

Chapter 5

The PC-Set Method

5.1 Static Scheduling

The material of Chapter 2 makes it seem that there is a natural unbreakable association between levelization, static scheduling, and the zero-delay timing model. However, this chapter will dispel that notion by showing that the zero delay timing-model is a result of the code-generation methodology, and not an inherent property of levelization.

In this chapter we will use levelization and static scheduling to create unit-delay and multi-delay simulations. The basic technique is called the Potential-Change-Set or PC-Set method. The PC-Set method is a generalization of the levelization techniques presented in Chapter 2. Like the levelization algorithms, the PC-Set algorithms can be used for a number of different purposes beyond those discussed in this chapter.

5.2 Theoretical Foundations

The correctness of the unit-delay simulation depends on performing gate simulations at certain specific instants of simulated time. It is the change in net values at specific moments of time that gives rise to both the unit-delay and multi-delay timing models. Because there is a delay in propagating signals to gates, and because signals can propagate along paths of differing delay, it is often necessary to simulate the same gate several times during the simulation of a single input vector.

The objective of any simulation is not to simulate gates in a certain order, or to schedule simulations in a certain fashion. The objective is to produce the correct sequence of output values. Any simulation technique that gives us the correct sequence of outputs is, by definition, correct, no matter how bizarre it might be. In the PC-Set method we will use levelization and static scheduling to perform the same simulations that would

have been performed by an event-driven simulation. Because static scheduling is used, it is necessary to perform any simulation that *could be* performed during a unit delay simulation, not just the ones that actually are performed. We note that there is a close relationship between leveled simulation and unit-delay simulation, since if all duplicate gate simulations are eliminated from a unit-delay simulation the remaining gate simulations will occur in leveled order. Thus the levelization procedure used for zero-delay simulation is essentially correct, as long as the additional simulations for the more complex timing model are also performed.

The first requirement for creating a statically-scheduled unit or multi-delay simulation is to determine the additional simulations that are required by the timing model. This is done by determining the simulated times at which events can potentially occur for the various nets in the circuit. The objective is to compute a set of Potential Change times for each net in the circuit, which is referred to as the PC-Set of the net.

5.3 Computing PC Sets

To simplify the discussion, assume that the objective is to create a unit-delay simulation, so all gate delays are equal to one. If the input of a gate can change at time t , then the outputs can change at time $t+1$. Furthermore, if all inputs of a gate remain constant at time t , then the outputs must remain constant at time $t+1$. As with levelization, the PC-Set computation begins at the primary inputs. A primary input may change at time zero, but will be constant thereafter. Thus, each primary input is assigned a PC-Set of $\{0\}$. The PC-Set of a gate and its output is computed by forming the union of the PC-Sets of the inputs, and adding one to each element. Figure 5-1 illustrates the procedure for computing PC-Sets.

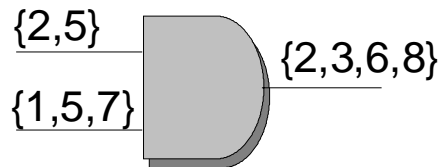


Figure 5-1. The PC-Set Computation.

If a net is driven by a single gate, then the PC-Sets of the net and its driving gate are identical. If a net is driven by two or more gates, then the PC-Set of the net is the union of the PC-Sets of the driving gates. This situation is illustrated in Figure 5-2.

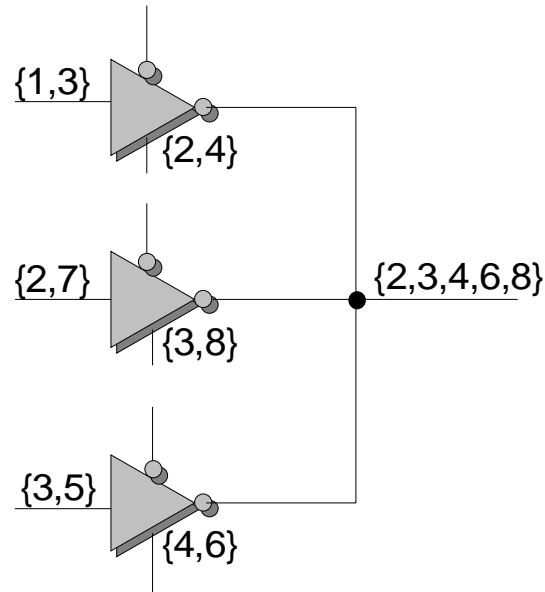


Figure 5-2. Gate PC-Sets and Wired Connections.

Algorithm 5-1 through Algorithm 5-4 show the algorithm for computing PC-Sets. Algorithm 5-1 shows the processing of the primary inputs and constant signals, Algorithm 5-2 shows the processing of nets, Algorithm 5-3 shows the processing of gates, and Algorithm 5-4 shows the overall flow of the algorithm.

```

For Each Primary Input or Constant Signal N Do
  N.PCSet = {0};
  For Each Gate G in the Fanout of N Do
    If all input nets of G have PC-Sets Then
      Place G on Ready Queue;
    End If
  End For
End For

```

Algorithm 5-1. Processing Primary Inputs and Constants.

```

S := Union of the PC-Sets of the Driving Gates of N;
N.PCSet := S;
For Each Gate G in the Fanout of N Do
  If all input nets of G have PC-Sets Then
    Place G on Ready Queue;
  End If
End For

```

Algorithm 5-2. Processing Nets.

```

S := Union of the PC-Sets of the Inputs of G;
Add 1 to Each Element of S;
G.PCSet:= S;
For Each Output Net N of G Do
  If all driving gates of N have PC-Sets Then
    Place N on Ready Queue;
  End If
End For

```

Algorithm 5-3. Processing Gates.

```

ProcessPrimaryInputs( );
While Ready Queue is Not Empty
  If Top of Queue is a Net Then
    N := Pop Queue;
    ProcessNet(N);
  Else /* Top of Queue is a Gate */
    G := Pop Queue;
    ProcessGate(N);
  End If
End While

```

Algorithm 5-4. Unit-Delay PC-Set Computation.

Figure 5-3 illustrates the computation of PC-Sets for a circuit.

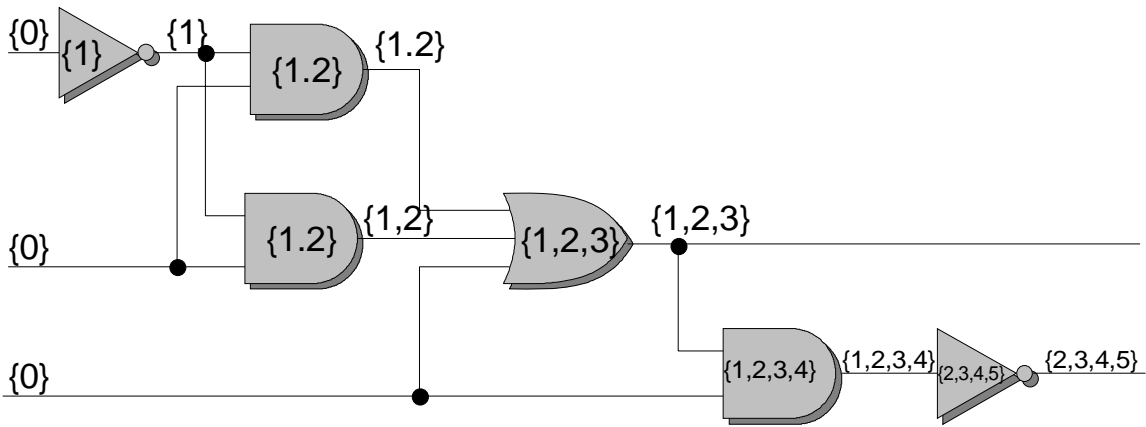


Figure 5-3. PC-Sets for an Entire Circuit.

5.4 Zero Insertion

To guarantee that each gate simulation processes the proper input values, it is necessary to make one additional modification to the PC-Sets generated by Algorithm 5-4. To see why this is necessary, consider the diagram given in Figure 5-4.

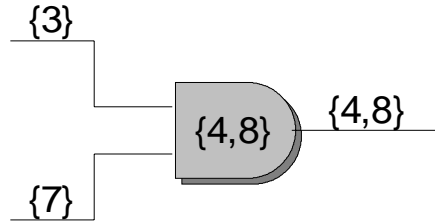


Figure 5-4. The Need for Zero-Insertion.

Because the gate of Figure 5-4 has a PC-Set element of 4, it will be necessary to simulate the gate at time 4. This must be done using the values of both input nets at time 3. However, the new value of the lower input will not be computed until time 7. It is therefore necessary to use the value of the net from the *previous input vector* for the time 4 simulation. Zero Insertion tags those nets whose values must be retained from the previous input vector, by inserting the element “0” into the PC-Set of the net. This process is illustrated in Figure 5-5.

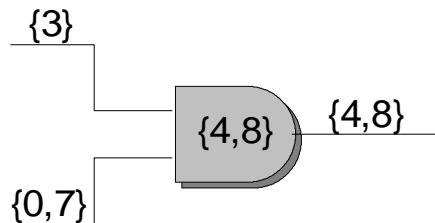


Figure 5-5. A Net After Zero-Insertion.

The PC-Set elements created by zero-insertion are not propagated through the network. A net will require zero insertion if it is the input to gate, G , with two or more inputs, and if the smallest element in its PC-Set is larger than the smallest element of the PC-Set of some other input of G . Algorithm 5-5 gives the procedure for zero insertion. Note that zero insertion does not affect the PC-Sets of gates.

```

For each multiple input gate  $G$  Do,
   $S :=$  The union of the PC-Sets of the Inputs of  $G$ ;
   $M :=$  Minimum( $S$ );
  For Each Input  $N$  of  $G$  Do
     $T := N$ .PCSet;
     $Q :=$  Minimum( $T$ );
    If  $M < Q$  Then
       $T := T$  Union  $\{0\}$ ;
       $N$ .PCSet  $:= T$ ;
    End If
  End For
End For

```

Algorithm 5-5. The Zero-Insertion Procedure.

5.5 Code Generation

The first step in code generation is to create a variable for each distinct net value. A set of variables is generated for each net, one for each element of the PC-Set of the net, including any elements created through zero-insertion. Figure 5-6 illustrates this process. Generated variable names are of the form N_k, where N is the name of the net and k is the PC-Set element.

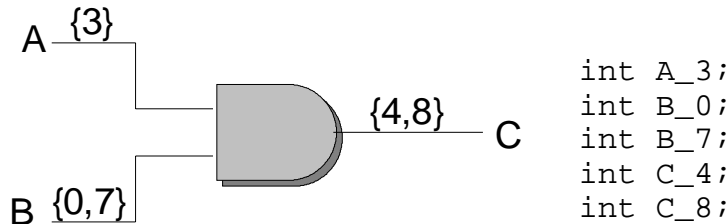
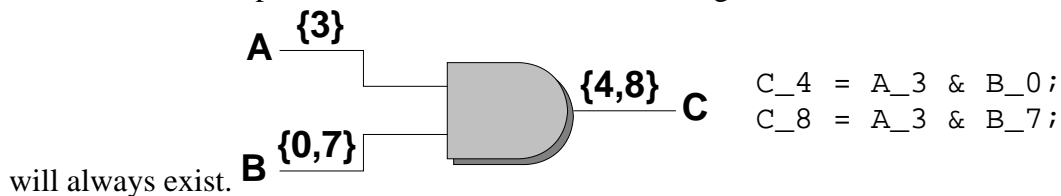


Figure 5-6. Generating Variables.

The Gate PC-Sets are used to generate the executable code. Gates are processed in leveled order, and one gate-simulation is generated for each element of the gate's PC-Set.

Each gate calculates a new value for one output-net PC-Set Element. Each simulation also references one operand value for each input net. For each input net, the proper input value is selected by choosing the largest PC-Set element which is strictly smaller than the output PC-Set element. Zero-Insertion guarantees that such an element



will always exist.

Figure 5-7 illustrates the process of code generation.

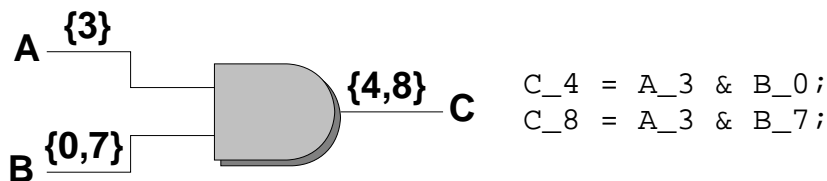


Figure 5-7. Code Generation.

Algorithm 5-6 gives the procedure for code generation. For simplicity, this algorithm assumes that all gates are two-input AND gates. Code generation for other types of gates is similar.

```

For each gate G in leveled order Do
  For Each element t of G.PCSet Do
    S1 := G.InputOne.PCSet;
    S2 := G.InputTwo.PCSet;
    S1A := All Elements of S1 Less than t;
    S2A := All Elements of S2 Less than t;
    u1 := Maximum(S1A);
    u2 := Maximum(S2A);
    C := G.Name;
    A := G.InputOne.Name;
    B := G.InputOne.Name;
    Output C,"_",t,"=",A,"_",u1,"&",B,"_",u2,"";
  End For
End For

```

Algorithm 5-6. Code Generation.

The code generated by Algorithm 5-6 sets the value of all variables, except those generated due to zero-insertion. These are variables of the form N_0, where N is the net name and 0 is the inserted PC-Set element. The value of these variables is set using statements of the following form, where L is the maximum PC-Set value for net N. These statements are initialization code and must precede the code generated by Algorithm 5-6.

$$N_0 = N_L;$$

5.6 Printing Output Vectors

Printing Output vectors is no longer straightforward, because there is no longer a single value for each output net. Furthermore, because all values for a particular gate are calculated at one time, the simulator no longer has recognizable timing points that correspond to instants of simulated time. One method of producing output is to simply print the value of all monitored nets after the simulation of an input vector is complete. However, this output would be difficult to read, even if the values are labeled with their PC-Set values.

To produce a set of conventional output vectors with the PC-Set method, it is necessary to coordinate the different values that are available for each net. The most straightforward method for doing this is to treat the vector print routine as if it were a gate whose inputs are the monitored nets of the circuits. Figure 5-8. The Vector Print Routine. illustrates this concept.

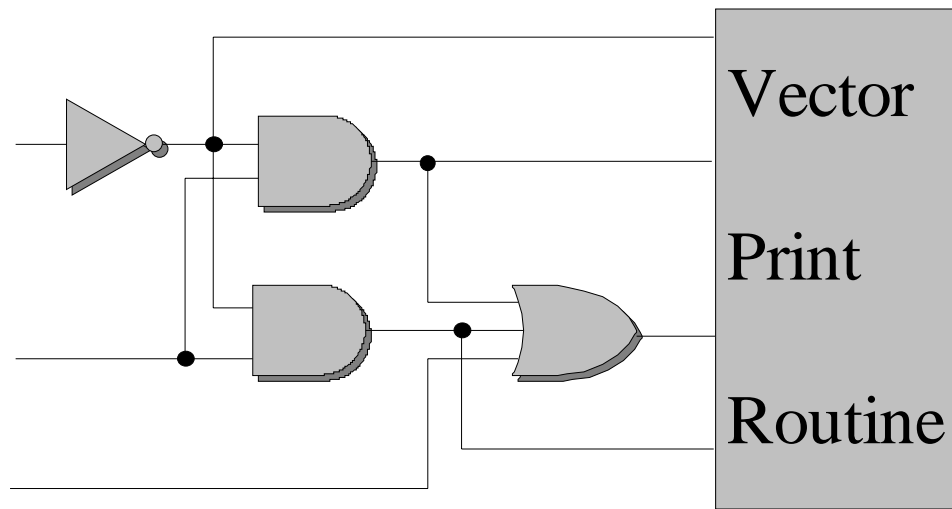


Figure 5-8. The Vector Print Routine.

Since the Vector Print Routine is being treated as a special gate, the inputs to the routine are subject to zero insertion. The Vector Print Routine must be “simulated” once for each distinct level in the circuit. Each of these simulations prints one output vector. The input values to each simulation are selected using a procedure similar to that of Algorithm 5-6.

5.7 Multi-Delay PC-Sets

The PC-Set Method, with minor modifications, can also be used to perform multi-delay simulations. The only real change that is necessary is replacing the shaded statement in Algorithm 5-4 with “Add G.Delay to Each Element of S;” A similar modification must be made to the levelization procedure. The PC-Sets are computed with respect to delays, rather than gate-counts, and the circuit is levelized with respect to delays. In all other respects, the Unit-Delay and Multi-Delay algorithms are identical.

The main disadvantage of the Multi-Delay PC-Set Method is the size of the PC-Sets that can be generated for nets near the primary outputs. If P is the PC-Set of net N , then P contains the element d if and only if there is a path of delay d between the primary inputs and N . When all gate delays are the same, there will be many paths with the same delay, but increasing the number of delays also increases the probability that two distinct paths will have differing delays. For circuits such as array multipliers, this can yield PC-Sets with millions of elements. PC-Sets of this size can create many megabytes of generated code even for circuits of modest size.

5.8 Vector Packing

The primary advantage of oblivious simulation is the ability to pack several input vectors into a single word, and simulate them simultaneously using bit-parallel operations. As was pointed out in Chapter 2, vector packing is most effective when there are no sequential dependencies between input vectors. However, in unit and multi-delay simulation, there are usually some sequential dependencies between each pair of

consecutive vectors, even if the circuit is combinational. The detection of hazards depends on combining values from two consecutive input vectors. To illustrate the problem suppose a circuit is to be simulated with sixteen single-bit input vectors, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, and P. For simplicity, assume that each word contains four bits. Conventional vector packing would produce the results illustrated in Figure 5-9.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figure 5-9. Conventional Vector Packing.

In Chapter 2, it was noted that vector packing is possible, even in the presence of dependencies, as long as independent vector sets are packed vertically into each bit. For sequential circuits, this technique is generally the only effective method of vector packing, regardless of the timing model used. For combinational circuits, it is possible to preserve most dependencies in a single vector set by packing the vectors vertically as illustrated in Figure 5-10.

A	E	I	M
B	F	J	N
C	G	K	O
D	H	L	P

Figure 5-10. Vertical Vector Packing.

Vertical packing preserves most dependencies, but the sequential dependencies between the pairs (D,E), (H,I), and (L,M) are lost. These dependencies can be preserved by adding an extra vector at the beginning, as illustrated in Figure 5-11.

x	D	H	L
A	E	I	M
B	F	J	N
C	G	K	O
D	H	L	P

Figure 5-11. Vertical Vector Packing with Correction.

The main disadvantage of vertical vector packing is that all vectors must be available to the packing routine. Real-time input of vectors is not possible. Nevertheless, when input vector sets are available beforehand, vector packing can substantially improve simulation performance.

5.9 The Convergence Algorithm

Because of the levelization algorithms used to generate simulation code, the PC-Set method is applicable only to combinational circuits and synchronous sequential circuits with certain obvious modifications. The PC-Set method cannot be applied directly to asynchronous circuits such as that pictured in Figure 5-12. The levelization cannot be completed for this circuit, and the forced levelization procedures described in Chapter 2 will not yield gate simulations at the proper times. For the circuit of Figure 5-12 to be simulated properly, it is necessary to simulate gates G1 and G2 simultaneously, or at least provide the same effect as simultaneous simulation. Forced levelization will not yield such an effect.

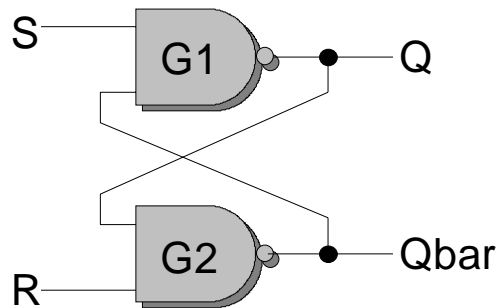


Figure 5-12. An Asynchronous Flip-Flop.

Because the circuit of Figure 5-12 can go into a permanent oscillation, the number of potential changes in nets Q and Q-Bar is infinite. Since it is impossible to deal with infinite PC-Sets, it is necessary to use a different approach for simulating asynchronous circuits.

Instead of accumulating a set of potential change times for each net, it is necessary to enumerate successive units of simulated time, and determine the gates that could potentially be simulated during each time unit. The gates that could be simulated during time slot one are those gates (and only those gates) attached to the primary inputs of a circuit.

If a net N is an output of gate G, and an input to gate H, then gate H is said to be an *immediate successor* of G, and G is said to be an *immediate predecessor* of H. The general successor and predecessor relationships are defined in the obvious way. A gate G could potentially be simulated at time $t > 1$, if and only if an immediate predecessor of G could potentially be simulated at time $t - 1$. The set of gates that could potentially be simulated at time t is known as the Gate-Set of t , and is denoted $GS(t)$. When it is necessary to distinguish between the Gate-Sets of two different circuits A and B, the notation $GS_A(t)$ and $GS_B(t)$ will be used. The immediate-successor relationship can be used to determine the Gate-Set for any time t . Algorithm 5-7 shows how Gate-Sets are generated, while Figure 5-13 shows the complete collection of Gate-Sets for the circuit of Figure 5-3.

```

For each primary input N do
  Add All Gates in the fanout of N to GS(1)
EndFor
Current := 1;
While GS(Current) is not empty do
  For each gate G of GS(Current) do
    For Each output net N of G do
      Add all gates in the fanout of N to GS(Current+1)
    EndFor
  EndFor
  Current := Current + 1;
EndWhile

```

Algorithm 5-7. The Gate-Set Algorithm for Combinational Circuits.

GS(1)	{G1,G2,G3,G4,G5}
GS(2)	{G2,G3,G4,G5,G6}
GS(3)	{G4,G5,G6}
GS(4)	{G5,G6}
GS(5)	{G6}
GS(6)	{ }

Figure 5-13. Gate Sets for a Simple Circuit.

For the circuit of Figure 5-3, all Gate-Sets after the fifth are empty, due to the combinational nature of the circuit. For any combinational circuit A, there is a number C_A such that $GS_A(t)$ is empty whenever $t > C_A$. The Gate-Sets of a combinational circuit can be used to generate static, oblivious code using Algorithm 5-11 through Algorithm 5-11. This code is identical to the code generated by the PC-Set method, with some of the statements rearranged.

```

/* Generate Standard Variables */
Generate the variables N_t for each primary input of the circuit.
for t := 1 to C_A do
  for each gate G in GS(t) do
    for each output N of GS(t) do
      Generate the variable N_t;
    endfor
  endfor
endfor

```

Algorithm 5-8. Generate the Standard Variables.

```

/* Do Zero Insertion */
for t := 1 to CA do
  for each gate G in GS(t) do
    for each input N of GS(t) do
      k := the smallest integer
           such that the variable N_k exists;
      if k >= t then
        Generate the variable N_0;
      endif
    endfor
  endfor
endfor

```

Algorithm 5-9. Generate the Variables for Zero-Insertion.

```

/* Zero-Insert Initialization Code */
for all nets N in the circuit
  if N is not a primary input and N_0 exists Then
    m := the largest integer such that the variable N_m exists;
    Generate the statement "N_0 := N_m;"
  endif
endif

```

Algorithm 5-10. Generate Zero-Insertion Code.

```

/* Simulation Code */
for t := 1 to CA do
  for each gate G in GS(t) do
    SI := Empty;
    SO := Empty;
    For each input N of G do
      k := The largest integer less than t
           such that the variable N_k exists;
      Add N_k to SI;
    endfor
    For each Output N of G do
      Add N_t to SO;
    endfor
    generate a simulation statement for G,
    using SI and SO;
  endfor
endfor

```

Algorithm 5-11. Generate Simulation Code.

For sequential circuits $GS_A(t)$ will be *non-empty* for all $t > 0$. However, since A is finite, and since $GS_A(t)$ is a subset of the gates in A , there must be a time j such that

$GS_A(j) = GS_A(t)$ for some $t < j$. It is possible to choose j so that whenever $t < j$ and $u < j$ $GS_A(t) \neq GS_A(u)$ unless $t = u$. In other words, $GS_A(j)$ is the first duplicate Gate-Set encountered. Since $GS_A(t)$ depends only on the structure of A and the previous set $GS_A(t-1)$, it is easy to see that if $GS_A(t) = GS_A(j)$, then $GS_A(t+1) = GS_A(j+1)$. Although the Gate-Set sequence is infinite, it consists of a non-repeating sequence of sets, followed by a repeating sequence of sets, as illustrated in Figure 5-14. The sequence of sets from 1 to t is known as the non-repeating sequence, while the sequence of sets from $t+1$ to j is called the repeating sequence. The final set in both sequences is the same. The algorithm for finding the repeating and non-repeating sections of the gate sequence is called the Convergence Algorithm, because it detects the convergence of the Gate-Set sequence to a repeating sequence of sets. The Convergence Algorithm is given in Algorithm 5-12.

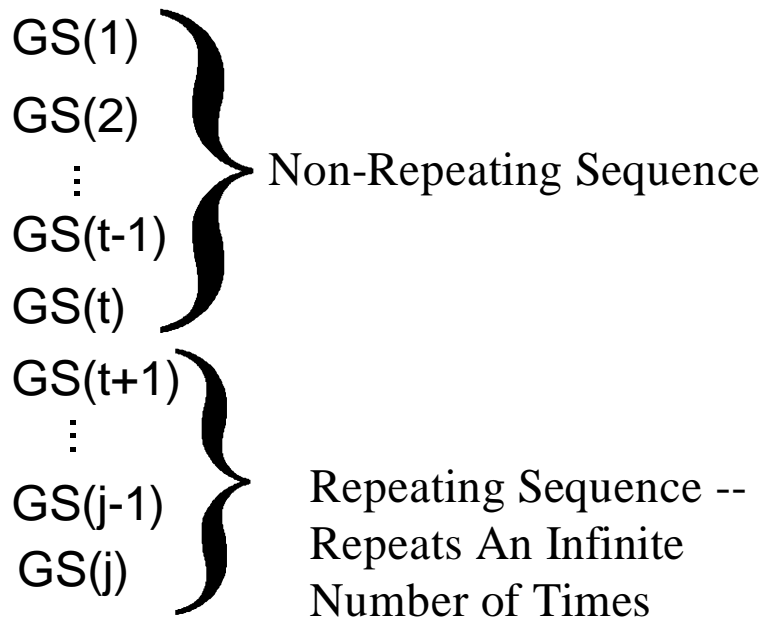


Figure 5-14. The Repeating Gate-Set Sequence.

```
For each primary input N do
  Add All Gates in the fanout of N to GS(1)
EndFor
Current := 1; AllDone := False;
While Not AllDone do
  For each gate G of GS(Current) do
    For Each output net N of G do
      Add all gates in the fanout of N to GS(Current+1)
    EndFor
  EndFor
  if GS(Current+1) is empty Then
    AllDone := True;
  Else
    For j := 1 to Current do
      if GS(Current + 1) = GS(j) Then
        Alldone := True;
      EndFor
    Current := Current + 1;
  EndWhile
```

Algorithm 5-12. Detecting Repeating Sequences.

Simulation code can be generated for an asynchronous sequential circuit by taking advantage of the repeating nature of the Gate-Set sequence. Code is generated for the non-repeating sequence just as if it were a sequence for a combinational circuit. The code for the repeating sequence is generated in the same manner, but is embedded in a repeat-until loop. The loop condition tests the outputs of the repeating sequence against those of the non-repeating sequence. Since the final sets of both sequences are the identical, the output nets are the same. If the output values of the two sections are identical, then the loop terminates. Otherwise, the outputs of the repeating sequence are copied into the outputs of the non-repeating sequence, and another iteration is performed. An absolute loop count must be maintained to protect against oscillations. Algorithm 5-13 shows the generated code for the circuit of Figure 5-12. In this circuit, all gate sets are identical and consist of the set {G1,G2}. GS(1) constitutes the non-repeating sequence and GS(2) constitutes the repeating sequence). The simulation statements of Algorithm 5-13 are the assignments to the variables Q_0, Q_2, Q_2, Qbar_0, Qbar_1, and Qbar_2. The assignments to Q_2 and Qbar_2 are embedded in a loop, and executed repeatedly until two consecutive simulations produce the same value. When the test is made, the variables Q_2, and Qbar_2 contain the most recent result, while the variables Q_1, and Qbar_1 contain the results from the previous iteration. The variable "Count" is used to guard against a permanent oscillation.

```

Q_0 := Q_2;
Qbar_0 := Qbar_2;
Q_1 := Not (Qbar_0 AND S_0);
Qbar_1 := Not (Q_0 AND R_0);
Print Intermediate Result
AllDone := False; Count := 0;
repeat
  Q_2 := Not (Qbar_1 AND S_0);
  Qbar_2 := Not (Q_1 AND R_0);
  Print Intermediate Result
  if Q_2 = Q_1 AND Qbar_2 = Qbar_1 Then
    AllDone := True;
  Else
    Count := Count + 1;
    if Count > 20 Then
      Report Oscillation
      AllDone := True;
    Else
      Q_1 := Q_2;
      Qbar_1 := Qbar_2;
    EndIf
  EndIf
until AllDone

```

Algorithm 5-13. Simulating an Asynchronous Circuit.

The convergence algorithm can be used to simulate any type of circuit and can be considered to be an extension of the PC-Set method.

5.10 Performance

Because the PC-Set Method and the Convergence Algorithm are oblivious, their performance with respect to the Event-Driven techniques of Chapter 3 depends on activity rate. When activity rates are high, the PC-Set method provides performance that is significantly superior to that of Event Driven Simulation. Experiments have shown an improvement of 4x with input activity rates as high as 50%. These experiments have given a break-even activity rate of around 3%.

For asynchronous sequential circuits, the performance of the convergence algorithm is roughly the same as that of event driven simulation. The reason for the relatively poor performance on asynchronous circuits is the pessimistic view that the convergence algorithm takes of activity rates. The sets in the gate-set sequence tend to become larger and larger until the convergence point is reached. The sets in the repeating sequence often contain most of the gates from the underlying circuit. This means that virtually the entire circuit is simulated for each iteration of the repeating sequence, sometimes more than once. The number of output nets that must be tested for changes is also large, typically most of the nets in the circuit. If the non-repeating sequence is long, the asynchronous flip-flops in the circuit may be simulated several times before the repeating

Design Automation: Logic Simulation

sequence is reached. This can lead to an enormous number of gate simulations per input vector. While it is theoretically possible for an asynchronous circuit to behave in this manner, a well designed circuit should exhibit a sharply reducing level of activity as simulation proceeds. This is the direct opposite of the behavior simulated by the convergence algorithm. Nevertheless, the convergence algorithm can provide high performance for combinational and synchronous sequential circuits. If the number of asynchronous circuits that need to be simulated is small, the convergence algorithm can be an effective tool for improving simulation performance.

5.11 Conclusion

The PC-set is an extension of the concept of levelization. If a gate G is at level n , that means that the longest path between the primary inputs and G passes through $n-1$ other gates. The PC-set gives a summary of the lengths of *all* paths between the primary inputs and the gate. In this chapter, we use the PC-set information to generate oblivious simulation code for a circuit, but we could use the PC-set for other purposes as well. (Some of these uses will be discussed in later chapters.) The PC-set simulations are considerably more efficient than conventional unit-delay simulations when the activity rate is reasonably high. For extremely low activity rates below 2-3%, conventional event driven simulation will still outperform the PC-set method.

The PC-set method can be extended to asynchronous sequential circuits by rearranging the order of the generated code, but because this code must take a worst-case view of the simulation, its performance is poor. In many cases there is no improvement over conventional unit-delay simulation.

Using the PC-set method, it is possible to embed a unit-delay circuit in a zero-delay circuit that exhibits the same behavior as the unit-delay circuit. It is necessary to replicate gates and nets to correspond to the variables and simulation statements generated by the PC-set method. The resultant circuit can be simulated using any zero-delay simulator, but will produce a unit-delay simulation. Naturally, the values from several different nets must be combined to detect hazards.

It is possible to extend the PC-set method to multi-delay simulations, but the implementation problems make this approach more academic than practical. However the PC-set method is a powerful unit-delay simulation technique that can be used both to generate simulations and to perform static analysis of a circuit.

5.12 Exercises