Chapter 2

Levelized Simulation

2.1 Gate Simulation

Simulating individual AND, OR, and NOT gates is simple. First we allocate an integer for each net. We will keep the value of the net in the low-order bit of the integer, and we will use bit-level AND, OR and NOT operators to perform the simulations. This procedure can be extended to networks of gates, but some care is necessary. Consider the circuit of Figure 2-1. We will attempt to create a simulator for this circuit by generating a simulation statement for each gate. The result is given in Figure 2-2, with the simulation statements in alphabetical order by gate name. All nets will be initialized to zero.

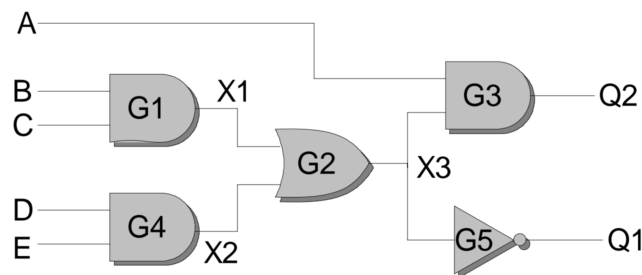


Figure 2-1. A Sample Circuit.

```
SimulateNet1()
{
   long A=0,B=0,C=0,D=0,E=0;
   long X1=0,X2=0,X3=0;
   long Q1,Q2;

   ReadNetValues(&A,&B,&C,&D,&E);
   /* Code for G1 */
   X1 = B And C;
   /* Code for G2 */
   X3 = X1 Or X2;
   /* Code for G3 */
   Q2 = A And X3;
   /* Code for G4 */
   X2 = D And E;
   /* Code for G5 */
   Q1 = Not X3
   PrintNetValues(Q1,Q2);
}
```

Figure 2-2. Simulation Code.

The program fails to produce the correct result for input A=1,B=0,C=0,D=1,E=1. The correct result is Q1=0,Q2=1, but the program prints Q1=0,Q2=0. The program is suspicious because the variable X2 is used before being assigned a value. Indeed, this is the problem that leads to the incorrect result. The simulation statements are not in the correct order. To correct this problem, we must use a technique called *levelization*.

2.2 Levelization

The problem with Figure 2-2 is that gates are not simulated in the correct order. Figure 2-3 shows a valid simulation of the circuit. We start by assigning binary values to the primary inputs, and proceed by propagating those values through the gates to the primary outputs. No gate is simulated until its input values are available. Arranging simulation statements in this order gives the code of Figure 2-4.

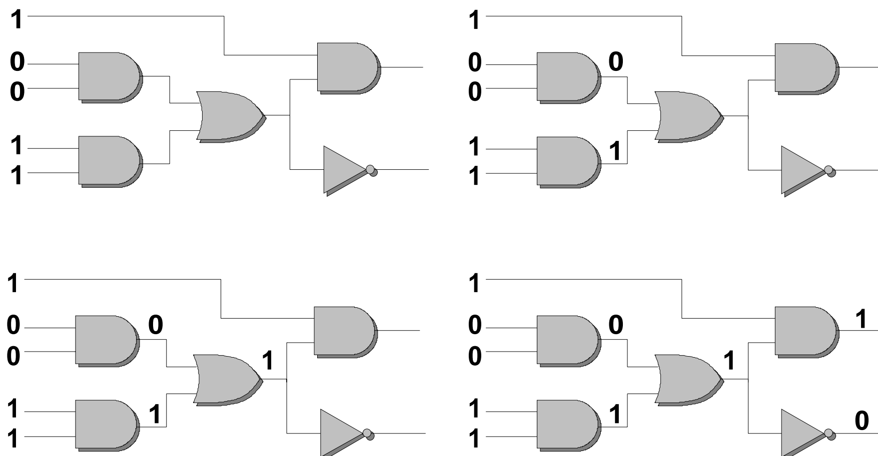


Figure 2-3. A Valid Simulation.

```
SimulateNet1()
{
   long A=0,B=0,C=0,D=0,E=0;
   long X1=0,X2=0,X3=0;
   long Q1,Q2;

   ReadNetValues(&A,&B,&C,&D,&E);
   /* Code for G1 */
   X1 = B And C;
   /* Code for G4 */
   X2 = D And E;
   /* Code for G2 */
   X3 = X1 Or X2;
   /* Code for G3 */
   Q2 = A And X3;
   /* Code for G5 */
   Q1 = Not X3
   PrintNetValues(Q1,Q2);
}
```

Figure 2-4. Corrected Simulation Code.

This example may seem a bit contrived, because the correct statement-ordering is obvious. However, when statements are being generated automatically it is necessary to have a systematic procedure for ordering them correctly. If the output of gate G1 is used, directly or indirectly, to compute the output of G2, then G1 must be simulated before G2. The process of determining correct statement ordering is called *levelization*. Levelization assigns a level number to each gate and each net in the circuit. The level number is used to determine the order of the simulation statements. Statements are sorted into ascending order by level number, which causes them to be executed in the proper order.

Levelization begins with the primary inputs and constant signals, which are assigned a level number of zero. The remaining level numbers are computed by processing gates one at a time. The algorithm maintains a set of ready gates, which are those gates whose inputs have been assigned level numbers. The level number of a gate must be larger than the level numbers of its inputs, so when the algorithm selects a gate for processing, it finds the maximum of the level numbers of the inputs and increments it by one. This number is assigned to the gate and each of its outputs. If the circuit contains wired connections, a slight modification to this procedure is required, because net may have more than one driving gate. In this case, the algorithm maintains a set of ready nets, which are nets whose driving gates have been assigned level numbers. The level number of a net must be greater than or equal to the level number of each of its driving gates, so when the algorithm processes a net it finds the maximum level number of each of the driving gates and assigns that to the net. The algorithm proceeds until there are no ready gates or ready nets. If the algorithm terminates without processing all gates and all nets, then the circuit contains feedback loops and cannot be levelized. These types of circuits must be simulated by other methods. The pseudo code for the levelization algorithm is given in Algorithm 2-1.

```
Variables:
   SET ReadyGates,ReadyNets;
   GATE g;
   NET n;

Part 1: Initialization
   // We use predecessor counts to determine when a gate or a net
   // should be placed in the ready set.
   Initialize ReadyGates and ReadyNets to The Empty Set;
   For each gate g
      Set PredecessorCount of g to InputCount of g;
   For each net n
      Set PredecessorCount of n to DriverCount of n;

Part 2: Bootstrapping
   // Primary inputs and constant signals have no driving gates.
   // They are always ready, so process them immediately.
   For all primary inputs and constant nets n do
      Set LevelNumber of n to 0;
      // Processing the net might make some gates ready.
      For all gates g in the fanout of n do
         Decrement PredecessorCount of g;
         If PredecessorCount of g is zero Then
            Add g to ReadyGates;
         EndIf
      EndFor
   EndFor

Part 3: Main Loop
   // In this loop we process all ready gates and ready nets.
   // We continue processing until all ready items have been
   // exhausted.
   While ReadyGates is not Empty or ReadyNets is not Empty do

      // process gates here
      If ReadyGates is not Empty Then
         Select gate g and remove it from ReadyGates;
         Set MaxLevel to 0;
         For each net n that is an input to g do
            If LevelNumber of n is greater than MaxLevel Then
               Set MaxLevel to LevelNumber of n;
            EndIf
         EndFor
         Increment MaxLevel by 1;
         Set LevelNumber of g to MaxLevel;
         For each output n of g do
            Decrement PredecessorCount of n;
            If PredecessorCount of n is equal to 0 Then
               Add n to ReadyNets
            EndIf
         EndFor
      EndIf

      // process nets here
      If ReadyNets is not Empty Then
         Select Net n and remove it from ReadyNets;
```

```
            Set MaxLevel to 0;
            For each driving gate g of n do
               If LevelNumber of g is greater than MaxLevel Then
                  Set MaxLevel to LevelNumber of g;
               EndIf
            EndFor
            Set LevelNumber of n to MaxLevel;
            For each gate g in the fanout of n do
               Decrement PredecessorCount of g;
               If PredecessorCount of g is equal to 0 Then
                  Add g to ReadyGates
               EndIf
            EndFor
         EndIf

      EndWhile

   Part 4: Error Check
      If there exist elements with no level number Then
         Print Error Message
```

Algorithm 2-1. Levelization.

The level of a net is also equal to the maximum number of gates through which a signal must pass to reach the net. Figure 2-5 illustrates the levelization process.

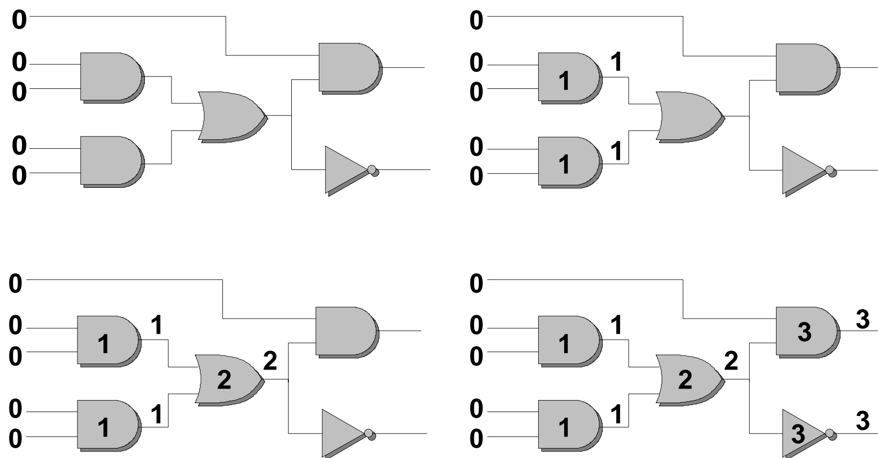


Figure 2-5. The Levelization Process.

There are some circuits, such as that shown in Figure 2-6, for which levelization does not work. If levelization is attempted, it will stop with level numbers assigned to nets **s** and **r**. Neither gate can be levelized, because *q* and *qbar* are not primary inputs or constants. Neither q nor qbar can be levelized because neither of their driving gates can be levelized. In general, levelization will fail for sequential circuits.

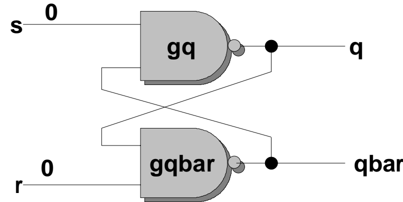


Figure 2-6. A Circuit that Cannot be Levelized.

It is possible to modify the process to handle sequential circuits, but the results are not always satisfactory. Consider a synchronous sequential circuit in which every cycle in the logic contains at least one synchronous flip-flop. The simulation can be broken into two phases, one in which flip-flops change state, and one in which all other gates are simulated. Removing the synchronous flip-flops from the circuit causes it to become a combinational circuit, which can be levelized easily. The flip-flops themselves can be treated as isolated logic elements. During simulation of the combinational portion of the circuit, the flip-flop outputs can be treated as if they were primary inputs and flip-flop inputs can be treated as if they were primary outputs.

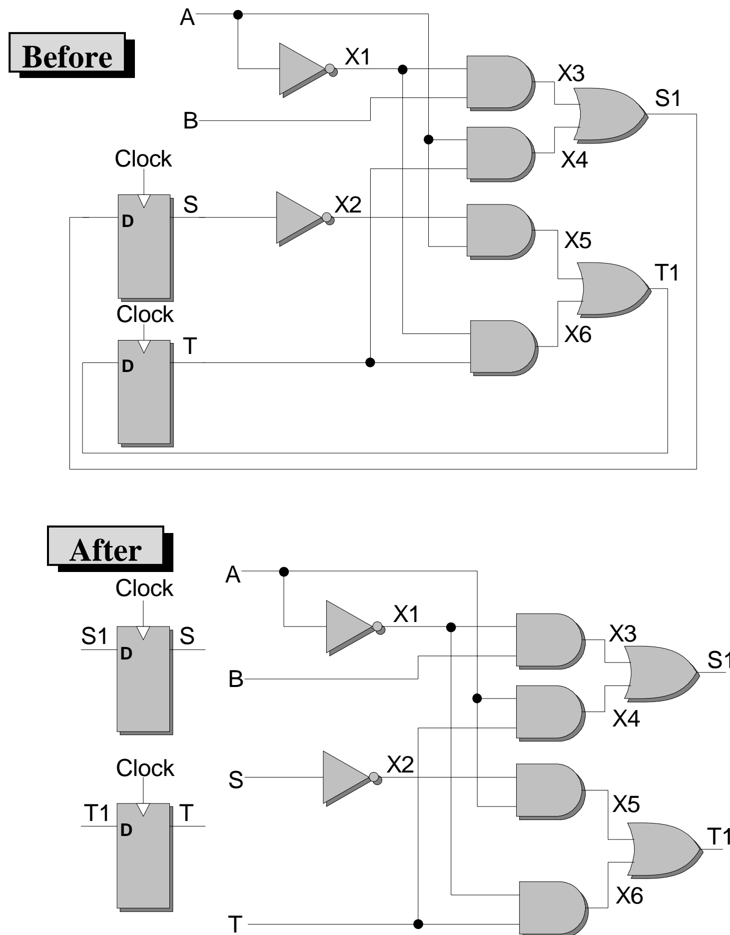


Figure 2-7. Breaking a Synchronous Circuit.

It is also possible to levelize asynchronous sequential circuits, such as that pictured in Figure 2-6, by forcibly assigning level numbers to circuit elements that are not levelized automatically. Consider the circuit of Figure 2-6. The levelization procedure will terminate before all gates have been assigned level numbers. When this happens, an unlevelized gate is chosen arbitrarily, and the gate is forcibly levelized. Any gate inputs that do not have level numbers are assigned a virtual level number of zero. The circuit is then levelized with respect to the virtual level number. Eventually a second level number will be assigned to the net with the virtual level number. For all practical purposes, the net has been broken into two nets, one which acts as a primary input and one which acts as a primary output. It may be necessary to break several nets to complete the levelization of the circuit. We call the broken nets *feedback arcs*. The levelization procedure is illustrated in Figure 2-8.

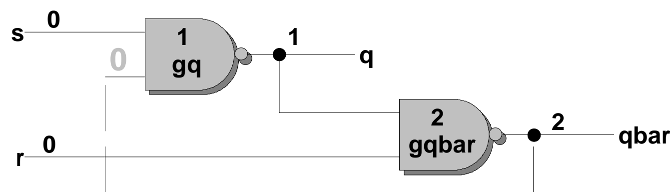


Figure 2-8. Levelizing an Asynchronous Circuit.

2.3 Interpreted Simulation

Simulation algorithms can be broken into two broad categories, *compiled* simulators and *interpreted* simulators. In both types of algorithms a circuit is compiled into a set of internal tables. Compiled simulators translate these tables into executable code which then must be compiled by a conventional compiler, while interpreted simulators use the internal tables directly to simulate the circuit. There are advantages to both techniques. Interpreted simulators can return results more quickly because the second compilation step is not necessary, while compiled simulators generally provide better performance.

Although levelized compiled code (LCC) simulation is more common than levelized interpreted simulation, we will begin by discussing the interpreted techniques. Interpreted levelized simulation can be further broken down into three different techniques based on the type of circuit being simulated. The techniques used for combinational circuits are basic to all algorithms and will be discussed first. Two different enhancements will be discussed, one for synchronous sequential circuits and one for asynchronous circuits.

2.3.1 Combinational Circuits

Throughout this section, we will use the circuit of Figure 2-9 to illustrate the interpreted simulation algorithm.

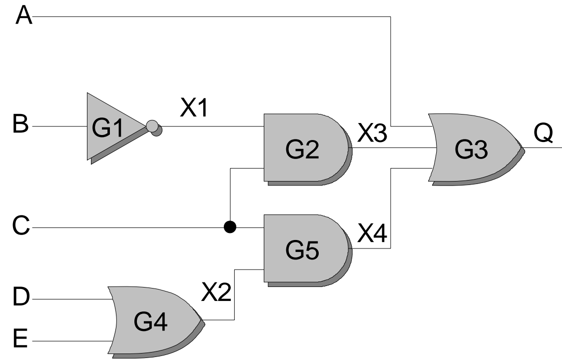


Figure 2-9. A Sample Circuit.

The tables corresponding to this circuit are given in Figure 2-10. These illustrations are a simplified view of the data structures that will be used to represent the circuit. Each table row is a record with several components. The Inputs, Outputs, and Fanout are contained arrays, with an additional data element to give their size. These arrays will contain pointers to other table elements. To simplify the presentation we will use the simple indices given in the Index column instead of full addresses. When the table is sorted, the index assigned to a row will move with the row.

Gate Table					
Index	*Name*	*Type*	*Inputs*	*Outputs*	*Level*
0	G1	or	1	6	x
1	G2	and	6,2	8	x
2	G3	and	0,8,9	5	x
3	G4	not	3,4	7	x
4	G5	not	2,7	9	x

Net Table				
Index	*Name*	*Type*	*Fanout*	*Value*
0	A	PI	2	x
1	B	PI	0	x
2	C	PI	1,4	x
3	D	PI	3	x
4	E	PI	3	x
5	Q	PO		x
6	X1		1	x
7	X2		4	x
8	X3		2	x
9	X4		2	x

Figure 2-10. Circuit Tables.

Before performing any simulation it is necessary to compute level numbers and sort the gate table into ascending sequence as shown in Figure 2-11. Note that the Index values are no longer in ascending sequence.

Gate Table					
Index	*Name*	*Type*	*Inputs*	*Outputs*	*Level*
0	G1	or	1	6	0
3	G4	not	3,4	7	0
1	G2	and	6,2	8	1
4	G5	not	2,7	9	1
2	G3	and	0,8,9	5	2

Figure 2-11. The Levelization Step.

During simulation, the values of the nets are contained in the net table. The first step in simulating an input vector is to read the vector, parse it, and place the values of the primary inputs into the net table. Figure 2-12 illustrates the results of reading a single input vector with the unaffected nets omitted for clarity. The simulation algorithm is given by Algorithm 2-2.

Net Table				
Index	*Name*	*Type*	*Fanout*	*Value*
0	A	PI	2	1
1	B	PI	0	1
2	C	PI	1,4	0
3	D	PI	3	0
4	E	PI	3	1

Figure 2-12. Reading the Input Vector.

```
While there are more input vectors do
   Read input vector;
   For I = 1 to NumberOfGates do
      Simulate(SortedGates[I]);
   EndFor
   Print results;
EndWhile
```

Algorithm 2-2. The Interpreted Combinational Algorithm.

To simulate an input vector, it is necessary to simulate each gate in the sorted gate table in ascending order by level number. The function Simulate of Algorithm 2-2 is assumed to provide this function. Algorithm 2-3 gives the structure of this function. (In a full-blown simulator, many more gate types would be available.) Figure 2-13 gives the result of running Algorithm 2-2 on the table of Figure 2-11.

Net Table				
Index	***Name***	***Type***	***Fanout***	***Value***
0	A	PI	2	1
1	B	PI	0	1
2	C	PI	1,4	0
3	D	PI	3	0
4	E	PI	3	1
5	Q	PO		1
6	X1		1	0
7	X2		4	1
8	X3		2	0
9	X4		2	0

Figure 2-13. Simulation Results.

```
SimulateGate(Gate)
{
   Boolean Result;

   Case Type of Gate

   And:
      Result = Value of Gate.Inputs[1];
      For I = 1 to InputCount of Gate do
         Result = Result And Value of Gate.Inputs[I];
      EndFor
      Set Value of Gate.Outputs[1] to Result;

   Or:
      Result = Value of Gate.Inputs[1];
      For I = 2 to Gate.InputCount do
         Result = Result Or Gate.Inputs[I].Value;
      EndFor
      Gate.Outputs[1].Value = Result;

   Not:
      If Value of Gate.Inputs[1] is 0 Then
         Set Value of Gate.Outputs[1] to 1;
      Else
         Set Value of Gate.Outputs[1] to 0;
      EndIf

   EndCase;
}
```

Algorithm 2-3. The Gate Simulation Function.

Some modifications to Algorithm 2-2 are required for sequential circuits. For synchronous circuits, the changes are minor, but more substantial modifications are required for asynchronous circuits.

2.3.2 Synchronous Circuits

All sequential circuits are characterized by cycles in their logic diagrams. For a circuit to be synchronous, all logic loops must be broken by clocked master-slave flip-flops. Furthermore, inputs must be applied during one correct clock phase and held constant during the opposite phase. These restrictions permit synchronous circuits to be simulated almost as if they were combinational circuits. Furthermore, since most sequential circuits are synchronous, these restrictions are not as severe as they might first seem.

We will use the circuit of Figure 2-14 as an example for this algorithm. After generating the internal tables, the first step is to create two separate tables for the gates, the first will contain all flip-flops and the second will contain all combinational gates. The tables for this circuit are given in Figure 2-15.

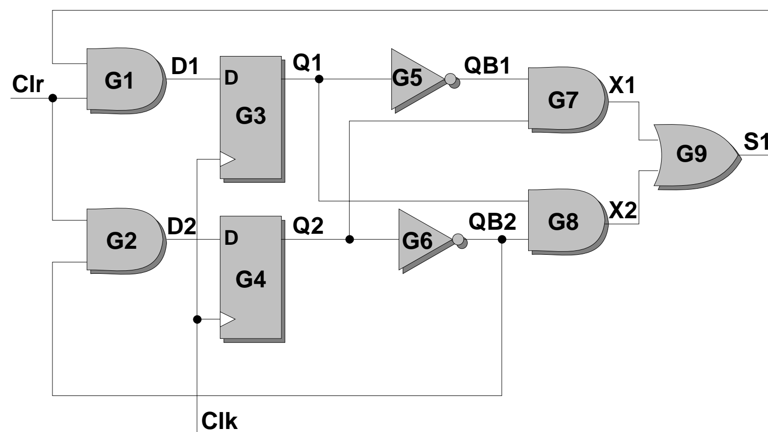


Figure 2-14. A Synchronous Circuit.

Gate Table					
Index	*Name*	*Type*	*Inputs*	*Outputs*	*Level*
0	G1	and	0,10	4	x
1	G2	and	0,7	5	x
3	G5	not	2	6	x
4	G6	not	3	7	
5	G7	and	6,3	8	
6	G8	and	7,2	9	
7	G9	or	8,9	10	

FlipFlop Table					
Index	*Name*	*Type*	*Inputs*	*Outputs*	*Value*
0	G3	dff	1,4	2	0
1	G4	dff	1,5	3	0

Net Table				
Index	*Name*	*Type*	*Fanout*	*Value*
0	Clr	PI	0,1	x
1	Clk	PI		x
2	Q1	PI	3,6	x
3	Q2	PI		x
4	D1	PO		x
5	D2	PO		x
6	QB1		1	x
7	QB2		4	x
8	X1		2	x
9	X2		2	x
10	S1			x

Figure 2-15. Tables for a Synchronous Circuit.

The synchronous simulation algorithm is given by Algorithm 2-4. The flip-flops are simulated in a separate phase from the combinational gates. Since correct functioning of the circuit requires flip-flop inputs to be held constant while the internal state changes, it doesn't matter whether the flip-flops are simulated before or after the combinational gates.

```
While there are more input vectors do
   Read input vector;
   For I = 1 to NumberOfFlipFlops do
      Simulate(FlipFlop[I]);
   EndFor
   For I = 1 to NumberOfGates do
      Simulate(SortedGates[I]);
   EndFor
   Print results;
EndWhile
```

Algorithm 2-4. The Synchronous Interpreted Algorithm.

The gate simulation algorithm of Algorithm 2-3 must be expanded to include the D flip flop. Algorithm 2-5 gives the code for simulating the master/slave D flip flop. The flip-flop has an internal state which we call the value of the flip-flop. This internal state is used to model the connection between the master flip-flop and the slave flip-flop. When the clock is active, the D input is copied into the internal state, and when the clock goes inactive, the internal state is copied to the output.

```
Dff:
   If Value of Gate.Inputs[1] is 1 Then
      Set Value of Gate to
            Value of Gate.Inputs[2];
   Else
      Set Value of Gate.Outputs[1] to
            Value of Gate;
   EndIf
```

Algorithm 2-5. Simulating a Master-Slave D-Flip-Flop.

Although synchronous simulation is efficient, it may not be able to detect certain design errors in the circuit. This is especially true if there are synchronization failures that cause the circuit to have asynchronous behavior. The forced synchronous simulation may cause these failures to be masked. Because of this, it may be necessary to treat the circuit as an asynchronous circuit during the early phases of the design.

2.3.3 Asynchronous Circuits

One of the simplest asynchronous sequential circuits is the RS flip-flop shown in Figure 2-6. This circuit can be levelized as shown in Figure 2-7, and it can be simulated using the algorithm given in Algorithm 2-6. Recall that the levelization was forced by breaking one or more nets in two. These nets are called feedback arcs and they play a special role in the asynchronous simulation algorithm. A new input vector is read, and the circuit is simulated repeatedly until no feedback arc changes value.

Algorithm 2-6 will give correct results in most cases, but it will fail in others. In particular, the simulation of the circuit from Figure 2-7 will not oscillate on the 0,0 to 1,1 transition as it should. This is because the output of gate *gq* is immediately available to gate *gqbar*. This error can be corrected by re-levelizing the circuit using two feedback arcs as is shown in Figure 2-16, but it is difficult to do this correctly without some form of human intervention. Correct simulation of asynchronous circuits is best done using the event driven algorithms of Chapter 3.

```
While there are more input vectors do
   Read input vector;
   Copy all Feedback Arc values to OldFeedback;
   While True
      For I = 1 to NumberOfGates do
         Simulate(SortedGates[I]);
      EndFor
      Print results;
      Copy all Feedback Arc values to NewFeedback;
      If NewFeedback = OldFeedback Then break;
      OldFeedback = NewFeedback;
   EndWhile
EndWhile
```

Algorithm 2-6. The Asynchronous Interpreted Algorithm.

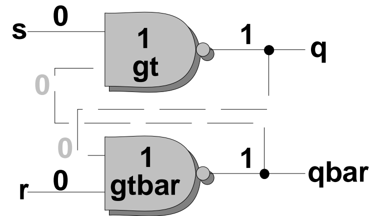


Figure 2-16. Re-levelizing an Asynchronous Circuit.

2.4 Compiled Simulation

Interpreted simulation gives results quickly, but it can be inefficient. Consider the following statement, which can be used to simulate a two-input AND gate. On most computers this statement can be implemented using no more than three or four instructions. Simulating the same circuit using any of the interpreted would require the same number of instructions for the simulation itself, but many additional instructions for decoding the instruction type and determining the number of inputs..

A = B **And** C;

Using compiled simulation, we can eliminate much of the work done by the interpreted algorithm. Rather than looping through a table of instructions and then decoding the gate types, we can generate exactly those statements needed to simulate the circuit. For example, the pseudo-code of Figure 2-17 can be used to simulate the circuit of Figure 2-9. Simulation statements must be generated in level-number order, which gives rise to the name *Levelized Compiled Code (LCC) simulation*. The function of Figure 2-17 must be called by a main program that reads input vectors and prints outputs.

Variables:

```
int   A,B,C,D,E;
int   Q;
int   X1,X2,X3,X4;
```

```
Simulation Function:
SimulateCircuit()
{
   X1 = Not B;
   X2 = D Or E;
   X3 = X1 And C;
   X4 = C And X2;
   Q = A Or X3 Or X4;
}
```

Figure 2-17. Compiled Simulation Code.

2.5 Vector Packing

In levelized simulation the value of every net in a combinational circuit is recomputed for every input vector. This allows input vectors to be simulated in any sequence as long as the correct output sequence is presented to the user. Using vector packing, it also allows several input vectors to be simulated simultaneously. In the preceding discussion, we assumed that Boolean variables would be used to represent net values, but at the hardware level, there is no such thing as a Boolean variable. Boolean values are represented using 8-bit, 16-bit, or 32-bit integers. Figure 2-18 shows how 8-bit integers are used to represent Boolean values..

One = 0000 0001, **Zero** = 0000 0000

And	0000 0000	0000 0001
0000 0000	0000 0000	0000 0000
0000 0001	0000 0000	0000 0001

Not	
0000 0000	0000 0001
0000 0001	0000 0000

Figure 2-18. Eight-Bit Boolean Operations.

When integers are used to represent Booleans the high-order bits are unused. This is wasteful, since the instructions used to perform the Boolean operations act on all eight bits simultaneously. It is possible to accumulate eight input vectors pack them into the bits of 8-bit integers and simulate them all simultaneously. Additional time will be required to combine the vectors into suitable eight-bit values and to separate the results into eight individual outputs, but even for circuits of modest size, this will result in substantial savings. Using 16 or 32-bit integers will result in even larger savings. Figure 2-19 illustrates how a three-input circuit can be exhaustively simulated using only two instructions.

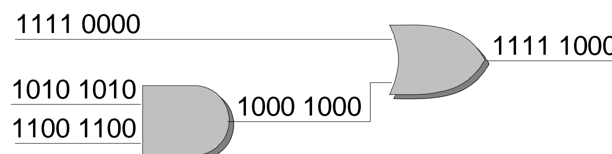


Figure 2-19. Bit-Parallel Simulation.

The technique illustrated in Figure 2-19 is called bit-parallel simulation. The simulation code that is required for bit parallel simulation is identical to that used for single vectors. The only change is in the input and output routines.

Bit parallel simulation is more difficult for sequential circuits, because there are dependencies between consecutive input vectors. These vectors must be simulated in strict sequential order to produce an acceptable output. However, under certain circumstances, it is possible to use bit-parallel simulation for sequential circuits. When a complex sequential circuit is designed, a large collection of tests is required to verify its correctness. Each test consists of a sequence of input vectors that verify the correctness of some feature. While there are sequential dependencies between the vectors of a single test, most tests are independent of one another. It is possible to pack several tests into a single word, so that each bit position in the word represents a single test. The collection of input vectors must be as long as the longest test. Figure 2-20 illustrates how this is done. For simplicity, only four tests are shown. The circuit has four inputs, A, B, C, and D.

Test 1			
A	B	C	D
0	0	1	1
0	1	1	0
1	0	0	1
1	1	1	0
1	0	0	1
1	1	1	0

Test 2			
A	B	C	D
0	1	1	1
1	0	1	1
0	1	1	1
1	0	0	1
0	1	0	1
1	0	0	1

Test 3			
A	B	C	D
1	1	1	0
1	1	0	0
1	0	0	0
1	0	1	0

Test 4			
A	B	C	D
1	0	0	0
1	1	0	1
0	1	0	0
0	1	0	1
0	1	0	0
0	1	0	1

Packed Tests			
A	B	C	D
0011	0110	1110	1100
0111	1011	1100	0101
1010	0101	0100	1100
1110	1001	1010	0101
1000	0101	0000	1100
1100	1001	1000	0101

Figure 2-20. Packing Independent Tests.

In the packed tests of Figure 2-20, the first bit of every four-bit group corresponds to Test 1, the second bit of every group corresponds to Test 2, and so forth. Since Test 3 is shorter than the others, zeros are inserted into bit position 3 following the end of the test.

2.6 Timing

No electronic circuit, including those that implement logic gates, can change state instantaneously. When the input of a NOT gate changes from one to zero, there will be some measurable delay between the change in the input and the change in the output. The

inherent delay that is present in every gate can affect the output of a circuit. For combinational circuits, delays cannot affect the final output, but there may be several intermediate transitional states before the outputs become constant. The same is true for synchronous sequential circuits, as long as the clock period is long enough to allow the combinational portion of the circuit to settle. Asynchronous sequential circuits, on the other hand, are quite sensitive to gate delays. Circuits that appear to be functionally correct may not function in the presence of delays.

Simulation algorithms differ in the way that gate delays are modeled. Although it is not apparent, levelized simulation behaves as though state changes were instantaneous. For this reason levelized simulation is said to implement the *Zero-Delay model* of simulation, implying that the delay of each gate is zero. For example, consider the circuit of Figure 2-21. Suppose that inputs A and B change simultaneously. In levelized simulation, the gate G1 will be simulated first producing an output of one. Next G2 will be simulated giving an output of one. The output Q will remain constant at one. However, if we assume that there is a non-zero delay between the change in B and the change in X1, there will be a brief period during which both inputs of G2 are zero. This will cause Q to become zero for a short period of time. Figure 2-22 shows the states of all five signals, assuming a constant, non-zero delay for both gates.

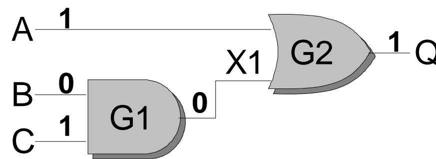


Figure 2-21. An Example with Delays.

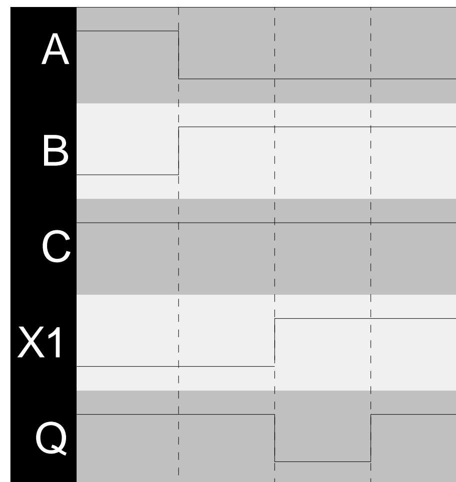


Figure 2-22. Net changes with delays.

The change in Q is called a *static hazard*, because the signal Q changes briefly even though it is supposed to be constant. The hazard may cause other portions of the circuit to function incorrectly. Timing models and the related issues will be discussed in depth in later chapters.

2.7 Logic Models

Up to this point we have assumed that nets carry binary values of either zero or one. In an real circuit, these values are represented as voltages, and changes between voltages are neither instantaneous nor discrete. When the voltage on a net changes from 0 to +5 (for example) the change will take some period of time, and during this time the net will have a continuous range of voltages from 0 through +5. The situation is complicated by the fact that the exact voltages used to represent one and zero are not fixed, but distributed over a range of values. Figure 2-23 illustrates how voltage ranges might be used to represent binary values.

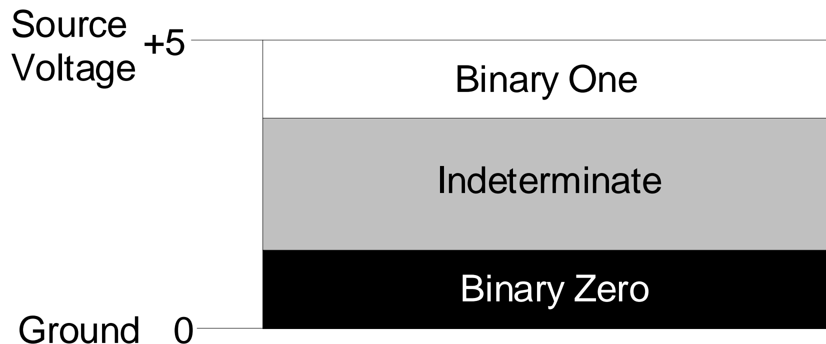


Figure 2-23. Representing values with voltages.

If the net has an intermediate value between 0 and +5 its value is neither one nor zero but indeterminate. The voltage may act like a one on some nets and a zero on others. It may have an intermediate or indeterminate effect on other gates. To accurately model this behavior, the value of the net must be modeled as a voltage rather than as a Boolean value. Furthermore, the dividing lines between zero, one, and the indeterminate state are not fixed, but depend on many factors, including the fanout of a net, and the size of the transistors used in a gate. In many cases we can ignore the intermediate behavior of the net and tag the net value as *unknown*. Effectively, this becomes a third logical value which we will designate as **U**. In addition to representing intermediate values, the logical value **U** has other uses. For example, when a circuit is first initialized it is not possible to predict the values of certain nets. These nets may actually have values of zero or one, we just don't know which. In this case we can assign the value U to the net, and after the reset cycle is complete, we can check to make sure that all nets have been initialized properly.

Another situation in which the unknown value is useful is when a circuit is determined to be in oscillation or in a metastable state. To illustrate, consider the circuit pictured in Figure 2-6. If s and r are both set to zero, and then simultaneously set to 1, the outputs will oscillate between one and zero. Eventually, the circuit will stabilize in some state, but the final state depends on slight differences in the delays of the two Nand gates and is not predictable at the logic level. The best that can be done is to detect the oscillation, and set both outputs to the **U** value.

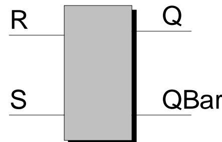
2.8 Conclusion

Levelized simulation is effective and relatively easy to implement. The timing model is inaccurate, and logic values other than zero and one are not handled properly. Levelized simulation is generally used in situations where detailed timing results are not as important as logical correctness. It is also an effective technique to use when a massive number of results are needed in a short period of time.

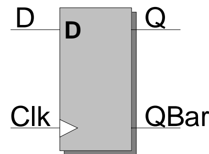
Levelization is an adaptation of the topological sort algorithm for directed acyclic graphs. As in levelization, topological sort is most often used for scheduling tasks or other items that can be represented as a directed acyclic graph. See [Knuth] for more information.

2.9 Exercises

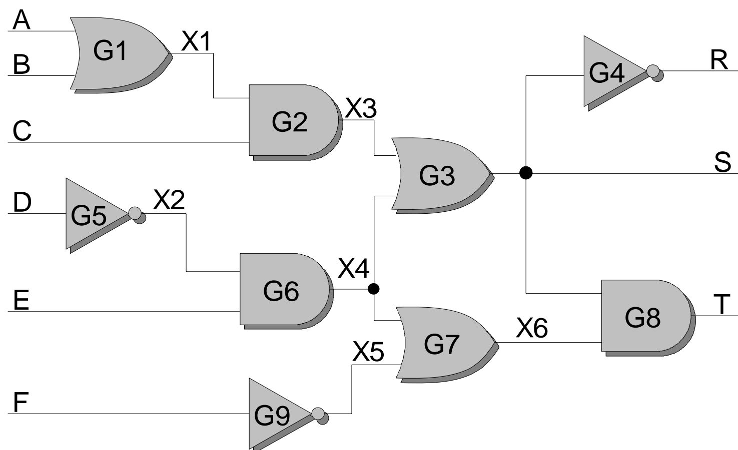
1. Using your favorite programming language, write the simulation code for the following RS flip-flop. Treat the flip-flop as a single gate. Do not attempt to simulate it as a pair of Nand gates. Display an error message on the 00-11 transition.



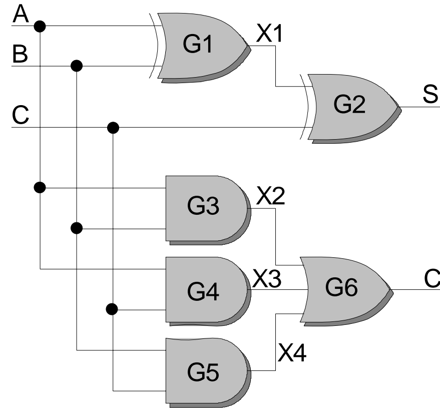
2. Using your favorite programming language, write the simulation code for the following D flip-flop.



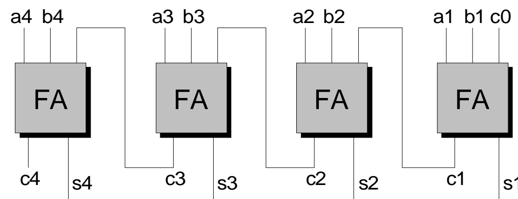
3. Write the simulation code for the Nand, Nor, Xor, and Xnor gates. (The Xnor gate is an Xor with an inverted output.)
4. Levelize the following circuit. Give level numbers for each net and each gate.



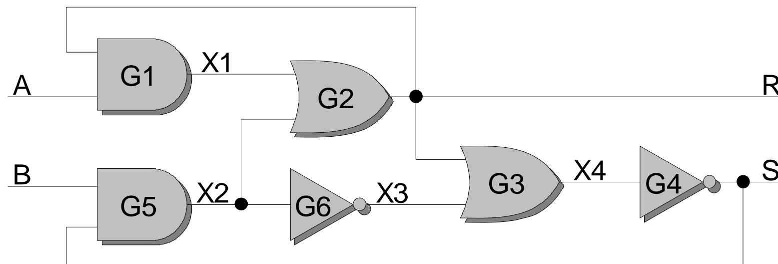
5. The following circuit is known as a full adder, because it adds three bits and produces sum and carry outputs. Levelize the full adder, giving level numbers for each gate and each net.



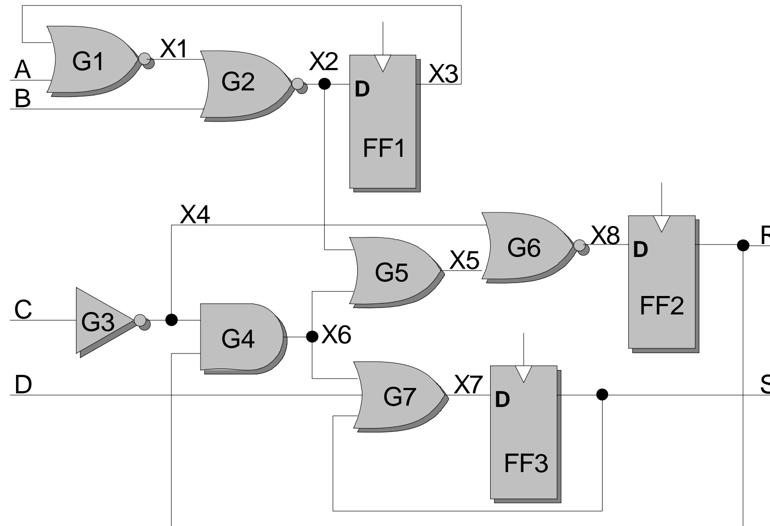
6. The full adder of Exercise 5 can be used to create a four-bit adder as illustrated below. Each of the blocks in this diagram represents a full-adder circuit. Expand this circuit converting each block to a full adder, and levelize the circuit. Give each net and each gate a unique name. Give level numbers for all gates and nets.



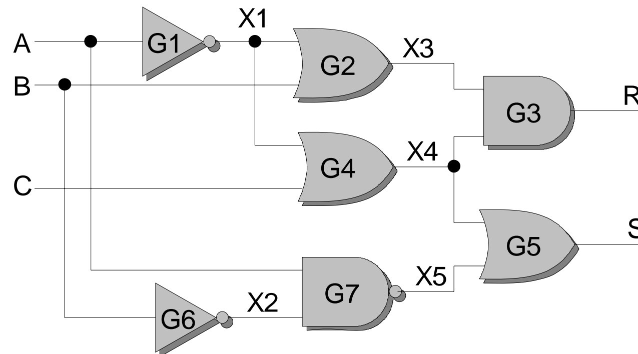
7. Levelize the following circuit. Give level numbers for each net and each gate. Identify all feedback arcs.



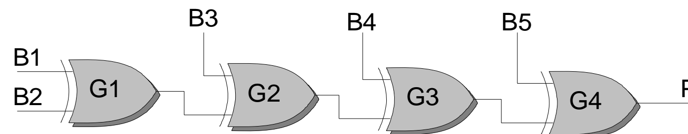
8. Levelize the following circuit. Separate the circuit into its two components. Give level numbers for all gates and nets.



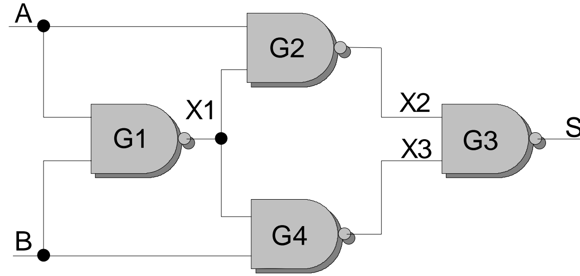
9. Create the gate and net tables for the following circuit. Initialize all nets to zero.



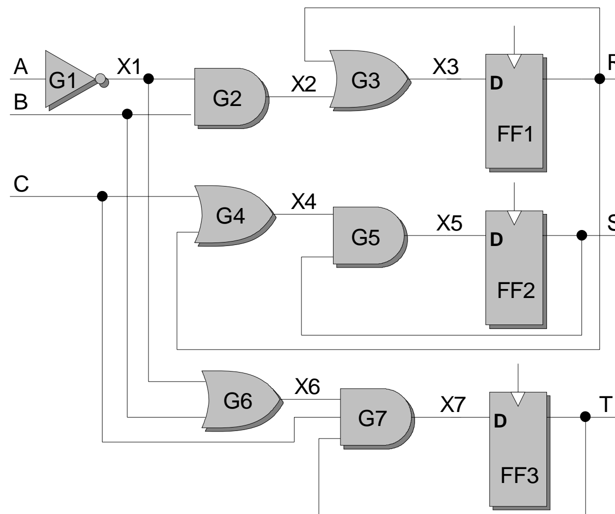
10. Levelize the circuit of Exercise 9. Show the sorted gate and net tables.
11. Simulate the circuit of Exercise 9 using the input vectors (A,B,C)=(0,0,0), (0,0,1), (1,0,1), and (1,1,0). Show the net table as it appears after simulating each input vector.
12. Using the interpreted levelized simulation technique, simulate the following circuit using inputs (B1,B2,B3,B4,B5)=(1,1,1,1,1), (1,0,1,0,1), (0,1,0,1,0), (1,0,0,1,0), (1,1,1,0,0), (0,0,1,0,0). Show the net table as it appears after simulating each input vector.



13. Using the interpreted levelized simulation technique, simulate the following circuit using inputs (A,B)=(0,0), (0,1), (1,0), (1,1).

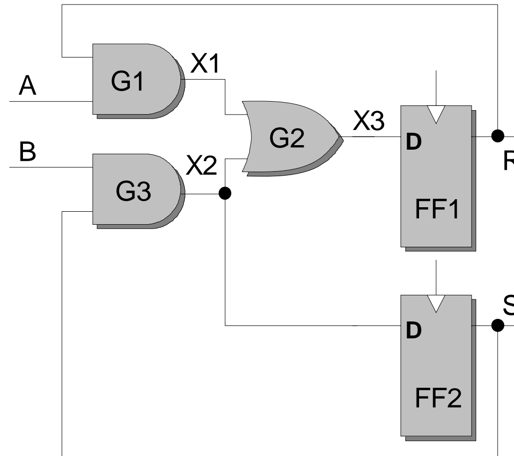


14. Using your favorite programming language, write the compiled simulation routine for the circuit of Exercise 9.
15. Using your favorite programming language, write the compiled simulation routine for the circuit of Exercise 12.
16. Using your favorite programming language, write the compiled simulation routine for the circuit of Exercise 13.
17. Write a program containing the code written for Exercise 16. Add appropriate input and output routines for reading input vectors and writing results. Run the program to simulate the circuit. Use the following input vectors: (A,B)=(0,0), (0,1), (1,0), (1,1).
18. Give the net and gate tables for the following circuit. Levelize the circuit, placing the flip-flops at the beginning of the list.

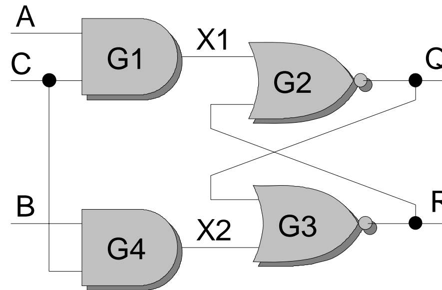


19. Simulate the circuit of Exercise 18 using the inputs (A,B,C)=(1,0,0), (0,0,0), (1,1,1), and (0,1,1).

20. Simulate the following synchronous sequential circuit using the inputs (A,B)=(0,0), (0,1), (1,0), (0,1), and (1,1).

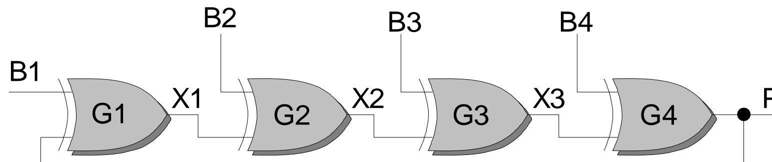


21. Give the net and gate tables for the following circuit. Levelize the circuit, and initialize all nets to zero.



22. Simulate the circuit of Exercise 21 using the inputs (A,B,C)=(1,0,1), (0,1,0), (0,1,1), (1,1,1), (0,0,1).

23. Simulate the following circuit with the inputs (B1,B2,B3,B4)=(0,0,0,0,0), and (1,0,0,0,0).



24. Suppose you are given a 4-input combinational circuit, and are asked to test the circuit for all 16 input combinations. Using vector packing, create one input vector to test all 16 combinations at once. Express your results as four 16-bit binary numbers.

25. Pack the following 16 vectors into five 16-bit words. Express your results in hexadecimal.

0,0,0,1,1	0,0,1,0,0	1,0,0,1,1	1,0,1,0,1
1,1,1,0,0	0,0,1,1,0	1,1,0,1,0	0,1,0,1,0
0,1,1,1,0	0,0,0,0,1	1,0,1,1,1	1,1,1,0,1
0,1,1,0,0,	0,1,0,0,0	0,0,1,0,0	1,0,1,1,1

26. Pack the following independent sets of vectors.

Test 1
1,0,0,1
0,1,1,1
1,0,1,0
1,1,0,0
0,0,1,1
1,0,1,1
0,0,0,1
0,1,0,0

Test 2
1,1,1,1
1,0,1,1
1,0,1,0
1,1,1,1
0,1,0,1

Test 3
0,0,0,0
1,0,0,1
0,1,1,0
1,1,1,1
0,1,1,1
1,0,0,0
1,1,1,1
0,0,0,1

Test 4
0,1,0,1
1,0,1,0
0,0,1,0
1,0,0,0
0,1,0,0
0,0,0,1

27. Pack the following independent sets of vectors.

Test 1
1,0,0
0,1,1
1,0,1
1,1,0
0,0,1

Test 2
1,1,1
0,1,1
0,1,0

Test 3
0,0,0
1,0,1
0,1,0
1,1,1
0,1,1

Test 4
1,0,1
1,1,0
0,1,0
1,0,0

Timing and Logic models

28. Three-valued truth tables can easily be computed from the two valued truth tables. To illustrate the method, consider the following partial truth table for the And function. We will fill in the two shaded cells.

And	0	1	U
0	0	0	
1	0	1	
U			

Let us start with the upper shaded cell. The U value can represent either a zero or a one, so the true inputs to the function are either (0,0) or (0,1). in both cases, the result the And function gives a zero, so the upper cell should look as follows.

And	0	1	U
0	0	0	0
1	0	1	
U			

For the second cell, the true inputs to the function are either (1,0) or (1,1). In the first case the And will give a 0, while in the second case the And function will give a 1. Therefore the second cell should contain the value U, as follows.

And	0	1	U
0	0	0	0
1	0	1	U
U			

Using these methods, complete the above table.

29. Create 3-valued truth tables for the following Boolean functions:. Or, Nand, Nor, Xor, Xnor, Not.

30. Create 3-valued truth tables for the RS flip-flop and the D flip-flop. Recall that the previous outputs of the flip flop are considered to be inputs to the function, as indicated in the following partial truth table.

S	R	Q_0	QB_0	Q_1	QB_1
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0		
...					
1	1	1	1	?	?

31. Using the compiled code technique, show the simulation code that would be generated for And, Or and Not gates. Assume that two bits are used to represent each net value, as follows.

Bit-Pair	Value
00	0
01	1
10	U
11	U

32. Under the same assumptions as the previous exercise, give the 3-value generated code for Nand, Nor, Xor, and Xnor gates.