Chapter 7

The Parallel Technique

7.1 The Basics of the Algorithm

In Chapter 6 we learned that oblivious unit- and multi-delay simulation may require several gate simulations to be performed for a single gate. These extra gate-simulations significantly increase simulation time over oblivious zero-delay simulation. The Parallel Technique was introduced as a method of speeding up these additional gate simulations. Chapter 2 introduced the concept of bit-parallel simulation, using the extra bits in a word to perform useful work. The Parallel Technique uses bit-parallel simulation in a more sophisticated way to perform the additional gate simulations required by unit-delay and multi-delay simulation.

Rather than treating a word as a simple collection of bits, the parallel technique indexes each bit in a word, starting on the right with index 0. Figure 7-1 illustrates this indexing, assuming that a word contains eight bits. A longer word will have a correspondingly larger number of indices. Each index corresponds to one unit of simulated time. Bit zero corresponds to time zero, bit one corresponds to time one, and so forth. As in Event-Driven Simulation, these time units are of unspecified length, and are assumed to occur during the processing of a single input vector.

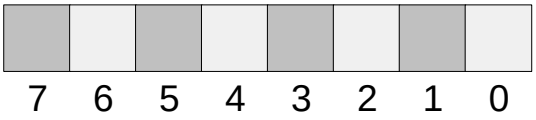

Figure 7-1. Assigning Times to Bit Positions.

Because the principles of the Parallel Technique do not depend on word-length, the term “Bit-Field” will be used to denote an indexed sequence of bits such as that pictured in

Figure 7-1. For each circuit, there is a minimum acceptable bit-field width, M+1, where M is the maximum level number over all gates in the circuit. As long as a bit-field is at least as wide as the minimum acceptable width, any bit-field width can be used. Bit-Fields are used to represent the values of a single net at all times during the simulation of a single input vector. The parallel technique requires one bit-field to be created for each net in the circuit. When a gate is simulated, the bit-fields corresponding to the gate-inputs are combined using bit-parallel operations, and the results are placed in the bit-fields corresponding to the gate outputs. To simplify matters, the discussion will focus on the Unit-Delay version of the Parallel Technique. Extensions to the multi-delay model are straightforward.

In most cases, the bit-parallel operations required to simulate a gate are nothing more than the operations used for zero-delay simulation. Figure 7-2 illustrates one such simulation. Note, however, that the low-order bit of the resultant bit-field corresponds to time 1 rather than time 0. In such cases, the index of the low-order bit is called the Alignment of the bit-field. The change in alignment is due to the delay of the AND gate. Since the gate has a delay of 1, the input values at time 0 combine to form the output value at time 1, and so forth.

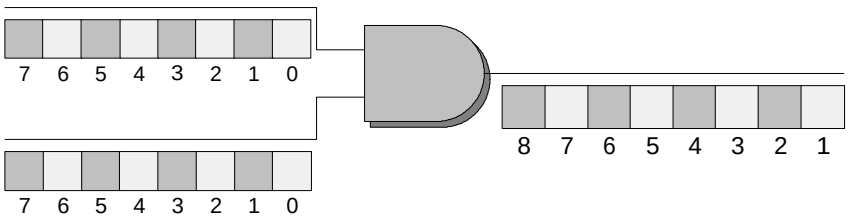

Figure 7-2. Parallel Technique Gate Simulation.

After the simulation of Figure 7-2, it is necessary to correct the alignment of the bit-field to 0. Bit-Field correction is illustrated in Figure 7-3. After AND gate is simulated by performing a bitwise AND operation on the two input bit-fields, it is necessary to shift the bit-field to the left one bit, and introduce a new value into the low order bit. The new value must correspond to the value of the net at time zero. Since time zero also corresponds to the value of the net before the first gate-simulation is performed, this new value must be the value of the net from the previous input-vector. For practical reasons, it is convenient to initialize the bit-field of each net, so that the value from the previous input vector is placed in bit-position zero. The remaining bit positions are initialized to zero. When a gate simulation is performed, the result of the bit-parallel operations is stored in a temporary variable. After the left-shift, the temporary bit field can be combined with the output bit-field by performing a bit-wise OR operation. The complete code for simulating the gate of Figure 7-2 is illustrated in Algorithm 7-1.

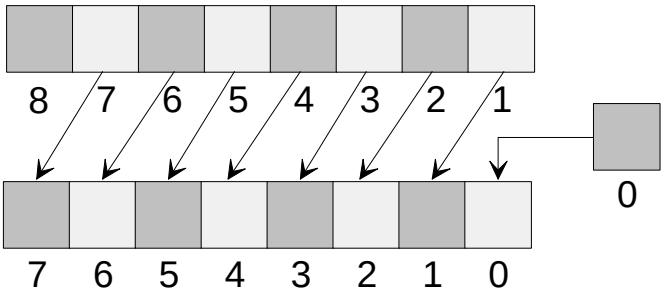

Figure 7-3. Bit-Field Realignment.

```
Temp := B And C
Temp := Temp ShiftLeft 1
A := A Or Temp
```

Algorithm 7-1. Parallel Technique Gate Simulation.

Gate simulations such as that pictured in xxx are generated for each gate in the circuit in levelized order. The gate simulation code must be preceded by the net initialization code. Although net initialization appears complicated, it can be accomplished quite simply. When the simulation of an input vector is complete, the high-order bit of each bit-field is guaranteed to contain the final value of the net. This bit can be shifted into the low-order bit position. If a logical rather than an arithmetic shift is used, the vacated high-order bits will be automatically filled with zeros. Algorithm 7-3 illustrates the process of initializing a net, assuming that the bit-field contains eight bits.

```
A := A RightShift 7
```

Algorithm 7-2. Initializing Nets.

Algorithm 7-2 is not applicable for primary input nets. Primary inputs are assumed to achieve their new values at time zero. Furthermore, primary inputs do not change their values during simulation. When a new value is assigned to a primary input, it is necessary to propagate the new input value through all bits of the bitfield. This can be done using Algorithm 7-3.

```
If A ≠ 0 Then
   A := -1
EndIf
```

Algorithm 7-3. Initializing Primary Inputs.

Simulation using the parallel technique is illustrated in xxx through xxx. It is assumed that this circuit has already been simulated with the input vector (A=0, B=1, C=1), and will now be simulated with the input vector (A=1, B=1, C=0). Since the maximum level number over all gates is two, a bit-field width of 3 is sufficient. It is possible to use a larger bit-field, but if this is done, all high order bits after the third will have the same value, and will be identical to the third bit. xxx illustrates the circuit to be simulated.

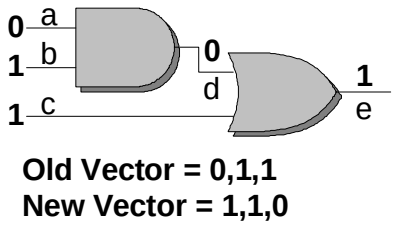

Figure 7-4. A Sample Circuit.

xxx illustrates the bitfield initialization, which is the first step in the simulation of the circuit of xxx.

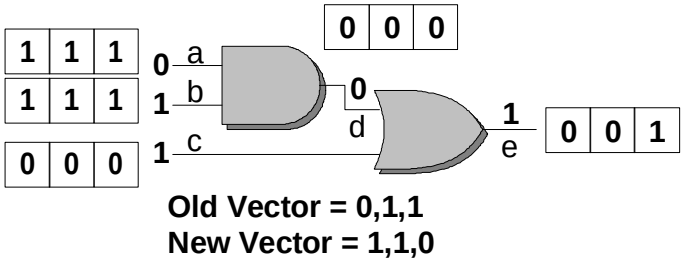

Figure 7-5. Initializing the Bit-Fields.

xxx illustrates the simulation of the AND gate, the generation of the intermediate bit-field and the final adjustment of the bit-field.

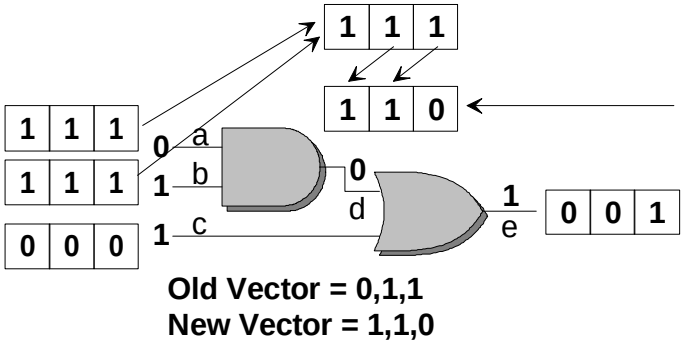

Figure 7-6. Simulating the AND Gate.

xxx illustrates the simulation of the OR gate, which is the final step in the simulation..

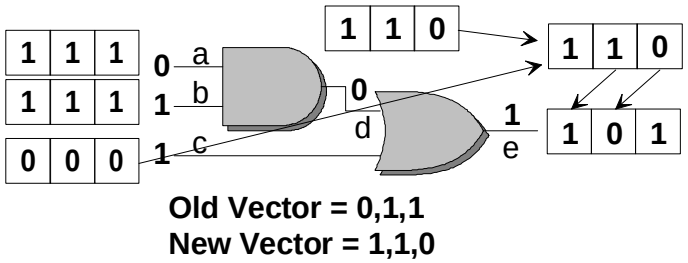

Figure 7-7. Simulating the AND Gate.

Note that the result of the simulation in xxx shows a static hazard in the output of the OR gate. The generated code for the circuit of xxx is illustrated in xxx. It is assumed that the input values have already been read, and placed in the low-order bit of the variables corresponding to the primary inputs.

```
if a <> 0 then
    a = -1;
endif
if b <> 0 then
    b := -1;
endif
if c<> 0 then
    c := -1;
endif
temp = a AND b;
d = (temp ShiftLeft 1) OR d;
temp = d OR c;
e = (temp ShiftLeft 1) OR e;
```

Algorithm 7-4. Simulation of a Simple Circuit.

7.2 Bit-Field Alignments

Apart from the **shift** and **or** operations, the simulation code generated for the parallel technique is identical to that of Zero-Delay Levelized Compiled Code simulation. If it were somehow possible to eliminate the shift operations, unit-delay simulation could be performed as quickly as zero-delay simulation. To investigate this concept further, consider the diagram of xxx. This diagram represents the output bit-field right after simulation is performed, but before the shift has been done. The intermediate output must be shifted to the left one bit to make room for the time-zero value of the net. But, is this value really needed? If the answer is NO, then the shift and OR operations can be dropped, and the alignment of the bit-field can be left at one.

It is useful to list the conditions under which the time zero value would be needed. If there was a change in the net at time one, it would be necessary to retain the time-zero value to show the change. Also, if some other gate simulation used the net as an input, the time-zero value might be required to properly simulate the subsequent gate. If the net

does not change at time one, then the time zero value might not be required. To determine whether the net will change at time one, it is sufficient to compute the PC-Set of the net, as described in Chapter 6. The minimum value of the PC-Set, also known as the minlevel of the net, is the earliest time at which it is possible for the value of the net to change. If a change does indeed occur at this time, it is necessary to retain the net-value from the previous time-unit, so that the change is visible.

The maximum value of the PC-Set of a net corresponds to the level of the net. Using the level and the minlevel of a net, it is possible to compute both the alignment and width of the bit-field. The alignment of the bit-field for the net must be no larger than the minlevel-1. The width of the bit-field must be no smaller than level-minlevel+2. If these two requirements are met, then the alignment and width of the bit-field allow all changes in the output net to be simulated.

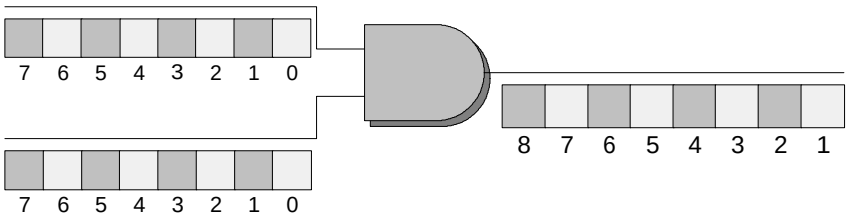

Figure 7-8. Changing Alignments.

To eliminate all shift operations from a simulation, the following conditions must be met.

1. **The alignment and width of a bit-field must allow all changes in the gate to be simulated.**
2. **When a gate is simulated, the bitfield alignments of all inputs must be the same.**
3. **The alignment of the output of a gate must be one larger than the alignment of the input.**

Let N be a net and let L_N be the level of the net and M_N be the minlevel. Requirement 1 can be satisfied by setting the alignment of the bitfield to M_N-1 and the width of the bitfield to L_N - M_N + 2. Unfortunately, as xxx illustrates, it is not always possible to enforce conditions 3 and 4 simultaneously.

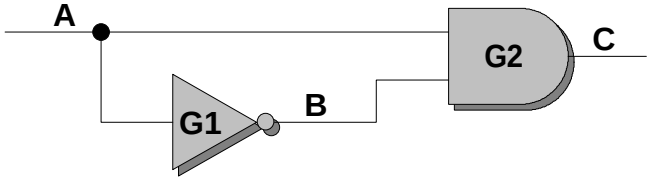

Figure 7-9. Alignment Conflicts.

Condition 2 requires net A and net B to have the same alignment, while condition 3 requires the alignment of net B to be one larger than that of net A. These two conditions cannot both be true. Because conditions 2 and 3 cannot be enforced for this circuit, it is impossible to eliminate all shifts. Most circuits require the retention of some shift

operations. To determine whether a circuit requires shifts, it is necessary to create a graph known as the undirected network graph. Each vertex in the undirected network graph represents either a net or a gate. There is an edge between the vertex for net N and the vertex for gate G, if N is either an input or output of G. The edge is undirected. The following is the undirected network graph for the circuit of xxx.

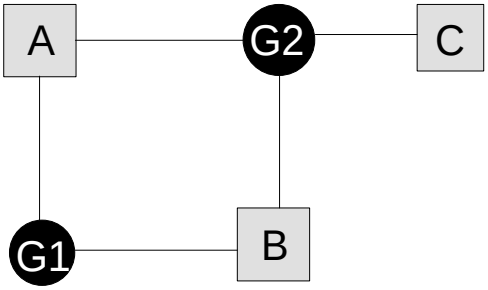

Figure 7-10. The Undirected Network Graph.

Note that the graph of xxx is cyclic. Cycles in the undirected network graph are a necessary, but not sufficient condition for the retention of shifts. If the undirected network graph is acyclic, all shifts can be eliminated from the circuit. If the graph is cyclic, it is necessary to compute the weight of each cycle to determine whether it is possible to eliminate all shifts. The weight of a cycle is calculated by traversing the cycle in either direction, starting with a net-vertex. A vertex-weight of zero is assigned to each net-vertex. The vertex-weight of a gate-vertex is determined during the traversal of the cycle. Gate-vertex values are recomputed for each different cycle, regardless of how many cycles contain the vertex.

When traversing a gate-vertex, it is necessary to keep track of the predecessor and successor of the gate-vertex in the traversal. (Since the traversal begins on a net-vertex, there will always be a predecessor and a successor vertex.) Suppose the net-vertex for gate G is being traversed, and the predecessor vertex represents net N_p, while the successor vertex represents net N_s. If N_p and N_s are both inputs of G, or both outputs of G, then the gate-vertex is assigned a weight of zero. If N_p is an input and N_s is an output of G, then the gate vertex is assigned a weight of 1. If N_p is an output and N_s is an input of G, then the gate vertex is assigned a weight of -1.

Once weights have been assigned to each vertex in a cycle, the weight of the cycle is computed by adding the weights of its component vertices. If the undirected network graph of the circuit contains a cycle of non-zero weight, then it will be impossible to remove all shifts from the circuit. The converse is also true. If all cycles in the undirected network graph have zero weight, then it is possible to eliminate all shifts from the circuit.

Assume that the cycle of xxx is traversed in counter-clockwise order. G1 will assigned a weight of 1, because A is an input of G1, and B is an output of G1. G2 is assigned a weight of 0, because A and B are both inputs of G2. The weight of the cycle is 1. This shows that it is impossible to remove all shifts from this circuit.

7.3 Eliminating Shifts

The "trick" to eliminating shifts is adjusting the alignments and widths of bit-fields to reduce the number of bit-field realignments. For any circuit, it is advantageous to have all bitfields the same width, so it is necessary to compute the maximum of the required bit-field width, $\mathbf{L_N}$-$\mathbf{M_N}$ +2, over all nets N. This value will be used as the bit-field width for all bit-fields.

Net alignments are set using a procedure called the path-tracing algorithm. Gates are processed starting with the primary outputs, and proceeding backwards through the circuit to the primary outputs. For each primary output net **N**, the alignment is set to $\mathbf{M_N}$ - 1. Gates are processed in reverse order by level. When a gate is processed, the alignment of each input net is set to one less than the alignment of the output net. In some cases, a net will already have an alignment when this procedure is executed. If this is the case, the new alignment replaces the old alignment *only* if the new alignment is less than the old alignment. xxx shows the path-tracinc algorithm, while xxx gives an example of computing alignments using the path-tracing algorithm. The alignment initialization value in xxx can be any number larger than the maximum level in the circuit, such as the total number of nets and gates in the circuit.

```
Compute N.PCSet for Each Net N in the circuit.
For Each primary output P do
        S := P.PCSet;
        L := Max(S);
        M := Min(S);
        P.Align := L-M+2;
EndFor
For Each Net N in the Circuit do
        N.Align := Some Large Value;
EndFor
For Each Gate G in Descending order by Level do
        N := G.OutputNet;
        NewAlign := N.Align - 1;
        For Each Input X of G do
                If X.Align > NewAlign Then
                        X.Align := NewAlign;
                EndIf
        EndFor
EndFor
```

Algorithm 7-5. The Path Tracing Algorithm.

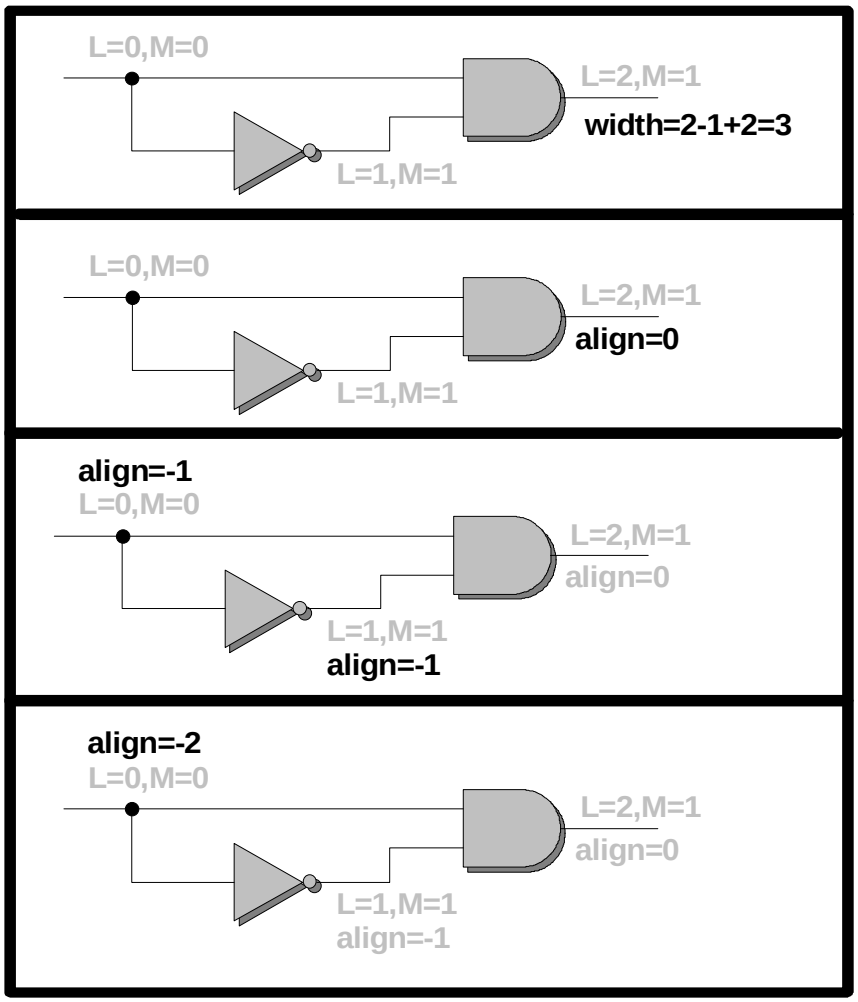

Figure 7-11. The Path-Tracing Algorithm.

In the unoptimized parallel technique, all shifts are to the left, and all shifts are one-bit shifts. If varying alignments are used,a retained shift could be either to the right or to the left, and multiple-bit shifts may be necessary. It is also possible to create alignments that are less than zero. Since any time-unit less than zero represents a time before the simulation begins, all bit-positions that have negative indices must contain the final value of the net from the previous input vector. This is true for primary inputs as well as other nets. Because primary inputs are assumed to change at time zero, any bit-positions with negative indices must contain values from the previous input vector. When new primary input values are read, the new value is propagated through all non-negative bit positions of the primary-input bit-field. The value read from the previous input vector is placed in all negative bit-positions of the bit-field.

The Path-Tracing algorithm has several desirable properties. If G is a gate with output N, and the alignment of N is A, then the alignment of the inputs of G will be A-1 or less. This implies that the bit-fields for the inputs of G will contain bits corresponding to time N-1, which are the values needed to compute the value of the low order bit of N. This implies that the initialization of intermediate net values is unnecessary. Another important property, is that the width of the bit-fields will never be larger than those for the

unoptimized circuit. When path-tracing is used, it is never necessary to shift the bit-field after performing a gate simulation, but it is often necessary to align some input bit-fields before a gate simulation occurs. If the alignment of a gate output is A, all inputs must be aligned to A-1. If the any input has a different alignment, a shift must be performed to correct it. xxx illustrates simulation with negative alignments.

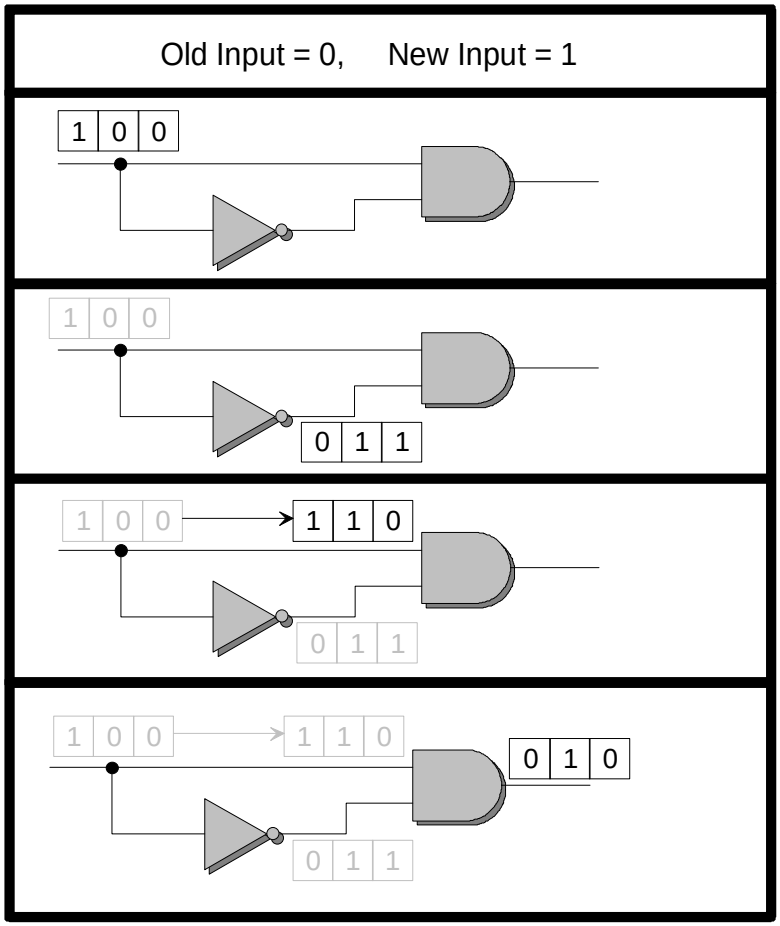

Figure 7-12. Simulating with Negative Inputs.

When inputs are aligned, arithmetic right shifts are used, so the high-order bit of the bitp-field is replicated throughout all vacated bit positions. For intermediate nets, it is never necessary to retain values from the previous input vector. If such values are required, they will be recomputed as needed.

Although the path-tracing algorithm can eliminate many shifts from a parallel-technique simulation, it is not optimal with respect to removing shifts. The circuit of xxx has an acyclic undirected graph, so all shifts can be eliminated from the simulation of this circuit. As xxx shows, the path tracing algorithm aligns nets so that one shift is required. When simulating G6, it is necessary to shift input I1 by two bits. Nevertheless, it is possible to eliminate all shifts from this circuit, as the alignments shown in xxx show. Eliminating this extra shift is not necessarily desirable, because the choice of alignments can affect the width of the bit-field. The formula given above, $\mathbf{L_N}$-$\mathbf{M_N}$ +2, applies only to the alignments created by the path-tracing algorithm. The maximum bit-field width in both

xxx and xxx can be determined from the alignment of the net I2. In xxx the bit-field for this net requires only 4 bits, while in xxx the bit-field requires six bits. In practical applications, the expansion of the bit-field can be a serious matter. On a computer with 32-bit words, expansion of the bit-field from a value less than 32 to a value greater than 32 will have a significant impact on the performance of the simulation. Since each bit-field will require two words instead of one, the number bit-parallel operations necessary to perform the simulation will double. Any required shifts will be significantly more complicated, because bits will need to be shifted from one word to another. The path-tracing algorithm is guaranteed not to expand the size of the bit-field beyond the basic requirement of $\mathbf{L_N}$-$\mathbf{M_N}$ +2, and eliminates a significant number of shifts.

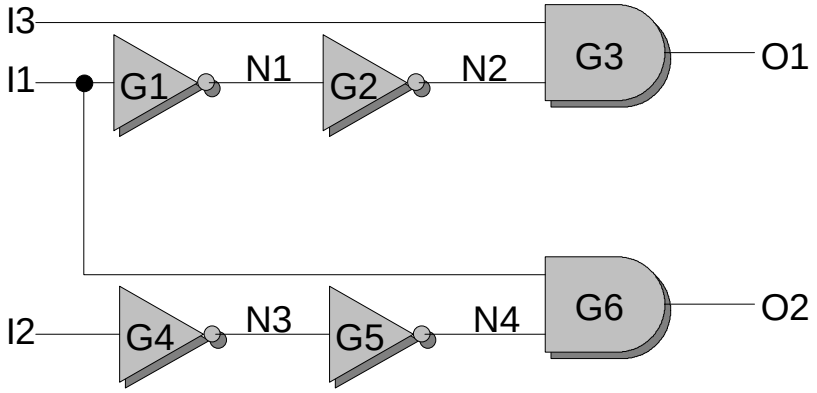

Figure 7-13. Non-Optimal Path-Tracing.

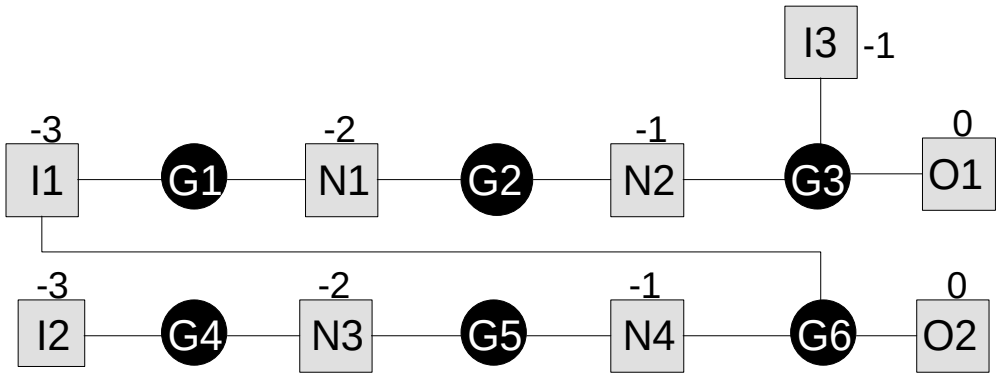

Figure 7-14. An Acyclic Undirected Network Graph.

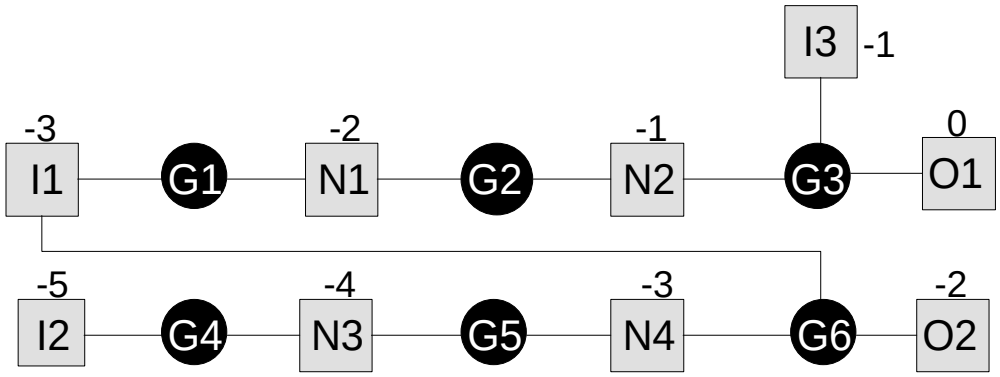

Figure 7-15. Expanding The Bit-Field.

7.4 Asynchronous Sequential Circuits

Because the parallel technique uses levelization, it is essentially a combinational technique. The algorithm can be extended to synchronous combinational circuits by breaking the circuit at the flip-flops as outlined in Chapter 2. This has the effect of converting the circuit into a combinational circuit, which can be handled in the obvious manner. This technique will not work for asynchronous sequential circuits, because the changes in the flip-flops are not synchronized with a clock pulse, and may not be simultaneous with one another.

Asynchronous circuits can be handled using a technique similar to the forced levelization technique of Chapter 2. The first step in this procedure is to identify the strongly connected components of the circuit. A strongly connected component of a circuit is a set of gates S, such that for any two gates G_1 and G_2 in S, the output of G_1 depends on the output of G_2 and the output of G_2 depends on the output of G_1. Stated more simply, two gates are in the same strongly connected component if there is a circular logic path that includes both gates. Finding strongly connected components is a standard problem in the theory of directed graphs, and the graph-theoretical algorithms for this problem can be adapted to finding the strongly connected components of a circuit. Algorithm 7-6 illustrates the strongly-connected component algorithm, which is recursive.

```
SCC
        For Each Vertex V do
                V.Visit := 0;
        EndFor
        Visit := 0;
        ComponentNum := 0;
        For Each Vertex V do
                if V.Visit = 0 Then
                        DFS(V);
                EndIf
        EndFor
EndSCC
```

```
DFS(V:Vertex)
        Visit := Visit + 1;
        V.Start := Visit;
        V.Done := False;
        V.Back := Visit;
        For Each Vertex W Adjacent to V do
                If W.Start := 0 Then
                        DFS(W);
                        V.Back := Min(W.Visit,V.Back);
                ElseIf not W.Done Then
                        V.Back := Min(W.Visit,V.Back);
                EndIf
        EndFor
        If V.Back = V.Start Then
                ComponentNum := ComponentNum+1;
                V.Done := True;
                V.Component := ComponentNum;
                While Stack is Not Empty And
                    Top(Stack).Visit > V.Visit do
                        V.Component := ComponentNum;
                        V.Done := True;
                        Pop(Stack);
                EndWhile
        Else
                Push V onto Stack;
        EndIf
EndDFS
```

Algorithm 7-6. Strongly Connected Components.

If xxx were run on the graph illustrated in xxx, four strongly connected components would be identified, namely {A1, A2, A3}, {B1, B2, B3}, {C1, C2, C3, C4, C5} and {D1, D2, D3, D4, D5}. Once the strongly connected components of a graph have been identified, each component can be collapsed into a single vertex, as illustrated in xxx. If there is an edge between vertices of two strongly connected components, then there must be an edge between the two vertices that correspond to these components. Duplicate edges between vertices are collapsed into a single edge. The collapsed graph is always guaranteed to be acyclic.

These graph-based algorithms can easily be adapted to handle logic-level circuits. After assigning each gate to a strongly connected component, each of these components can be collapsed into a sub-circuit. (Many of these sub-circuits will contain a single gate.) The circuit is levelized with respect to the sub-circuits, and simulation code for each sub-circuit is generated in levelized order.

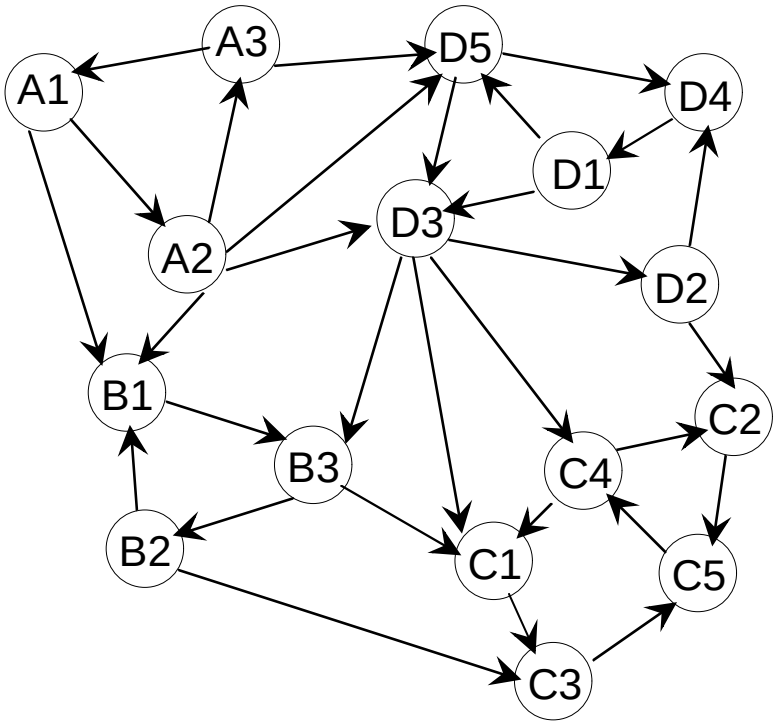

Figure 7-16. Strongly Connected Components: An Example.

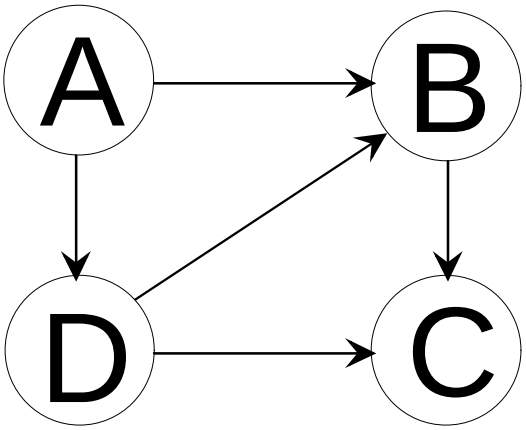

Figure 7-17. A Collapsed Graph.

The gates of a strongly connected component are all contained in one or more logic-loops, or *cycles*, as they are more commonly called. To understand how code is generated for strongly connected components, it is necessary to understand the effect of simulating a cycle of gates, and the effect of simulating a single gate in that cycle. To motivate the discussion, consider the circuit illustrated in xxx. In xxx, there are gate inputs of two different types, those inputs that come from outside the component are completely known for all simulation times, while those that are generated inside the circuit have known initialization values, but all other values are unknown.

Rough notes from here on.

The first step in the process is to find the strongly connected components of a circuit. Each such component is collapsed into a sub-circuit, yielding an acyclic graph. The circuit is levelized with respect to this graph, and code is generated for the blocks in levelized order. The code for a strongly connected component must be special, but the rest of the code is the same as for combinational circuit. In a strongly connected component, the component is simulated over and over again until nothing changes. If the longest (shortest?) cycle in the strongly connected component is of length x, then each simulation of the SCC is guaranteed to produce x correct bits for each net in the circuit. These bits accumulate in units of x bits with each iteration of the component. Convergence can be detected by comparing segments of bit-fields.

Figure xxx. Show accumulation of bits.

The asynchronous parallel technique uses REALLY wide bit-fields, which makes it slow.

7.5 4.1. The algorithm.

The fundamental idea is to break the circuit into strongly connected components, and simulate the strongly connected components in levelized order. (A strongly connected component is a maximal set of gates such that the outputs of any two gates in the set depend on each other.) Within each strongly connected component, feedback arcs are identified and broken, and the gates in the component are levelized. Code is generated for each gate in the component in levelized order, as described in Section 2 above. This code is then executed iteratively to generate a predetermined number of output bits.

To understand why this approach works, suppose that one is simulating a simple cycle containing *n* gates, as illustrated in Figure 8. The values of all inputs to the cycle will have been completely computed before this component is simulated. Because of the standard initialization code, the correct value of the feedback arc, F1, is known at time zero. After simulating G1, the correct value of N1 will be known at time zero and time one. Similarly, after simulating gate G2, the value of N2 will be known at times zero, one, and two. After the component is completely simulated once, the value of the feedback arc, F1, will be known at times zero through *n*. After the first iteration, each subsequent iteration will produce *n* new correct bits for each net. The successive *n*-bit values produced on the feedback arc can be tested for changes to determine when to terminate the iteration, but the current simulator simply iterates until a fixed number of bits have been generated. This number, which is supplied manually to the compiler, is determined by running the interpretive simulator and measuring the maximum number of simulation cycles done for a single vector. We have experimented with a version of the compiler that generates dynamic tests instead of a fixed number of iterations, but the performance of this technique turned out to be worse than the results shown in the next section.

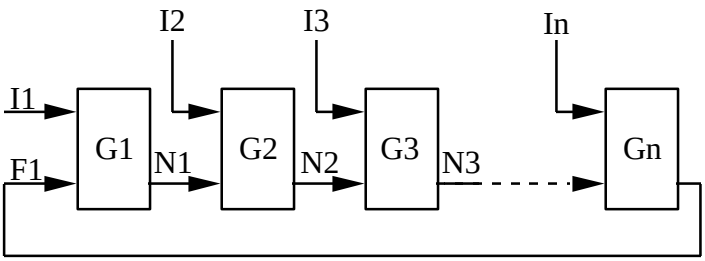

Figure 8. A Simple Loop.

There is some room for improvement in this algorithm. The number of bits generated per iteration depends on the length of the shortest loop in the strongly connected component. Obviously, the more bits per iteration, the better the algorithm will perform. The performance also depends on the number of feedback arcs detected in the strongly connected component. The fewer arcs that are detected, the better the performance will be. (The number of feedback arcs is *not* constant, and depends heavily on where the algorithm begins its search, and in what order gates are visited.) Although the relationship between the number of feedback arcs and the length of the shortest cycle is not known, minimizing the number of feedback arcs appears to produce longer cycles than choosing some non-optimal number. Unfortunately, the problem of minimizing feedbacks is known to be NP-Complete, which will make future improvements to this algorithm difficult.

7.6 4.2. Experimental Results.

Figure 9 contains experimental data for the asynchronous parallel technique. This data was obtained in the manner described in Section 2.2 above. As in Figures 2 and 7, the numbers represent CPU-seconds of execution times. The times for interpreted simulation may differ from those presented in Figures 2 and 7 due to inaccuracies in the UNIX /bin/time command, even though the same simulator and the same input vectors were used for all simulations.

	Interpreted	Compiled	%-Change
s27	1.9	3.1	-63.16
s208	7.0	12.9	-84.29
s298	10.3	21.2	-105.83
s344	16.5	41.3	-150.30
s349	16.6	40.2	-142.17
s382	13.0	30.1	-131.54
s386	15.0	13.9	7.33
s420	12.8	40.0	-212.50
s444	14.1	32.7	-131.91
s510	9.5	20.2	-112.63
s526	15.3	27.9	-82.35
s526n	15.2	28.4	-86.84

Figure 9. Asynchronous Parallel Technique Data.

7.7 Performance

The parallel technique works well for combinational circuits, giving performance improvements of up to 10x. The synchronous parallel technique doesn't work so well because of the very wide bit-fields.

7.8 Summary

7.9 Exercises