Chapter 6

The PC-Set Method

6.1 Static Scheduling

Chapter 2 discussed methods for static scheduling of gate simulations at compile time. The levelization procedures discussed in Chapter 2 give rise to the zero-delay timing model. The association between levelization, static scheduling, and the zero-delay timing model seems so natural, that the terms are sometimes used synononmously. However, as this chapter will show, there is no inherent association between levelization, static scheduling and zero-delay simulation. The zero delay timing-model is a result of the code-generation methodology, and not an inherent property of levelization.

The PC-Set method is a static scheduling technique which uses levelization to create Unit-Delay and Multi-Delay simulations. Like the methods of Chapter 2, the PC-Set method generates oblivious code. The number of gate simulations is fixed regardless of the content of the input vectors.

6.2 Theoretical Foundations.

To understand the PC-Set method, it is important to understand the difference between zero-delay and unit-delay simulation. The correctness of the unit-delay algorithms discussed in Chapter 3, depends on performing gate simulations at precise instants of simulated time. It is necessary to do this so that the simulation of gates will be synchronized with changes in net values. It is the change in net values at precise moments of simulated time that gives rise to both the unit-delay and multi-delay timing models. To produce the correct sequence of changes, it is usually necessary to simulate the same gate several times using different input values.

To produce the correct sequence of changes in net values, it is necessary for an oblivious unit-delay simulation to perform the same simulations that would have been performed by an event-driven simulation, and it is necessary that these simulations be performed using the correct input values. Although it may appear that the gate-simulation order for the zero-delay model is different from that of the unit-delay model, this is not precisely true. If all duplicate gate simulations are eliminated from a unit-delay simulation, retaining only the final simulation of each gate, the remaining gate simulations will occur in levelized order. The levelization procedure used for zero-delay simulation is essentially correct, as long as the additional simulations for the the more complex timing model are also performed.

The first requirement for creating a statically-scheduled unit or multi-delay simulation is to determine what additional simulations are required by the timing model. This is done by determining the simulated times at which events can occur for the various nets in the circuit. This determination can be made using a procedure which is similar to levelization. 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.

6.3 Computing PC Sets

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}. 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 must remain constant at time *t*, then the outputs must remain constant at time *t*+1. To compute the PC-Set of a gate output, it is necessary to know the PC-Sets of the inputs. The PC-Set of the output is computed by forming the union of the PC-Sets of the inputs, and adding one to each element. Figure 6-1 illustrates the procedure for computing PC-Sets.

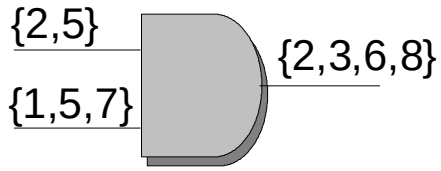

Figure 6-1. The PC-Set Computation.

To handle wired-or connections, and other situations where a net may be driven from more than one source, it is convenient to compute PC-Sets for both gates and nets. 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 procedure is illustrated in Figure 6-2. Gate PC-Sets are also useful for generating code.

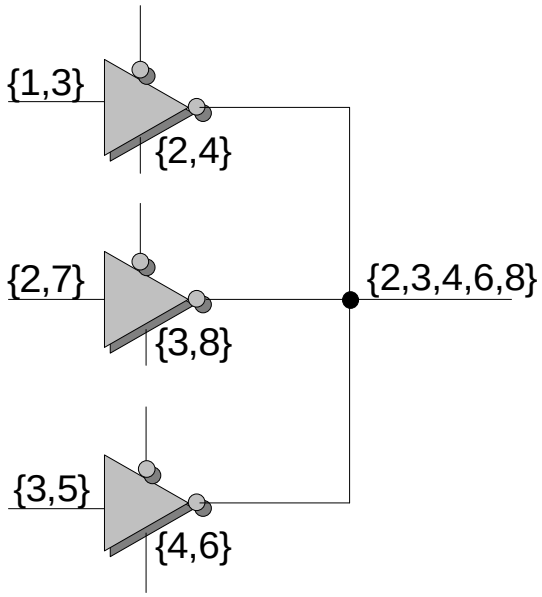

Figure 6-2. Gate PC-Sets and Wired Connections.

Algorithm 6-1 gives the algorithm for computing PC-Sets

```
For Each Primary Input 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
While Ready Queue is Not Empty
    If Top of Queue is a Net Then
            N := Pop Queue;
            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
    Else /* Top of Queue is a Gate */
            G := Pop Queue;
            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
            End If
        End While
```

Algorithm 6-1. Unit-Delay PC-Set Computation.

Figure 6-3 illustrates the computation of PC-Sets for a circuit.

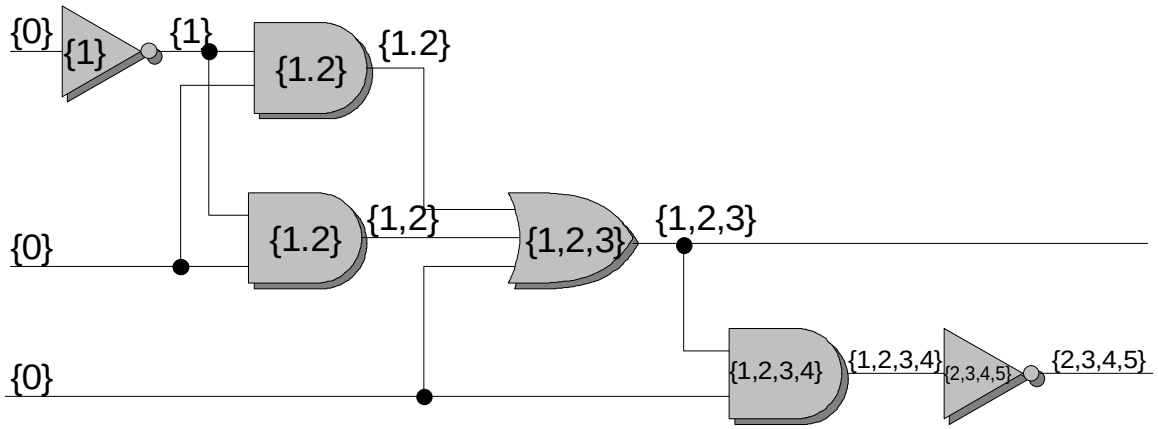

Figure 6-3. PC-Sets for an Entire Circuit.

6.4 Zero Insertion

The Gate and Net PC-Sets can be used to generate a static unit-delay simulation, but to guarantee that each gate simulation processes the proper input values, it is necessary to make one additional modification to the PC-Sets. To see why this is necessary, consider the diagram given in Figure 6-4.

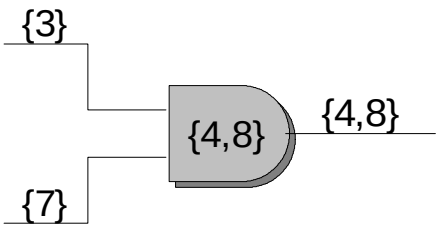

Figure 6-4. The Need for Zero-Insertion.

Because the gate of Figure 6-4 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 6-5.

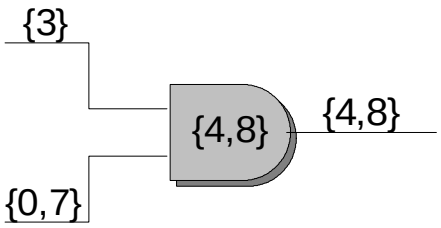

Figure 6-5. A Net After Zero-Insertion.

The PC-Set elements created by zero-insertion are not propagated through the network. A net will require zero inserion 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 6-2 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 6-2. The Zero-Insertion Procedure.

6.5 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 6-6 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.

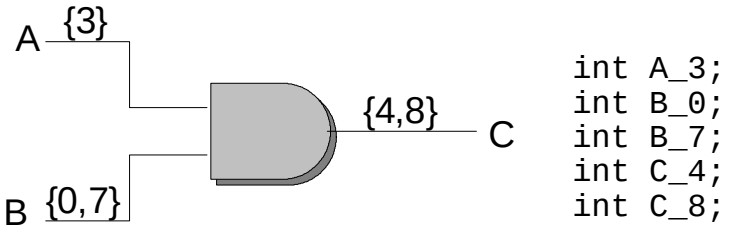

Figure 6-6. Generating Variables.

The Gate PC-Sets are used to generate the executable code. Gates are processed in levelized 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 6-7 illustrates the process of code generation.

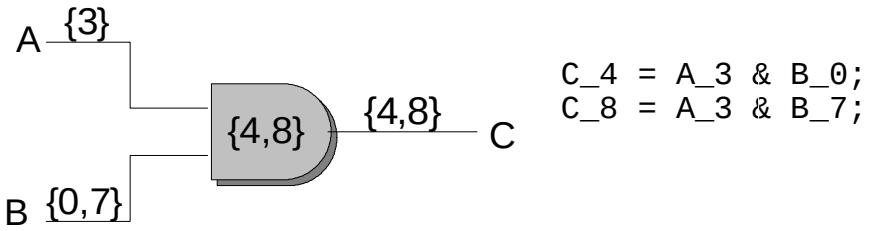

Figure 6-7. Code Generation.

Algorithm 6-3 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 levelized 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 6-3. Code Generation.

The code generated by Algorithm 6-3 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. This code must precede the code generated by Algorithm 6-3.

```
N_0 = N_L;
```

6.6 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 6-8 illustrates this concept.

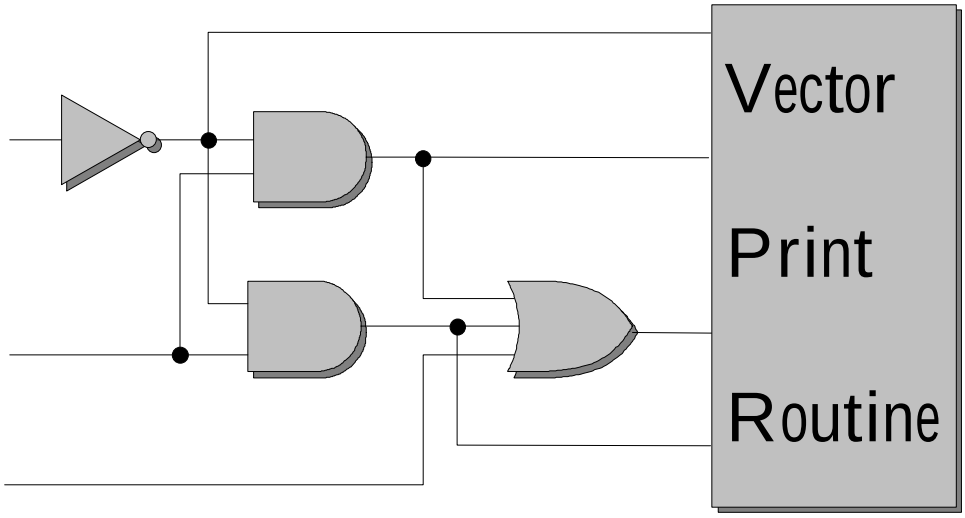

Figure 6-8. The Vector Print Routine.

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 6-3.

6.7 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 6-1 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.

6.8 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 for combinational circuits. 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 6-9.

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

Figure 6-9. 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 the 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 6-10.

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

Figure 6-10. 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 6-11.

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

Figure 6-11. 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.

6.9 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 the obvious modifications. The PC-Set method cannot be applied directly to asynchronous circuits such as that pictured in xxx. 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 xxx 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.

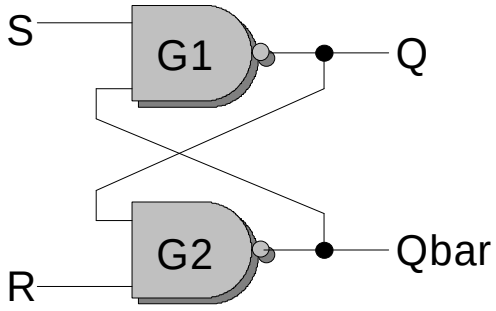

Figure 6-12. An Asynchronous Flip-Flop.

Because the circuit of xxx 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 possible 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. In xxx it is gates x, y, z, and q could potentially be simulated during time slot 1.

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 gate G, and G is said to be an *immediate predecessor* of H. The (not necessarily immediate) 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 distiguish 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*. xxx shows how Gate-Sets are generated, while xxx shows the complete collection of Gate-Sets for the circuit of xxx.

```
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 6- 4. The Gate-Set Algorithm for Combinational Circuits.

For the circuit of xxx, 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 as illustrated in xxx. This code is essentially identical to the code generated by the PC-Set method, with some of the statements rearranged.

```
/* Generate Standard Variables */
Generate the variables N_0 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
/* 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
    /* 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
    /* 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 6- 5. Code Generation for Combinational Circuits.

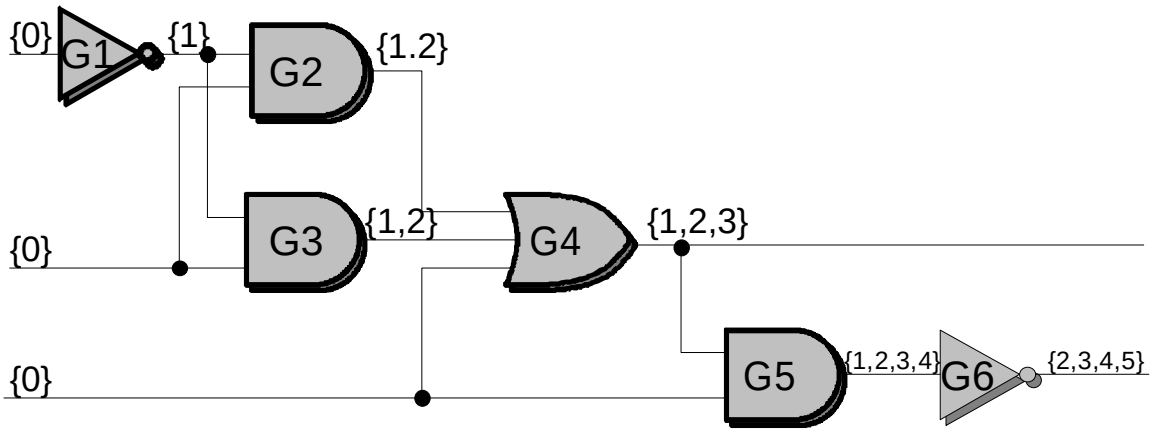

Figure 6- 13. GS(1) For a Simple Circuit.

For combinational 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 $u<j$ $GS_A(t)<>GS_A(u)$ 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 xxx. 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. xxx must be enhanced to detect a repeating sequence of Gate-Sets as shown in xxx. 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.

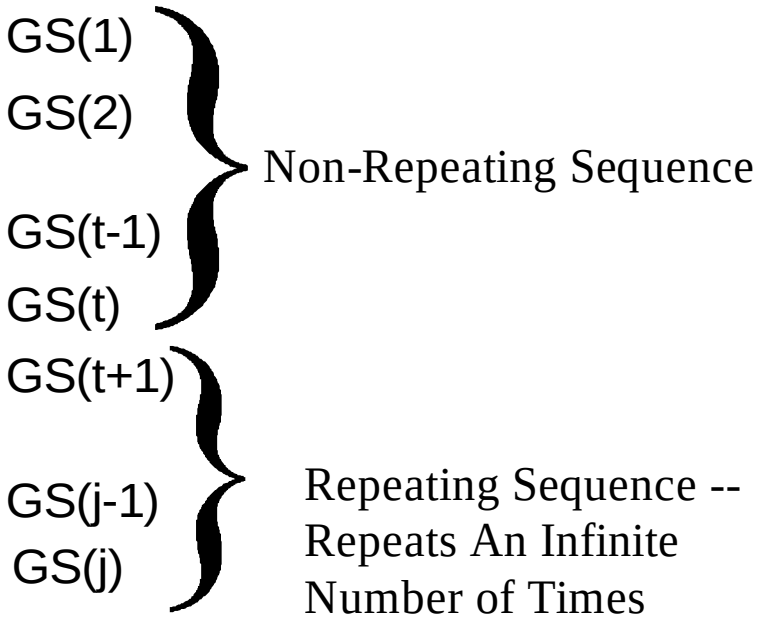

Figure 6- 14. The Repeating Gate-Set Sequence.

```
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 6- 6. 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 exactly 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 two final sets of both sequences are the same, 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 loop iteration is performed. An absolute loop count must be maintained to protect against oscillations. xxx shows the generated code for the circuit of xxx. In this circuit, all gate sets are identical and consist of the set {G1,G2}. The non-repeating sequence consists of GS(1), while the repeating sequence consists of GS(2). The simulation statements of xxx are the assignments to the variables Q_0, Q_2, Q_2, Qbar_0, Qbar_1, and Qbar_2. These statements are identical to those that would be generated for a combinational circuit. 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 6- 7. 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. It is important to note that while the PC-Set Method is not as powerful as the convergence algorithm, PC-Sets provide important information about circuit structure, and are useful in other simulation techniques.

6.10 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 xx with activity rates as high as xx. These experiments have given a break-even activity rate of xx.

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, at least once, for each iteration of the repeating sequence. The number of output nets that must be tested for changes is also large, typically most of the nets in the circuit. Furthermore, if the non-repeating sequence is long, the asynchronous flip-flops in the circuit may be simulated several times before the repeating 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 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.

6.11 Summary

6.12 Exercises