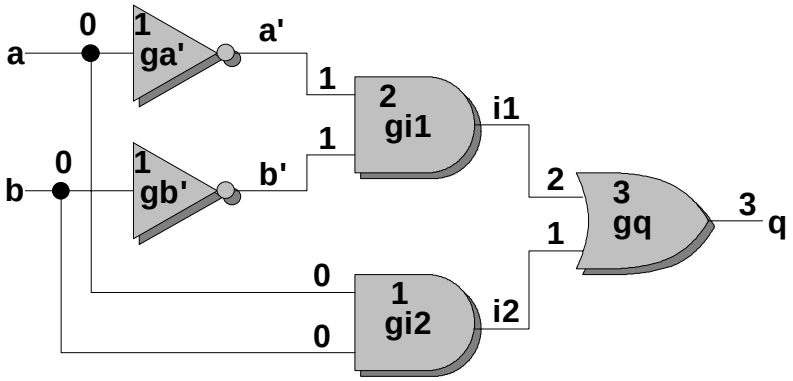
This is obviously incorrect, since these inputs will produce the value **1** at output **q**. The problem is that the **or** gate, **g2** was simulated before the values of its inputs had been computed correctly. To guarantee that gate simulations are performed in the correct order, it is necessary to make sure that the value of all gate inputs have been computed before the gate is simulated. This is done by *levelizing* the circuit, which assigns a level number to each net and each gate in the circuit. The process starts by assigning a level number of zero to each primary input.



Once all inputs of a gate have been assigned level numbers, then the gate and its outputs can be assigned level numbers. This is done by finding the maximum of the level numbers of the inputs add adding one, as illustrated below.



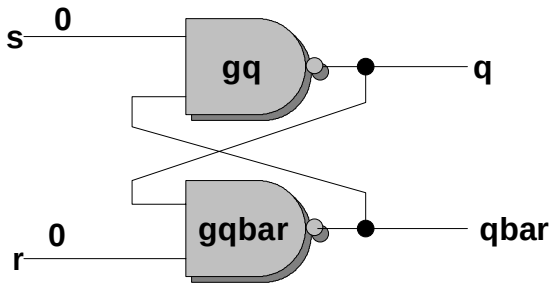
This process continues until all gates have been assigned level numbers, or until no more gates can be assigned level numbers, as illustrated in the following diagram.



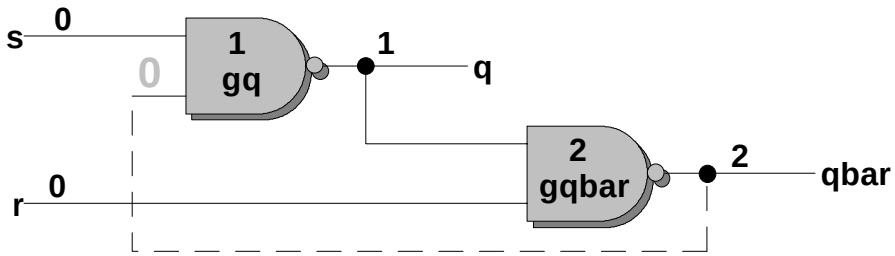
Once gates have been levelized, they are sorted into ascending order by level number, as illustrated in the following table.

Gate Table				
Index	*Name*	*Type*	*Inputs*	*Outputs*
3	gaprime	not	0	5
4	gbprime	not	1	6
1	gi2	and	0,1	4
2	gi1	and	5,6	3
0	gq	or	3,4	2

There are some circuits for which levelization does not work. Consider the following example. It is not possible to assign a level number to either gate in this circuit, because each gate has one input with no level number.



This circuit can be levelized and simulated by modifying the levelization procedure. When levelization "gets stuck" before all gates have been levelized, a gate is chosen at random and its inputs are assigned a virtual level number of zero. It is generally a gate with some known inputs that is chosen, but this is not absolutely necessary. The circuit is then levelized with respect to the virtual level number. This procedure will usually assign a second level number to the net with the virtual level number. Effectively, the net has been broken into two nets, one which appears to be a primary input and one which appears to be a primary output of the circuit. Broken nets are also referred to as *feedback arcs*. The levelization procedure is illustrated in the next diagram.

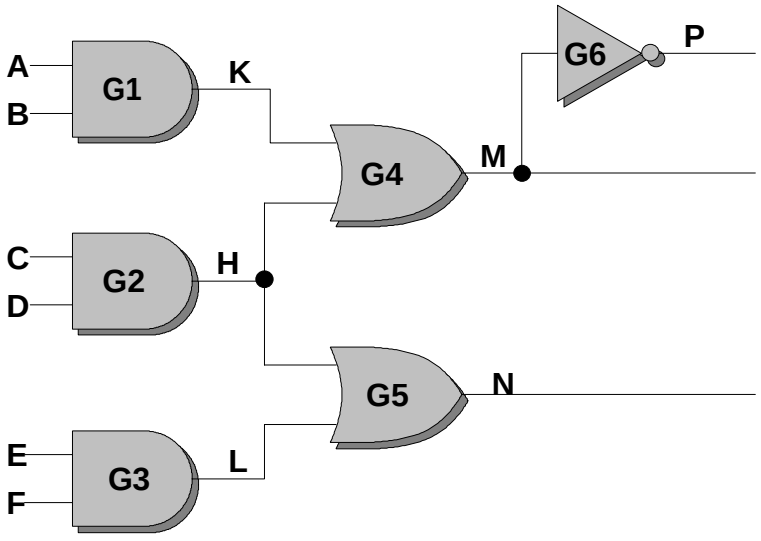


Instead of simulating each gate once, all gates are simulated repeatedly, in levelized order, until no feedback arc changes value. Since the circuit may go into a permanent oscillation, it is necessary to count the number of iterations, and halt the simulation after a predetermined number of iterations. If there are *n* feedback arcs, simulation can be terminated after 2^n iterations, although if *n* is large, a much smaller limit will be used. A limit of 10-20 is generally more than sufficient.

Chapter 2
Event Driven Simulation.

The previous chapter presented the concept of levelized simulation. Gates are sorted into levelized order before simulation, and each gate is simulated once per input vector. Observe, however, that after a simulation, the net table contains values for every net. Assume that for a particular gate **G**, the inputs of **G** have the same values for two consecutive input vectors. Since **G** represents a Boolean function, the output of G will have the same value for both vectors. Because the value of **G** from the first input vector is still in the table during the simulation of the second vector, the simulation results will be correct *even if **G** is not simulated* for the second input vector. The rule is that if the inputs of a gate don't change, then the outputs will not change. It is possible to use this rule to reduce the amount of simulation that is done for a particular circuit and set of input vectors.

Unfortunately, a naive approach will not result in any improvement in performance. The amount of work required to test the inputs of a gate for changes generally exceeds the amount of work needed to simulate the gate. It is not possible to simply test the inputs of every gate to see if they have changed. To be effective, it is necessary to eliminate both simulations and testing of inputs. The following diagram illustrates why this is possible.



Suppose the above circuit were simulated with two consecutive input vectors, (0,0,0,0,1,0), and (0,0,0,0,1,1). Since **F** has changed from **0** to 1, it is necessary to simulate **G3**, and since this causes a change in L, it is necessary to simulate **G5**. It is not necessary to simulate **G1** or **G2**, since their inputs have not changed. Furthermore, it is not necessary to test **K** or **H** for changes, since the gates driving them have not been simulated. By the same argument, it is not necessary to test **M** or **P** for changes.

The event-driven simulation algorithm is capable of taking advantage of these ideas. There are three phases to the algorithm, Input Processing, Event Processing, and Gate Simulation. In its most basic form the algorithm uses two queues and three separate processing routines. The algorithm is centered around the concept of an *Event*, which is a change in the value of a net. Internally, an event is represented by a data structure which contains a pointer to an entry in the net table, and the new value of the net, as illustrated below.

Events are collected together into the *Event Queue*, which contains all events that have occurred, but which have not yet been processed. The algorithm uses a second queue, called the *Gate Queue*, which contains a list of gates that have been scheduled for simulation, but have not yet been simulated.

The first set of events is created by the input processing routine. The pseudo-code for this routine is illustrated below.

```
Input_Processing:
{
    Read New_Input_Vector.
    For Each Primary Input PI do
        If the Value of PI in New_Input_Vector
            is different from the old value of PI Then
            Create a New Event.
            Set Net Identifier to PI
            Set New Value to the value from New_Input_Vector
            Add the event to the end of Event_Queue.
        End_If
    End_For
}
```

If the value of a primary input has not changed, then no event will be added to the Event Queue. The next step in Event-Driven simulation is to process all events in the event queue, and schedule gates for simulation. When an event is processed for a net **N**, all

gates in the *fanout of **N*** are scheduled for simulation. The fanout of **N** is the set of gates that use **N** as an input. Pseudo-code for the event processing routine is given below.

```
Process_Events:
{
    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
            Remove E from Event Queue
        End For
    End For
}
```

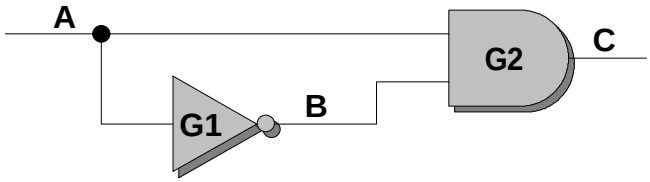
After Process_Events has completed execution, the Gate Processor is invoked. The Gate Processor will simulate each gate in the gate queue. Pseudo code for the Gate Processor is illustrated below.

```
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
}
```

The Gate Processor and the Event Processor are executed repeatedly until both queues become empty at the same time.

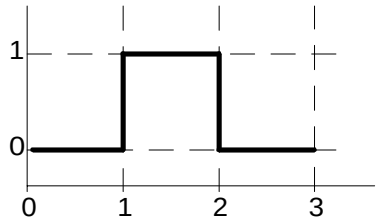
The timing models are different between Event-Driven simulation and Levelized Simulation. Event-Driven simulation uses the unit-delay model, while the Levelized Simulation algorithm uses the zero-delay model. In Event-Driven simulation, it is possible for one gate to be simulated many times during the processing of a single input vector.

To illustrate the difference between these two timing models, consider the following sample circuit.



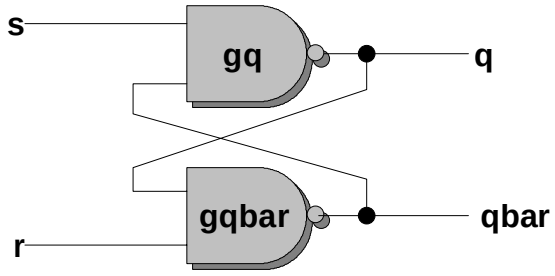
Suppose that the circuit has been simulated using the input **A=0**. This would result in the final values of **B** and **C** being **B=1** and **C=0**. If the circuit is being simulated with the new input **A=1**, levelized simulation **G1** would be simulated first, producing the output **B=0**. Then **G2** would be simulated producing the output **C=0**.

In the same situation, Event Driven Simulation would schedule an event for **A**. When this event is processed, **A** will assigned the value **1**, and both **G1** and **G2** will be scheduled for simulation. Suppose **G1** is simulated first. (The simulation order does not matter.) This produces an event for **B**, but does not change the value of **B**, which is currently **1**. When the gate **G2** is simulated, both **A** and **B** have the value **1**, so an event is created for **C** which will change its value to **1**. When the events for **B** and **C** are processed, the value of **B** will be set to **0**, and the value of **C** will be set to **1**. The gate **G2** will be scheduled for simulation a second time. The second simulation of **G2** will create an event for **C** which will set its value back to zero. The graph of the value of **C** looks as follows.



This is known as a *Static Hazard*. A *Hazard* occurs when a signal changes value more than once during the simulation of a single value. If the starting and ending values of the signal are the same then the hazard is a *Static Hazard*. If the starting and ending values are different then the hazard is a *Dynamic Hazard*.

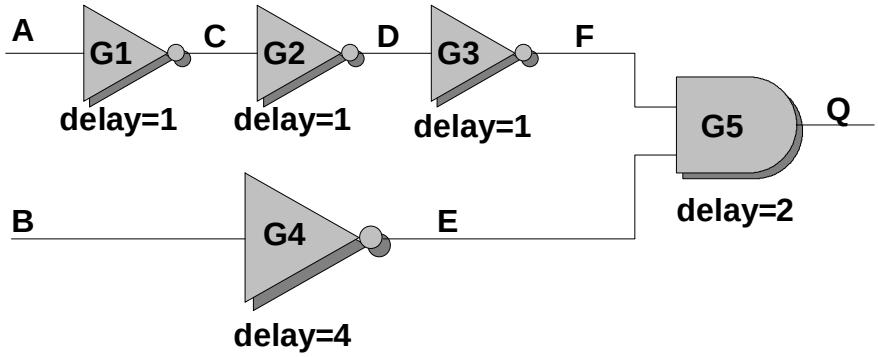
The difference between the two timing models is even more dramatic. Suppose the following circuit is simulated with the inputs **s=0,r=0** and then with **s=1,r=1**. In zero-delay simulation, the circuit will stablize with **q=0,qbar=1** or **q=1,qbar=0**, depending on which gate is simulated first. In event-driven simulation, this sequence of inputs causes the simulation to go into an unending oscillation. When circuits such as this are simulated in unit delay mode, a gate-simulation counter is required to detect oscillations. If a gate is simulated too many times, the simulation is forcably terminated, and the error is reported in some suitable fashion.



Because hazards and oscillations exist in real circuits, event-driven unit-delay simulation is considered to be more accurate than zero-delay levelized simulation.

Chapter 3
Multi-Delay Simulation.

As noted in the previous chapter, the unit-delay model of simulation is considered to be more accurate than the zero-delay model. An even more accurate method of simulation is to assign different delays to each gate. However, this complicates the process of simulation. For example, consider the following diagram.



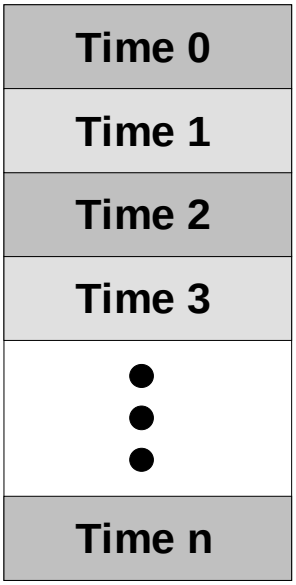
Suppose that there are changes in inputs A and B at time zero. Because of the delays in G1, G2, and G3, the change will not reach net F until time 3. The change in net E will arrive one time unit later, at time 4. This presents a problem for event-driven simulation. The changes at time zero will cause 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. A simpler method is to perform the simulation at time 0, but avoid processing the event until time 4.

Although delaying the processing of an event seems simple, there are problems. Suppose all events are time-stampped and placed in a single queue. 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, and process them. Although this may not be apparent, this algorithm is simply the **Repeated Minimum** sort algorithm, which is known to be of complexity $O(n^2)$. The the algorithm could keep the queue in ascending order by

searching for the proper position for an event before placing it in the queue, but this is the **Insertion Sort** algorithm, which is also known to be of complexity $O(n^2)$.

The two approaches that are actually used in practice are the *Timing Wheel*, and the *Priority Queue*. The timing wheel is a variation of the **Bucket Sort** algorithm, which is known to be of complexity O(n), while the priority queue is a variation of the **Heap Sort** algorithm, which is known to be of complexity O(n lg n). The Timing Wheel can be used only in restricted situations, while the Priority Queue can be used anywhere.

The Timing Wheel is used when delays are expressed as Integers, and the largest delay is reasonably small, say 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 structure is simply a zero-based array of queues, as illustrated below.



The most simplistic implementation would create one queue for each instant of simulated time, but this is not really 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. This queue can be reused to store events for time T+D. By this reasoning, the maximum number of queues required is D, where D is the maximum delay of any gate in the circuit. The queues are reused as illustrated below.

As the preceding diagram illustrates, the queues are reused as many times as necessary. Given a particular simulation time T, the index of the queue for that time is computed using the statement Q = T % D where % is the C notation for the remainder operation.

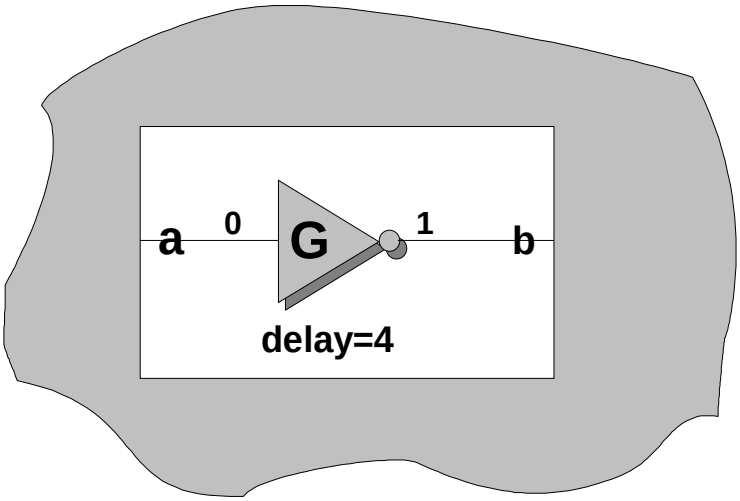
To keep indexes straight, it is necessary for the simulator to keep track of the current time. The event processing routines are much the same as those for unit delay simulation with some modifications. The Input Processing routine is identical to that for unit delay simulation. The new event processor is illustrated below. Changes are double-underlined. The variable *CurrentTime* must be initialized to -1 for the simulation of each input vector.

```
Process_Events:
{
    CurrentTime := CurrentTime + 1
    CurrentQueue := CurrentTime % MaxDelay
    For each Event E in Event_Queue[CurrentQueue] 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[CurrentQueue]
        End For
    End For
}
```

The new gate processor is illustrated below. As before, changes are double-underlined.

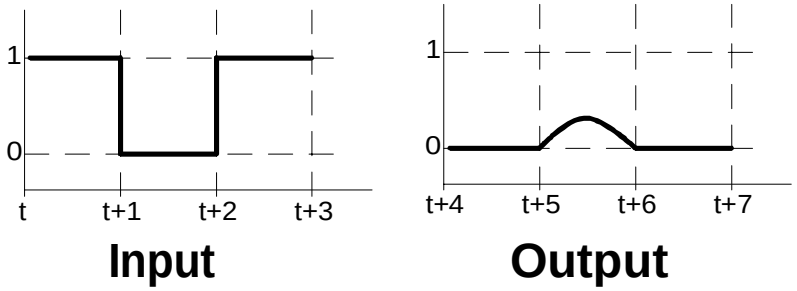
```
Gate_Processor:
{
    For Each Gate G in Gate_Queue do
        N := The Output of G
        D := The Delay of G
        Q := (CurrentTime+D) % MaxDelay
        Simulate G, put result in New_N
        If New_N is different from The current value of N Then
            Create a new event E
            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
}
```

These changes will produce an accurate Multi-Delay simulation only if there is never the possibility of queueing more than one event at a time for any net. This algorithm can produce erroneous results under these circumstances, as illustrated in the following diagram. G is assumed to be one gate in a much larger circuit, and the illustrated 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 0 to 1. This causes an event to be scheduled for b at time 105, which will change the value of b from 1 to 0. At time 102, another event is processed that changes the value of a from 1 back to 0. 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 0, but because the output of G is already 1, no event is created. Suppose no further events occur on a. At time 105 the event is processed for b changing its value to zero. The final values for a and b are both zero, an impossible situation.



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. There are two general approaches to handling this problem, which are based on two models of gate-delay. The first model is call Transport Delay, while the second is called Inertial Delay. 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. For a NOT gate, any pulse will be transported through the gate, regardless of its length.

The Inertial Delay model looks at the gate as a physical device which is not capable of changing state instantaneously. The device exhibits some resistance to change, which manifests itself as a delay between changes in the input and output. This model of gate delay is more accurate than the transport delay model. For a real not gate with a delay of 4, a short pulse on the input will not produce a sharp wave-form at the output. Instead, the wave-form may look something like the following.



The fuzzy pulse at the output will probably not be “recognized” as a 1 by any of the following gates. The rule for the Inertial Delay model is that a gate cannot transmit any pulse shorter than its delay.

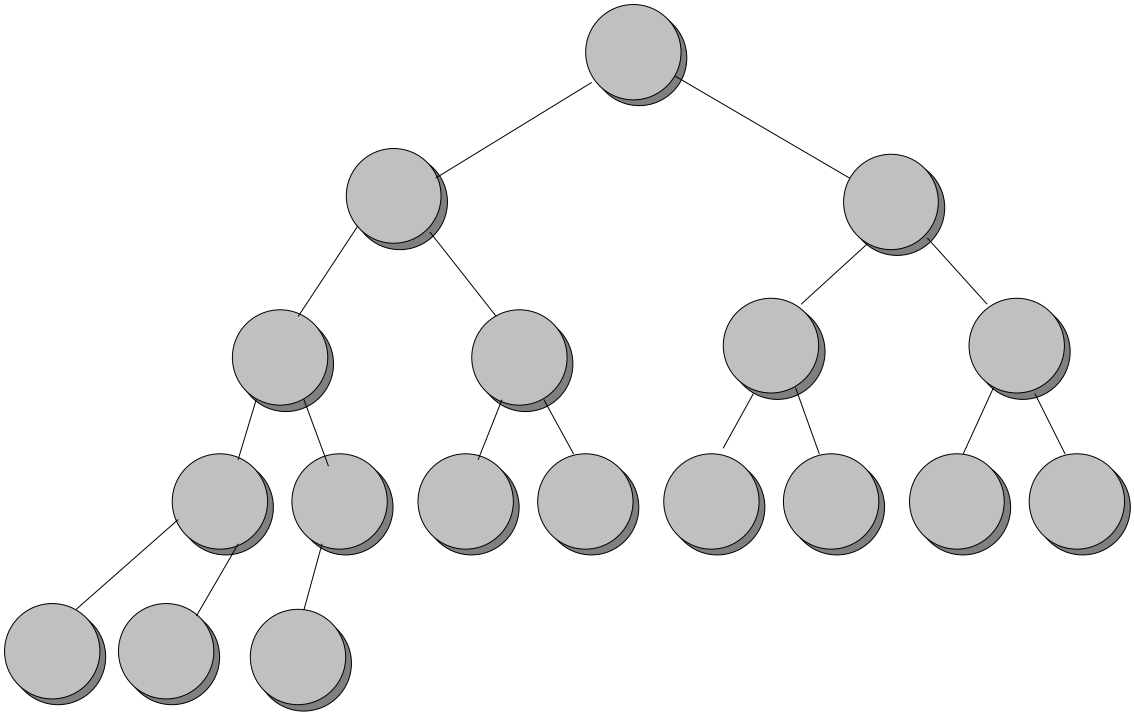
As a practical matter, one of the following steps must be taken in the simulator.

1. For Transport Delay, two different approaches can be used.
    a. Instead of testing for changes when a gate is simulated, unconditionally create an event. When the event is processed, test the value in the event against the current value of the net. If the values are the same, discard the event, otherwise schedule all gates in the fanout for simulation.
    b. After simulating the gate, examine the queue to see if there are any existing events for the same net. If so, compare against the value in the latest event rather than against the current value of the net.
2. For inertial delay, event-cancellation is used. When a gate is simulated, and there are events already in the queue for the output net, test the new value of the net against the value in the event. If the values are different, the events are assumed to cancel one another. The existing event is removed from the queue, and no additional event is placed in the queue.

As stated above, the timing wheel is not applicable to all situations. In particular suppose the maximum delay of a gate is large, say 1000 or more. Or suppose it is possible to use a floating-point number for the delay, which would allow events to occur at times such as

2.5, 2.25, 2.333, and so forth. There are variations on the Timing Wheel approach that have been used in such situations, but these techniques are all inherently $O(n^2)$ in complexity.

More modern approaches use the Priority Queue, which is modeled as a "nearly complete" binary tree, as in the following illustration.



(To be Continued ...)