Chapter 2

Levelized Simulation

2.1 Gate Simulation

Simulating basic AND, OR, and Not gates is a simple matter. Most modern programming languages have bit-level AND, OR and Not operators that can be used to perform the required simulations. One obvious way to construct a simulator for the circuit pictured in Figure 2-1 would be to allocate variables for each of the nets and then write simple assignment statements for each of the gates. Additional code would be required for reading input values and printing results.

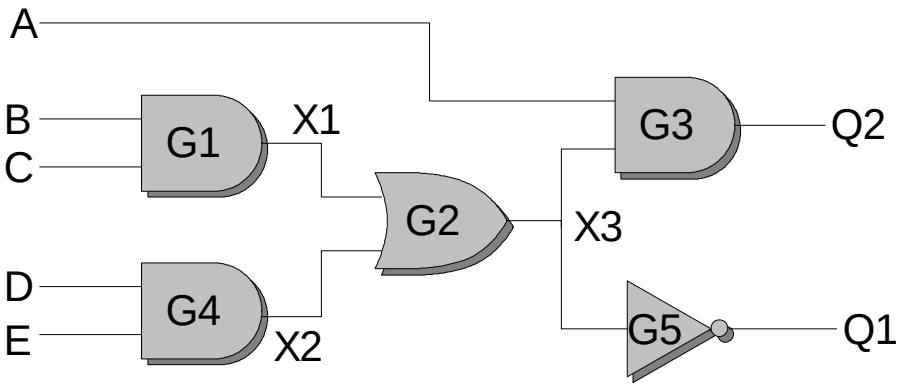


Figure 2-1. A Sample Circuit.

Unfortunately, the procedure is not quite so simple, as Figure 2-2 illustrates. Figure 2-2 contains the pseudo-code for simulating the circuit of Figure 2-1. Simulation statements have been created in alphabetical order by gate name.

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

   Read values for A, B, C, D, and 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
   Print the values of Q1 and Q2.;
}
```

Figure 2-2. Simulation Pseudo-Code.

The code of Figure 2-2 is suspicious, because the variable X2 is used before being assigned a value. Even if all new variables are initialized to zero, the code 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 produces Q1=0,Q2=0.

2.2 Levelization

The problem with the simulation code of Figure 2-2 is that gates are not simulated in the correct order. Figure 2-3 illustrates a valid simulation of this circuit. One starts by assigning binary values to the primary inputs, and proceeds by propagating those values through the gates to the primary outputs. A gate is not simulated until all of its input values are available.

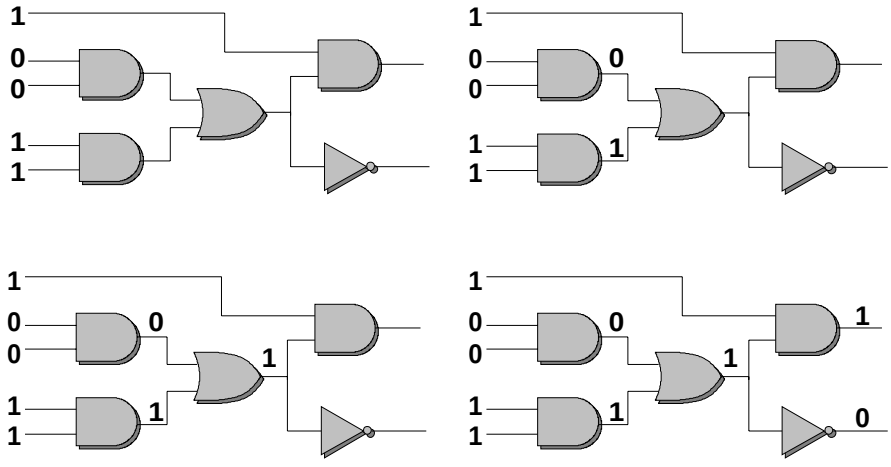


Figure 2-3. A Valid Simulation.

Although it is easy to simulate this circuit correctly by visual inspection, a more systematic technique is required for automatic simulation of the circuit. If the output of gate G1 is used to compute the output of G2, either directly or indirectly, then G1 must be simulated before G2. The process of determining the correct gate-simulation order is called levelization. Levelization assigns a level number to each gate and each net in the circuit. The process begins with the primary inputs of the circuit, which are assigned a level number of zero. Any constant signals are also assigned a level number of zero. A level number can be assigned to a gate only if all gate-inputs have been assigned level numbers. Similarly a net can be assigned a level number only if all driving gates have been assigned level numbers. (Most nets will have a single driving gate.) To assign a level number to a gate, the level numbers of the inputs are collected, and the maximum is extracted. The maximum is incremented by one, and the result is assigned to the gate as its level number. The process for nets is similar. The level numbers of the driving gates are extracted, and the maximum is obtained. The maximum is assigned to the net as its level number, without incrementing. In most cases the level number of a net is identical to the level number of the single driving gate. Algorithm 2-1 contains pseudo-code for the levelization process.

```
Levelize(MyCircuit)
{
   SET ReadyGates,ReadyNets;
   GATE g;
   NET n;

   ReadyGates = TheEmptySet;
   ReadyNets = TheEmptySet;
   For all primary inputs n of MyCircuit do
      n.level = 0;
      For all gates g in the fanout of n do
         If all inputs of g have level numbers Then
            Add g to ReadyGates;
         EndIf
      EndFor
   EndFor
   Repeat for constant one, constant zero signals;
   While ReadyGates is not Empty or ReadyNets is not Empty do
      If ReadyGates is not Empty Then
         Select gate g and remove it from ReadyGates;
         MaxLevel = 0;
         For each net n which is an input to g do
            If n.Level > MaxLevel Then MaxLevel = n.Level;
         EndFor
         g.Level = MaxLevel + 1;
         For each output n of g do
            If all driving gates of n have level numbers Then
               Add n to ReadyNets
            EndIf
         EndFor
      EndIf
      If ReadyNets is not Empty Then
         Select Net n and remove it from ReadyNets;
         MaxLevel = 0;
         For each driving gate g of n do
            If g.Level > MaxLevel Then MaxLevel = g.Level;
```

```
            EndFor
            n.Level = MaxLevel;
            For gate g in the fanout of n do
               If all inputs of g have level numbers Then
                  Add g to ReadyGates
               EndIf
            EndFor
         EndIf
      EndWhile
   }
```

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 a the net. Figure 2-4 illustrates the levelization process.

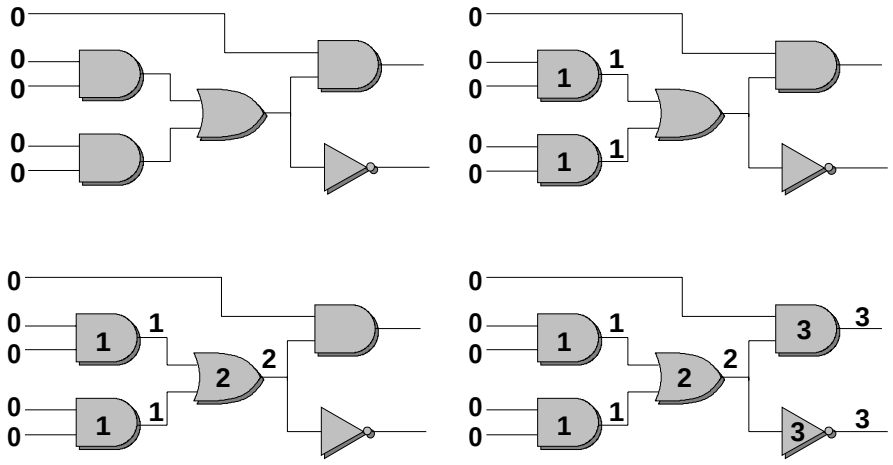


Figure 2-4. The Levelization Process.

Once gates have been levelized, they are sorted into ascending order by level number, as illustrated in the following table. Once level numbers have been assigned to all gates in the circuit, the circuit is *levelized* by sorting the gates into ascending order by level number. When the circuit is simulated, the gates will be simulated in ascending order by level number.

There are some circuits, such as that shown in Figure 2-5 for which levelization does not work. If levelization is attempted, it will stop with level numbers assigned to nets **s** and **r**, and no other level numbers. The logic-loop in this circuit prevents levelization from proceeding. This example illustrates that levelization is possible only for combinational circuits.

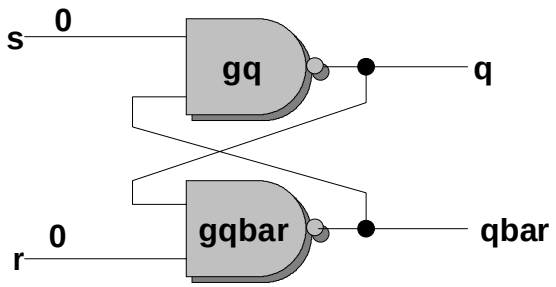


Figure 2-5. An Unlevelizable Circuit.

Although the simple levelization process fails for sequential circuits, it is possible to modify the process to handle such circuits. For synchronous sequential circuits, every logic loop must contain 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. During simulation of the other gates, 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. Effectively, the circuit is broken at the synchronous flip-flops. Since every logic-loop must contain at least one synchronous flip-flop, the resultant circuit is combinational, and can be levelized. Figure 2-6 illustrates this process.

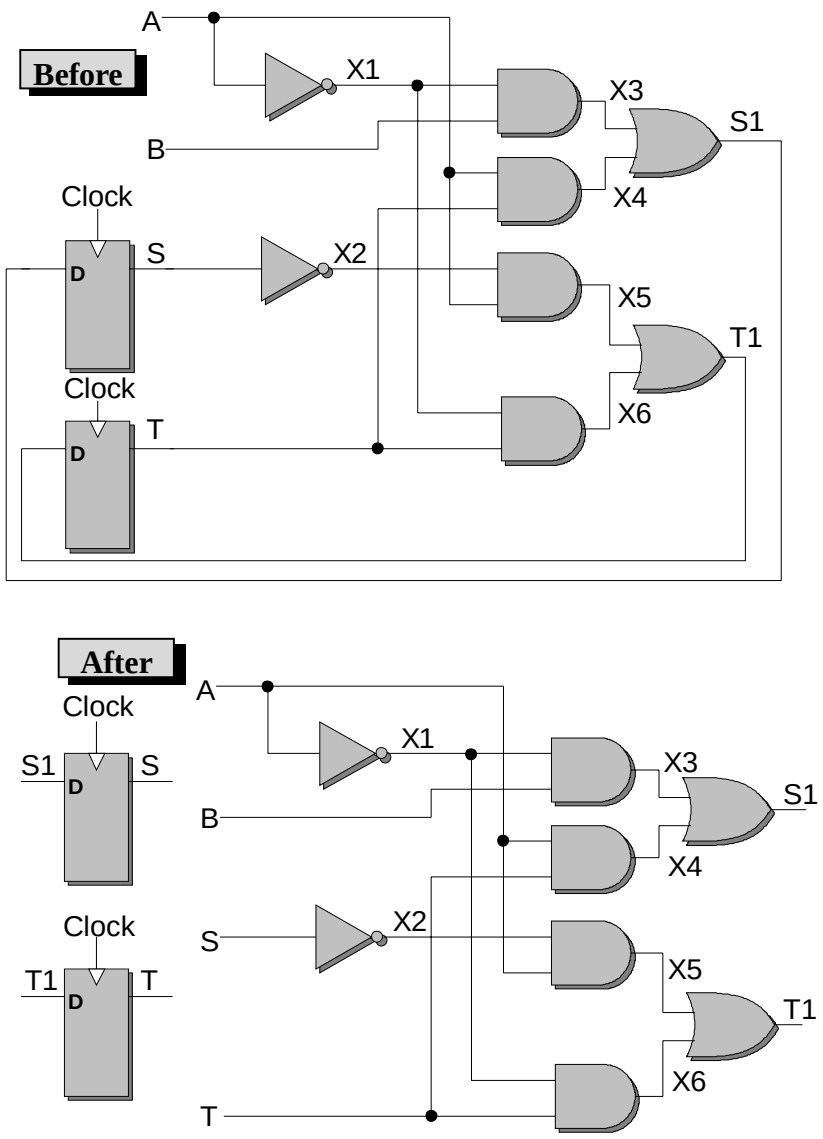


Figure 2-6. Breaking a Synchronous Circuit.

It is also possible to levelize asynchronous sequential circuits, such as that pictured in Figure 2-5, by modifying the levelization procedure. For this circuit, 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 force-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 been broken into two nets, one which acts as a primary input and one which acts as a primary output. Broken nets are referred to as *feedback arcs*. It may be necessary to create several feedback arcs to complete the levelization of the circuit. The levelization procedure is illustrated in Figure 2-7.

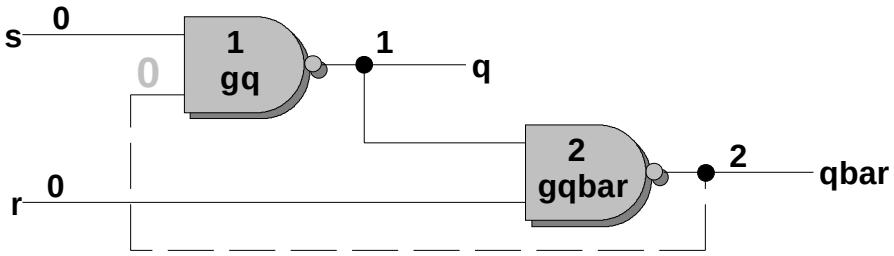


Figure 2-7. Levelizing an Asynchronous Circuit.

It is also necessary to modify the simulation procedure for asynchronous sequential circuits, as described in Section 2.2.5.

2.3 Interpreted Simulation

Once a circuit has been parsed and levelized, the net and gate tables can be used to simulate the circuit. Algorithm x illustrates the basic simulation technique. Simulation techniques that use the parser-generated tables to perform the simulation, are called interpreted techniques to distiguish them from the compiled simulation technques discussed in the next section. These terms have been borrowed from the theory of programming languages. Figure 2-8 gives a simple circuit that will be used to illustrate the simulation process.

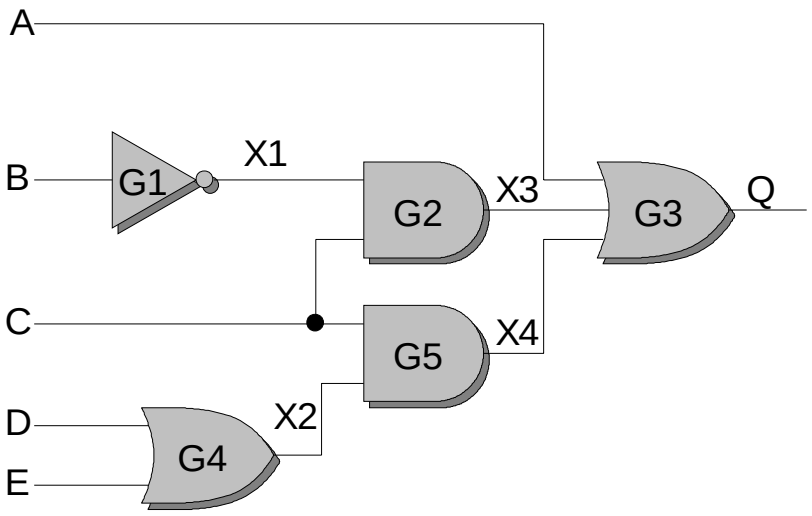


Figure 2-8. A Sample Circuit.

The tables corresponding to this circuit are given in Figure 2-9. These illustrations are a simplified view of the data structures that will be used to represent the circuit. Although many different implementation strategies could be used, it is necessary to make some assumptions about the underlying data structures to facilitate the description of the algorithms. Each table row will be assumed to be a record data structure with several components. The Inputs, Outputs, and Fanout components will be arrays, with additional components giving their size. The contents of the Inputs and Outputs arrays will be

pointers to structures in the other table. The tables themselves will consist of arrays of pointers with variables giving their. The Index column of the tables is assumed to represent the pointers used to access the rows of the table. The numbers in the Inputs and Outputs columns refer to the Index values in the other table. 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 2-9. Circuit Tables.

The first step in the simulation process is to compute level numbers for each gate, and sort the gate table into ascending sequence by level number. Figure 2-10 illustrates this process. Note that the Index values in the sorted table 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 2-10. The Levelization Step.

Just as there are many implementation strategies for the underlying data structures, there are many techniques for representing simulation inputs. Regardless of the

implementation strategy, it is necessary to be able to read successive inputs vectors and copy their values into internal data structures where they are accessible to the simulation routines. The general simulation procedure is given in Algorithm 2-2. Figure 2-11 illustrates the results of reading a single input vector. For clarity, the nets that are not affected by this process are omitted from the illustration.

```
Simulate()
{
   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. Simulation Algorithm.

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 2-11. Reading the Input Vector.

To simulate the input vector of Figure 2-11, it is necessary to simulate each gate in the sorted gate table, in the order given in Figure 2-10. Algorithm 2-3 gives the general procedure for performing gate simulations. Figure 2-12 gives the resultant net values. The nets whose values are not affected by the simulation are omitted from this illustration. The remaining nets are listed in the order in which their values are computed.

```
SimulateGate(Gate)
{
   Boolean Result;

   Case Gate.Type of
   And:
      Result = Gate.Inputs[1].Value;
      For I = 2 to Gate.InputCount do
         Result = Result And Gate.Inputs[I].Value;
      EndFor
      Gate.Outputs[1].Value = Result;
   Or:
      ...
   EndCase;
}
```

Algorithm 2-3. Gate Simulation.

Net Table				
Index	*Name*	*Type*	*Fanout*	*Value*
6	X1		1	0
7	X2		4	1
8	X3		2	0
9	X4		2	0
5	Q	PO		1

Figure 2-12. The Simulation Step.

The overall flow of this procedure is illustrated in Figure 2-13.

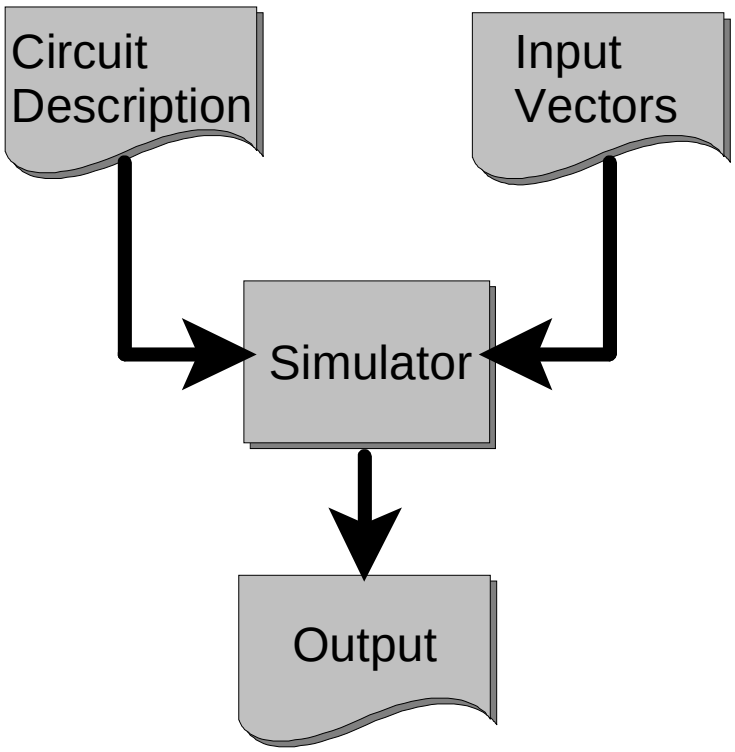


Figure 2-13. Interpreted Simulator Flow.

Although this simulation technique works well for combinational circuits, some modifications are required for sequential circuits. For synchronous circuits, the changes are minor, but more substantial modifications are required for asynchronous circuits.

2.4 Compiled Simulation

The simulation technique discussed in the previous section has several drawbacks, the most obvious of which is the number of instructions that must be executed to simulate a single gate. Suppose a circuit consists of a single AND gate. Apart from reading inputs and printing outputs, this circuit can be simulated with a single high-level statement:

A = B **And** C;

On most computers this statement can be implemented with at most three or four instructions. Some machines will require only a single instruction. Simulating the same circuit using Algorithms 2-2 and 2-3 would require substantially more instructions. To illustrate this point, let us analyze the low-level operations that must be performed. We will make no attempt to obtain a precise instruction count, because this cannot be done without specifying a particular computer. The interpretive approach would require the following steps to be performed. We analyze the execution from just after reading inputs to just before printing outputs.

Set I_1 to 1. (There are two distinct variables named I.)
Compare I1 to 1, Jump if I > 1 (No jump is performed.)
Push Address of SortedGates[1] onto stack
Call SimulateGate
Allocate Stack Space for Result
Load Gate.Type into Register X (Jump table implementation of CASE)
Multiply Register X by 4 (Assume numeric code for gate type)
Add Base address to Register X
Jump Indirect using Register X
Set Result to Gate.Inputs[1].Value
Set I_2 to 2
Compare I_2 to 2, Jump if $I_2 > 2$
And Result with Gate.Inputs[2].Value (Result receives output)
Increment I_2
Compare I_2 to 2, Jump if $I_2 \leq 2$ (No Jump)
Store Result into Gate.Outputs[1].Value
Clean up Stack
Return
Increment I1
Compare I_1 to 1, Jump if I_1 <= 1 (No Jump)

In a circuit containing many gates, a sequence of operations similar to the above would be executed for each gate in the circuit. Of all the operations in this sequence only three are directly concerned with simulating the gate. Clearly, there is room for substantial optimization in this algorithm. The primary difficulting in achieving such an optimization is the general purpose nature of the interpretive algorithm. Since every circuit has a different number of gates, the algorithm must provide a loop that can execute an arbitrary number of times. Since it is impossible to predict what types of gates will appear in various places in the circuit, it is necessary to decode the gate type every time a gate simulation is done. Since it is not possible to predict the number of inputs that a gate may have, it is necessary to provide a loop that iterates through all the inputs that are actually present.

If these constraints were not present we could construct a much more efficient simulator for the circuit. In particular, the Algorithm 2-4 could be used in place of Algorithm 2-2 and Algorithm 2-3.

```
Simulate()
{
   Boolean A,B,C;

   While there are more input vectors do
      Read B,C;
      A = B And C;
      Print A;
   EndWhile
}
```

Algorithm 2-4. Simulation Algorithm.

Compiled simulation, particularly Levelized Compiled Code (LCC) simulation, addresses the problem of simulation efficiency by generating circuit-specific simulation code. The general flow of this technique is illustrated in Figure 2-14.

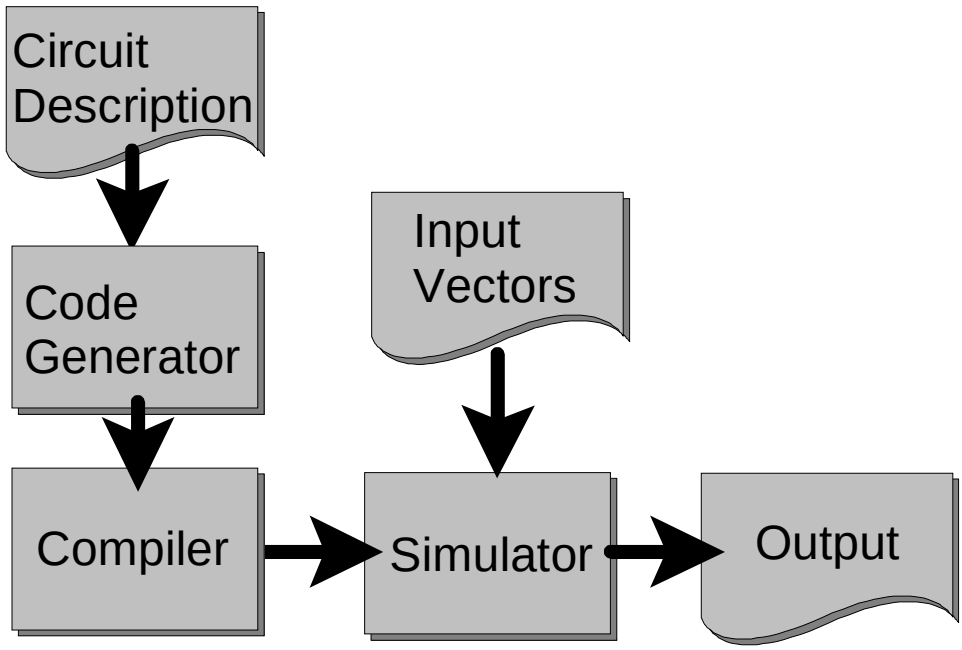


Figure 2-14. Compiled Simulation Flow.

As Figure 2-14 illustrates, the overall flow of a compiled simulator is quite different from that of interpreted simulation. Once the simulator has been created, it acts as a independent program that can be used to simulate a specific circuit. Simulating several circuits requires the creation of several simulation programs.

A major problem in designing a compiled simulator is choosing an output language for the code generator, and selecting a compiler to create the executable files. If a standard language such as C or PASCAL is used, any users of the simulator must have a suitable compiler installed on their systems. If the simulator is intended to be used with a Graphical User Interface (GUI) such as MS Windows, or MacIntosh system 7, the

compilation process must integrate the necessary GUI components with the output of the code generator. For maximum efficiency, the code generator should generate assembly language, but this may limit the portability of the simulator for operating systems such as UNIX, which are not dedicated to a specific type of hardware.

To illustrate the process of compiled simulation, let us start with the circuit pictured in Figure 2-8. The first steps in the compiled simulation process are identical to those of interpreted simulation, namely producing tables similar to those shown in Figure 2-9 and levelizing the circuit as shown in Figure 2-10. The next step is to generate a set of variables to hold net values. These variables are normally static or global variables. The next step is to process the sorted gate table in ascending order, and generating simulation statements for each gate. The results of this process are illustrated in Figure 2-15.

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

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

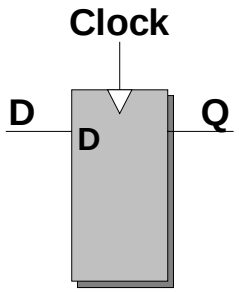
Figure 2-15. Circuit Simulation Code.

The subroutine illustrated in Figure 2-15, must be called by a main program that reads input vectors and prints outputs. In many cases, the input and output routines are also custom-generated routines. Regardless, the main routine will be similar to that presented in Algorithm 2-4.

2.5 Sequential Circuits

As presented, neither the interpretive nor compiled techniques can be used with sequential circuits. The levelization procedure will fail for any circuit containing a logic loop. It is possible to alter the levelization procedure as described in section 2.2.2, but these changes require additional changes in the simulation procedure. The procedure is simpler for synchronous circuits, so let us begin there. We will detail the required changes for the compiled code simulation technique. The changes for the interpreted technque are similar, and left as an exercise.

For simplicity, assume that the circuit designer has represented a master-slave pair as a single gate. As illustrated in Figure 2-6, the first step is to break the circuit into two parts, one containing the synchronous flip-flops, and one containing the remainder of the circuit. The remainder of the circuit is levelized, and code is generated in the usual manner. (If levelization fails, the circuit is not synchronous, and must be redesigned.) The simulation code for the synchronous flip-flops is generated in a straightforward manner, as illustrated in Figure 2-16, and placed after the simulation code for the remainder of the circuit. Figure 2-17 shows the generated code for the circuit of Figure 2-6.




```
If Clock = 1 Then
   Q = D;
Endif
```

Figure 2-16. Generated Code for Flip-Flops.

```
SimulateCircuit()
{
   X1 = Not A;
   X2 = Not S;
   X3 = X1 And B;
   X4 = A And T;
   X5 = A And X2;
   X6 = X1 And T;
   S1 =  X3 Or X4;
   S2 = X5 Or X6;
   If Clock = 1 Then
      S = S1;
   Endif
   If Clock = 1 Then
      T = T1;
   Endif
}
```

Figure 2-17. Syncronous Simulation.

As long as circuit inputs are correctly constructed, the simulation will behave as if the flip-flop were a master-slave pair. Input vectors must appear in pairs. The first must set the clock to zero, and the second must set the clock to one. All other inputs must have identical values in both vectors. In a master-slave pair, the output of the pair changes only when the clock is zero. These changes, along with changes in the primary inputs, will cause changes in the outputs of the circuit. When the clock is one, the internal connection between the two flip-flops will change, but the output of the pair will remain constant. By placing the simulation code for the flip-flops after the simulation code for the remainder of the circuit any changes in the flip-flop output are delayed until the next input vector is processed. Since the output changes when the clock is one, this implies that the effects of the change will not be propagated to the remainder of the circuit until the clock becomes zero.

There are two alternatives to the procedure illustrated in Figure 2-17. The first is to model the flip-flop more accurately by using an internal variable to represent the first half of the master slave pair. Figure 2-18 illustrates how this is done. When this procedure is used, it is necessary to place the flip-flop simulation code ahead of the code for the rest of the circuit.

```
If Clock = 1 Then
   DTemp = D;
Else
   Q = DTemp;
Endif
```

Figure 2-18. Master-Slave Simulation.

The second alternative is based on the idea presented in Figure 2-18, but strives for greater efficiency by simulating only the parts of the circuit that need to be simulated for each input. When the clock is equal to 1, the master portion of each flip flop is simulated, and when the clock is equal to zero, the slave portion of each flip-flop and the remainder of the circuit is simulated. To illustrate this technique, Figure 2-19 shows the code that would be generated for the circuit of Figure 2-6.

```
SimulateCircuit()
{
   If Clock = 1 Then
      STemp = S1;
      TTemp = T1;
   Else
      S = STemp;
      T = TTemp;
      X1 = Not A;
      X2 = Not S;
      X3 = X1 And B;
      X4 = A And T;
      X5 = A And X2;
      X6 = X1 And T;
      S1 = X3 Or X4;
      S2 = X5 Or X6;
   Endif
}
```

Figure 2-19. Syncronous Simulation.

Asynchronous circuits require different simulation technques than synchronous circuits. Asynchronous simulation technques are also useful to verify that a synchronous circuit does indeed function in a synchronous manner. Because synchronous simulation techniques assume that a circuit is synchronous, they generally cannot uncover problems of this nature. As with synchronous circuits, we will assume that compiled code technques are being used, with the interpreted technques left as an exercise.

The first step in simulating an asynchronous circuit is to levelize it as described in Section 2.2.2. During the levelization process, a record must be made of the feedback arcs. The simulation code is generated as usual, but must be embedded in a loop that tests the value of the feedback arcs for changes. The simulation terminates when no feedback arc changes values from one iteration to the next. To prevent oscillations, an absolute limit is usually placed on the number of iterations. If the circuit is designed properly the circuit should stablize after two or three iterations, so the absolute limit can be set to a relatively low number. The theoretical limit is 2^n+1 iterations, where n is the number of feedback arcs, but in practice an absolute limit of 10-20 or so will be sufficient. Figure 2-20 shows the code that would be generated for the circuit of Figure 2-7.

```
SimulateCircuit()
{
   Count = 0;
   OldQbar = Not Qbar;
   While Qbar ≠ OldQbar And Count < 10 do
      Q = Not (S And Qbar);
      Qbar = Not (R And Q);
   EndWhile
}
```

Figure 2-20. Asynchronous Simulation.

Although these techniques permit the simulation of asychronous flip-flops and circuits, the stablization properties of a circuit depend heavily on timing. Since the technques presented in this chapter ignore timing issues, they are not suitable for the simulation of asynchronous circuits in all cases. See Chapter 3 for a more in-depth discussion of this issue.

2.6 Vector Packing

One consequence of simulating combinational circuits in levelized order is that there are no sequential dependencies between input vectors. (This statement is obviously untrue for *sequential* circuits.) This allows input vectors to be simulated in any sequence, as long as the correct output sequence is presented to the user. Specifically, it allows several input vectors to be simulated simultaneously. In the preceding discussion, it was more or less assumed that Boolean variables would be used to represent net values. However, at the hardware level, there is no such thing as a Boolean variable. Boolean values are represented using, at least, 8-bit quantities. Figure 2-21 illustrates how such quantities would be used in practice.

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 2-21. Eight-Bit Boolean Operations.

It is obvious from Figure 2-21 that the seven high-order bits are unused. This seems especially wasteful, since the instructions used to perform the boolean operations are bit-level operators that act on all eight bits simultaneously. Since all input vectors are independent of one another, it is possible to accumulate eight input vectors, and simulate them 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. Figure 2-22 illustrates how a three-input circuit can be simulated using all eight input combinations simultaneously.

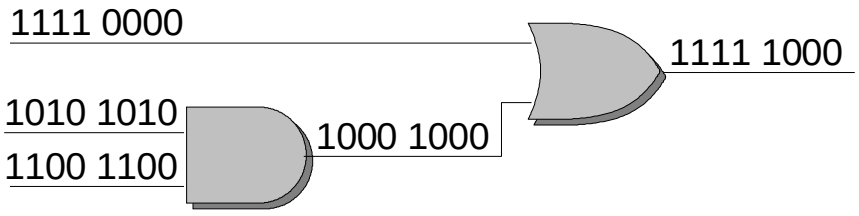


Figure 2-22. Bit-Parallel Simulation.

The technique illustrated in Figure 2-22 is called bit-parallel simulation. Placing several input vectors into a single variable is known as vector packing. The simulation code that is required for bit parallel simulation is identical to that used for single vectors. The only required 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, but it is possible to compete several tests simultaneously. Figure 2-23 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 2-23. Packing Independent Tests.

In the packed tests of Figure 2-23, 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. Although the examples in this section show groups of four and eight bits, it is customary to use full 32-bit words for bit-parallel simulation. In many cases, operations on full words are slightly faster than 8 or 16-bit operations. For reasonably large circuits, the time required to pack and unpack vectors will be small, so doubling the number of bits doubles the speed of simulation.

2.7 Timing and Logic Models

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 signals of a circuit. For combinational circuits, delays cannot affect the final output, but there may be several intermediate transisional states before the outputs settle and 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 achieve its final state. (The process of achieving the final state is called settling.) In contrast, gate delays can have a profound affect on the outputs of an asynchronous sequential circuit. Circuits that stablize when delays are ignored can oscillate when delays are taken into account. A circuit may also cease to function correctly when delays are taken into account.

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. More formally, levelized simulation is said to implement the Zero-Delay model of simulation, implying that the delay of each gate is zero. To see why this is so, consider the circuit of Figure 2-24. 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 of time during which both inputs of G2 are zero. This will cause the output Q to briefly become zero before settling to one. Figure 2-25 shows the states of all five signals, assuming a constant, non-zero delay for both gates.

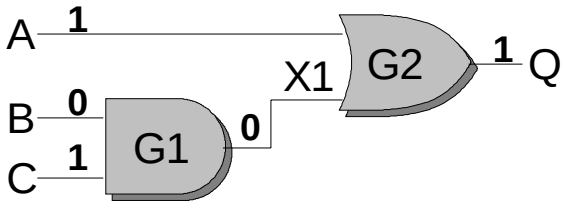


Figure 2-24. An Example with Delays.

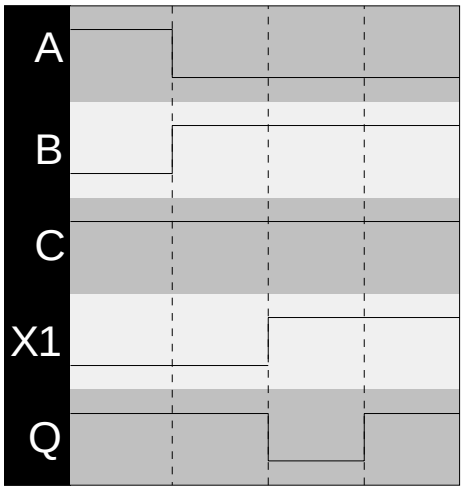


Figure 2-25. Net changes with delays.

Chapters 3 and 4 address timing issues, while Chapters 5 and 6 address the issue of timing in levelized simulation.

In all examples in this chaper we have modeled net values as zero and one. In an actual circuit, these values are represented as low and high voltages. In a real circuit the voltage on a net does not change instantaneously, but moves continuously between the low and high values. Furthermore, the exact voltages used to represent one and zero are not fixed, but distributed over a range of values. Figure 2-26 illustrates the voltage ranges used to represent binary values.

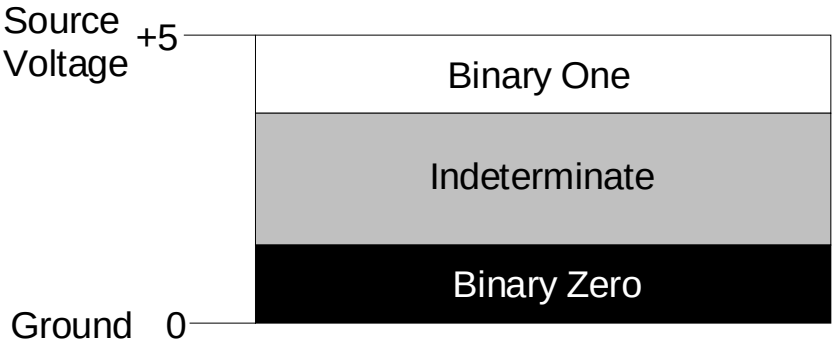


Figure 2-26. Representing values with voltages.

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. For some circuits, it may be necessary to allow for the possibility of a net being in the indeterminate state. This can be done by introducing a third logical value U, or unknown. There are other cases where having an unknown state would be useful. During the simulation of the first input vector, most nets have never been assigned a value of either zero or one. For levelized simulation, it is generally acceptable to assign a default value of zero to these nets, but for the more elaborate algorithms discussed in later chapters, it will be necessary to assign the Unknown value to uninitialized nets.

Another situation in which the unknown value is useful is when a circuit is determined to be in oscillation. To illustrate, consider the circuit pictured in Figure 2-5. If s and r are both set to zero, and then simultaneously set to 1, the outputs will oscillate between one and zero. (Using the simulation code of Figure 2-20, the circuit will stablize, but other algorithms will correctly simulate the oscillation.) Eventually, the circuit will stablize in some state, but the final state depends on slight differences in the delays of the two Nand gates. At the logic simulation level, the best that can be done is to detect the oscillation, and set both outputs to the Unknown value.

2.8 Summary

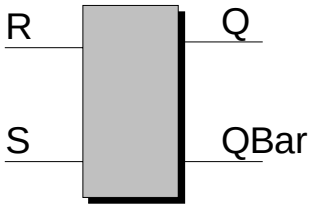
For the most part, the techniques for simulating individual gates are simple. The primary difficulty in logic simulation is not the simulation of individual gates, but the scheduling of gate simulations. The scheduling technique presented in this chapter is static, in the sense that all scheduling decisions are made before any simulation begins. The levelization technique can be used for both interpreted and compiled simulation, and can be used for both 2-valued and 3-valued simulation.

Although the levelization technique is applicable only to combinational circuits, adaptations can be made to handle both synchronous and asynchronous sequential circuits. The techniques for asynchronous circuits are less popular, because they ignore gate delays. Relative gate delays can have a profound effect on the correctness and performance of a asynchronous sequential circuits.

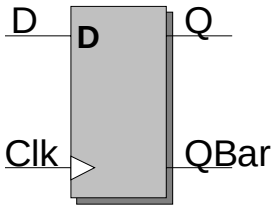
The levelization technique is an adaptation of the topological sort of a directed acyclic graph. As in levelization, topological sort is most often used for scheduling tasks or other items that can be represented as a directed acyclic graph. See Knuth for more information.

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 transision.

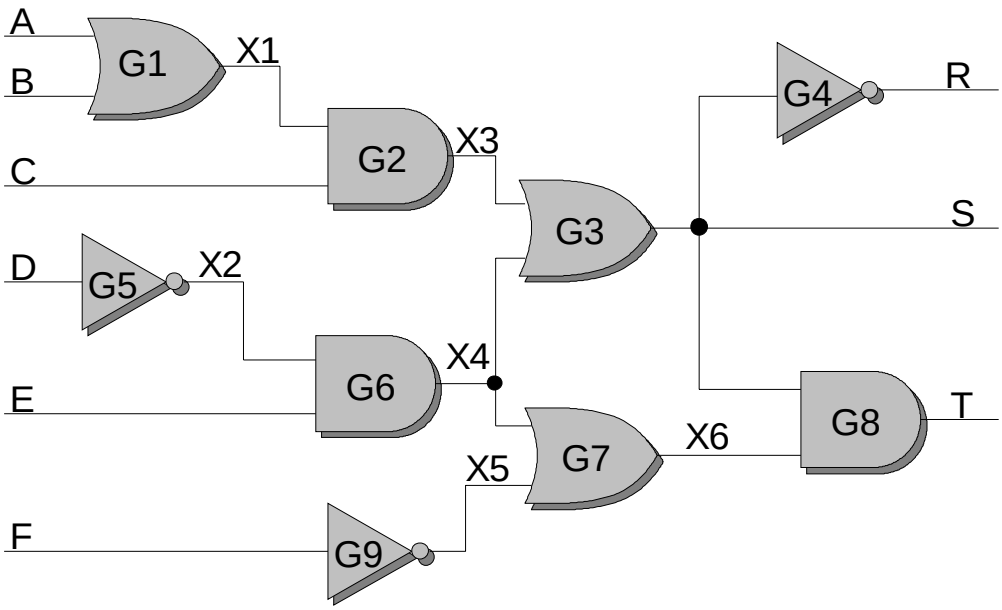


2. Using your favorite programming language, write the simulation code for the following D flip-flop.

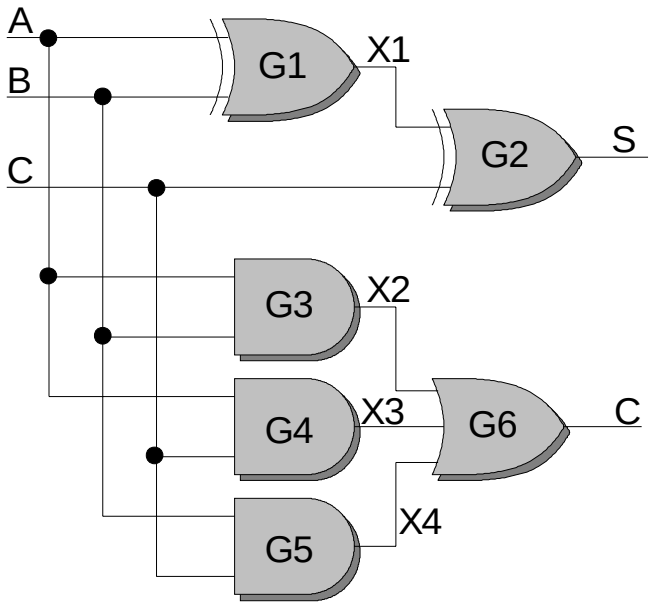


3. Write the simulation code for the Nand, Nor, Xor, and Xnor gates. (The Xnor gate is an Xor with an inverted output.)

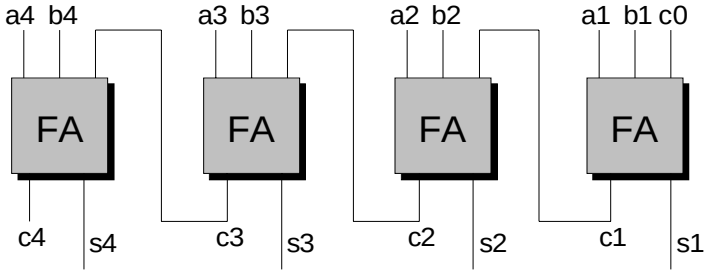
4. Levelize the following circuit. Give level numbers for each net and each gate.



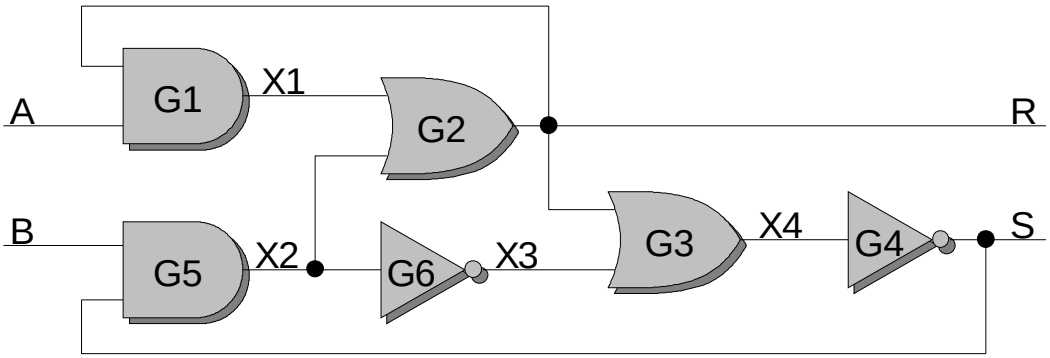
5. The following circuit is known as a full adder, because it adds three bits and produces sum and carry outputs. Levelize the full adder, giving level numbers for each gate and each net.



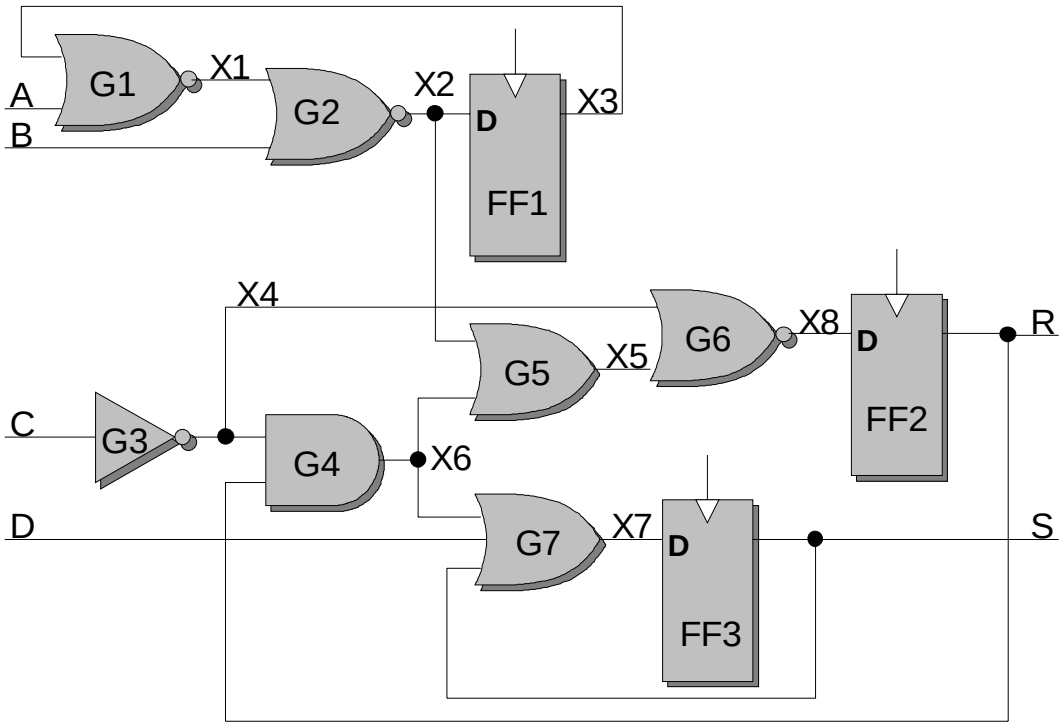
6. The full adder of Exercise 5 can be used to create a four-bit adder as illustrated below. Each of the blocks in this diagram represents a full-adder circuit. Expand this circuit converting each block to a full adder, and levelize the circuit. Give each net and each gate a unique name. Give level numbers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



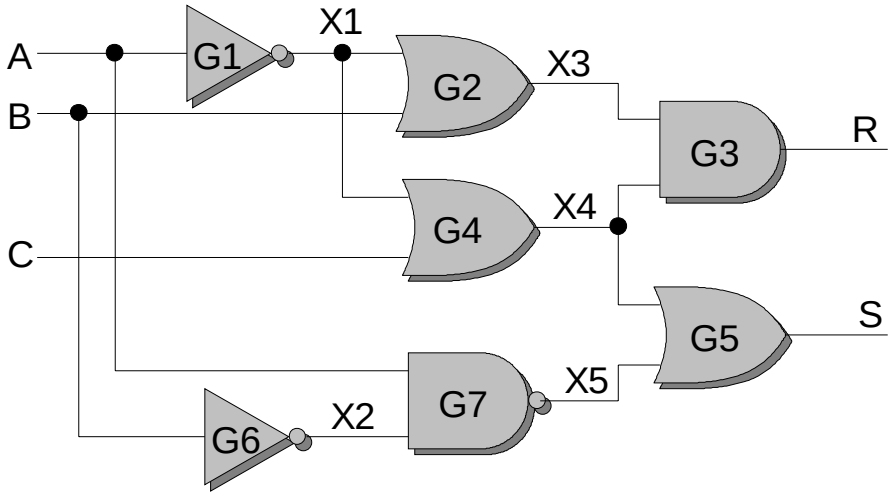
7. Levelize the following circuit. Give level numbers for each net and each gate. Identify all feedback arcs.



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



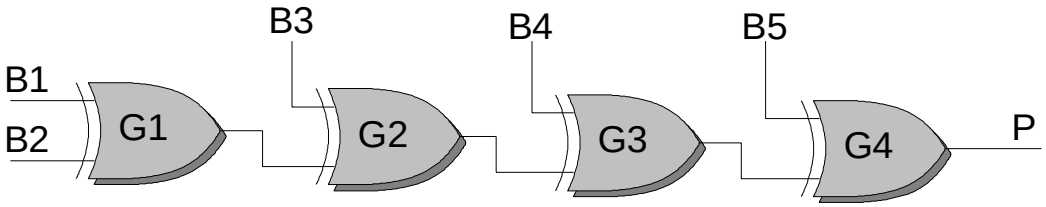
9. Create the gate and net tables for the following circuit. Initialize all nets to zero.



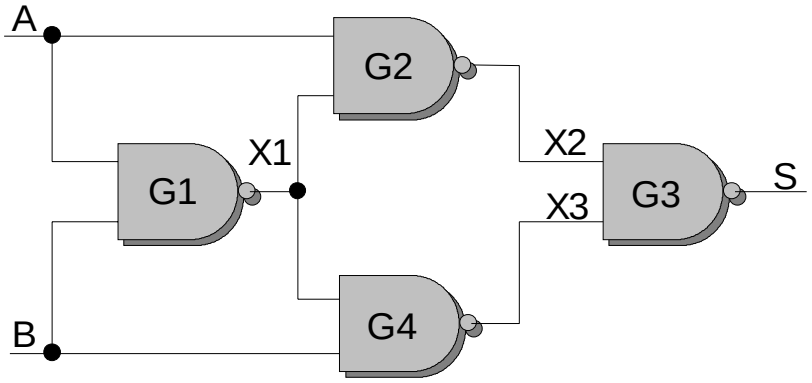
10. Levelize the circuit of Exercise 9. Show the sorted gate and net tables.

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.

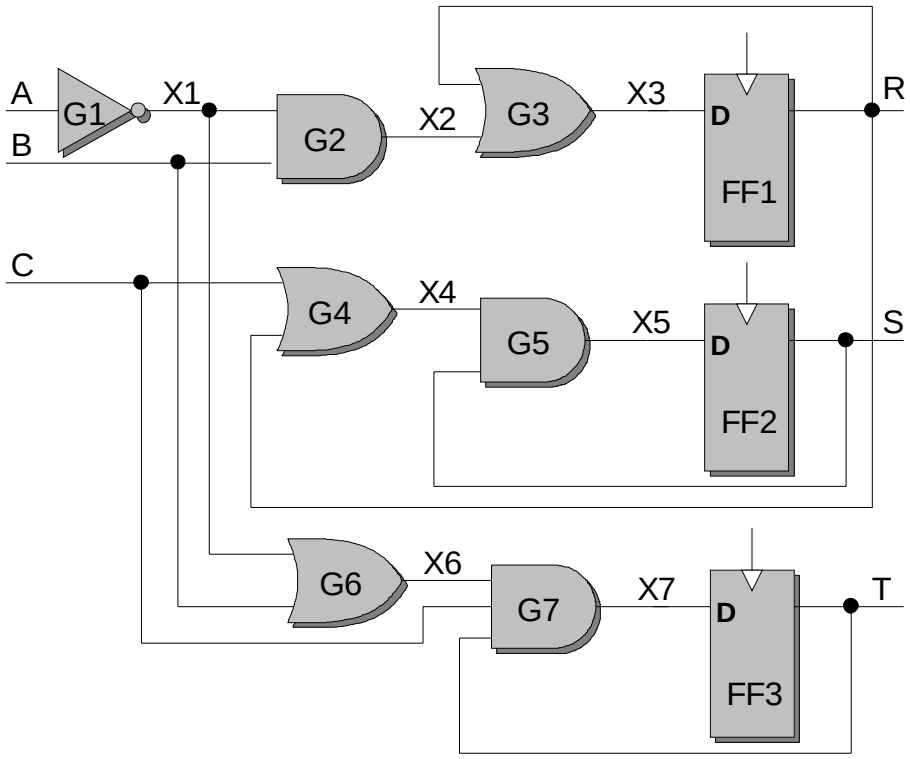


13. Using the interpreted levelized simulation technique, simulate the following circuit using inputs (A,B)=(0,0), (0,1), (1,0), (1,1).



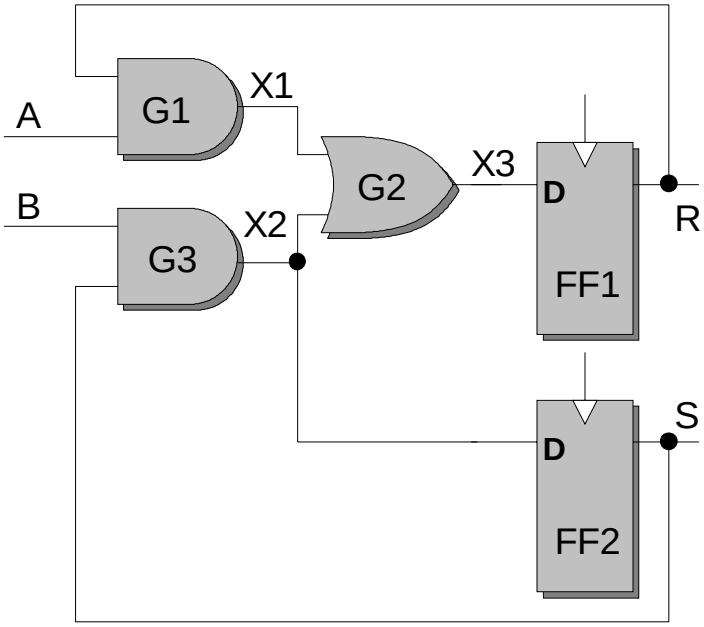
14. Using your favorite programming language, write the compiled simulation routine for the circuit of Exercise 9.
15. Using your favorite programming language, write the compiled simulation routine for the circuit of Exercise 12.
16. Using your favorite programming language, write the compiled simulation 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.

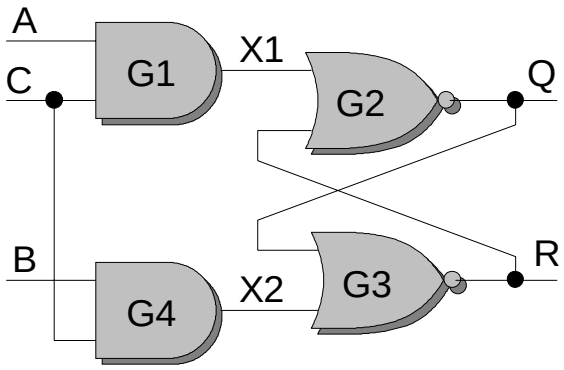


19. Simulate the circuit of Exercise 18 using the inputs (A,B,C)=(1,0,0), (0,0,0), (1,1,1), and (0,1,1).

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

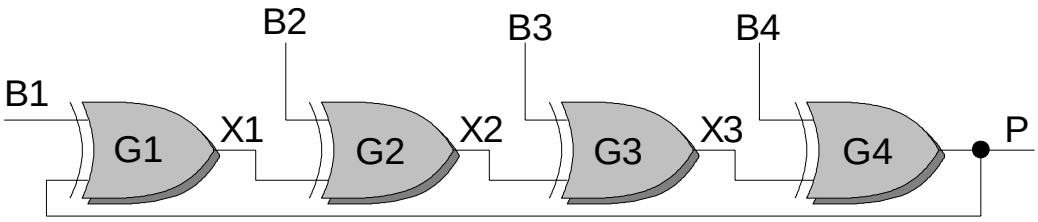


21. Give the net and gate tables for the following circuit. Levelize the circuit, and initialize all nets to zero.



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.

A.