Chapter 5

The Generation Algorithm

This chapter applies the knowledge gained in the previous chapters to the problem of *generating* benchmark circuits. Our fundamental goal is to be able to automatically create synthetic circuits which are good proxies for real circuits.

5.1 Overall Approach to Circuit Generation.

Before deciding on a method for generating circuits, it is necessary to refine our primary goal of "generating good circuits," by introducing a number of specific requirements:

Requirement 1. The generation algorithm must scale, and must be fast enough to generate *very large* circuits.

Put simply, the user should be able to specify the circuit-size, and the algorithm should react accordingly to generate a reasonable circuit of the requested size. Since state of the art large ASIC circuits are in the one million gate range, the algorithm cannot use more than $O(n \log n)$ time or space—quadratic time for 10,000 LUT-nodes would amount to weeks of processing time for one circuit.

Requirement 2. The generation algorithm must use *reasonable* input parameters.

Later, we will discuss the concept of *cloning* an existing circuit, by extracting its exact parameterization for input to the generation tool. This begs the question of "how much" information should be included in such a parameterization. We will restrict our generation algorithm to taking a *constant* amount of information, that is the parameterization cannot grow arbitrarily with the size of the circuit being generated. To do otherwise would not only violate the spirit of benchmark generation, but would simply introduce too many variables

into the problem. For the purposes of this restriction, we assume that combinational delay and maximum fanout are no more than logarithmic in circuit size, since they must be close to constant for electrical and performance reasons (with a small number of exceptions for clocks, clears and presets, which we consider special cases).

So, the Rent exponent r (a single number) or the shape vector from Chapter 3 (a vector of length $d+1$) would be considered reasonable in this sense. However, a mincut partition tree, an initial placement, or a "seed" circuit would be prohibited as input parameters.

Requirement 3. The circuits that we generate must have reasonable behaviour with respect to unspecified metrics.

If the method generates circuits with a specific size, shape and delay, it should have reasonable expectations on, for example, wirelength after global routing, even if wirelength is not a parameter. Similarly, if the circuit is generated simply as a graph with a specified wirelength, it must have reasonable combinational delay and fanout, and must not have undesirable properties such as combinational loops or pathological properties such as large cliques in the underlying graph.

With these requirements in mind, there are a number of approaches to generating random circuits:

One method is to simply use random graphs, generated by known methods (one of which is discussed in Section 6.1). This method is attractive in the sense that it is relatively easy to generate random undirected graphs, or random undirected graphs with restrictions on degree under a natural model. Such graphs have been used in famous partitioning papers by Kernighan and Lin [46] and Johnson [45]. However, random graphs from natural models are known to exhibit behaviour such as having too many edges [64] and inordinately high cut-sizes [2, 3]. There are few known methods for generating directed acyclic graphs under natural models, and no known ability to control longest path and cycles in such graphs, such as would be needed for Requirement 3.

A second approach would be to work from a geometric placement, independent nodes on a grid, and add edge-connections based on statistical wirelength distributions and cut-sizes, essentially working from the wireability studies of El Gamal, Donath, Feuer and others (see Chapter 2). The difficulty with this method again lies with the realism of the circuits for anything other than the placement or partitioning problems. The effects of combinational delay and combinational cycles cannot be controlled, because the method inherently has no

concept of directed edges or combinational delay. In a modern CAD system delay is often the most important consideration in layout, so we require an approach which models delay appropriately.

Another approach is offered by Rent's Rule. Darnauer and Dai took this approach in their work, previously mentioned in Section 2.2.3. Though this can yield reasonable undirected graphs for partitioning, it suffers, as does the previous method, from an inability to control delay, fanout and other important electrical features of the circuit.

Our method will be to generate a circuit according to the model which we have developed in the previous two chapters. Doing so provides a number of desirable properties. By making delay and fanout an intrinsic part of the circuit, we obviate dealing with the above problems in other methods. However, we then lose other physical properties of the circuit, namely the existence of a good partition tree as would be guaranteed by Darnauer and Dai, or a known wireability distribution as per the second method. The locality discussion of Section 3.5 addresses this issue, and our empirical validation will illustrate our success in dealing with both delay and locality at the same time.

5.1.1 How We Generate Circuits.

Our algorithm for generating circuits is divided into three topics: combinational circuits, sequential circuits and implementation details.

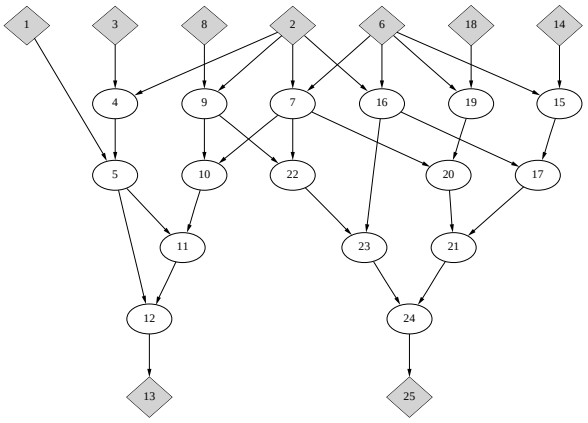
In the next section, Section 5.2, we discuss how to generate purely combinational circuits. We model combinational circuits using the descriptions of delay, shape and fanout from Chapter 3, and build combinational circuits to that model.

In Section 5.3 we expand on the algorithm to generate sequential circuits using the model of Chapter 4. This involves two aspects: how to modify the combinational algorithm to deal with new sequential parameters, and how to generate complete circuits from sub-circuits.

Section 5.4 discusses some implementation details for the algorithm and the tool GEN which realizes it. We discuss the issues of parameterization scripts, circuit "clones," runtime of the algorithm and the ability of the tool to meet its specification. The issue of the quality of circuits (empirical validation) is left to a separate Chapter 6.

5.2 Combinational Circuit Generation.

We begin with an example. Figure 5.1 shows the output from GEN for the parameterization: n=23, n_{edges} = 32, k=2, n_{PI}=7, n_{PO}=2, d=4, shape=(.38,.31,.19,.12), max_out=4, fanouts=(.09,.65,.13,.04,.09), edges=(0,.9,.1) and L=6. (Note that L has not yet been defined.)



$$
\begin{aligned}
n &= 23 \\
n_{\text{edges}} &= 32 \\
k &= 2 \\
n_{\text{PI}} &= 7 \\
n_{\text{PO}} &= 2 \\
d &= 4 \\
\text{shape} &= (7,6,5,3,2) \\
\text{edges} &= (0,29,3) \\
\text{max\_out} &= 4 \\
\text{fanouts} &= (2,15,3,1,2) \\
L &= 6
\end{aligned}
$$

Figure 5.1: Example of a completely parameterized combinational circuit.

The combinational portion of the GEN algorithm consists of two functional stages.

The first stage is to determine an exact and complete parameterization of the circuit to be generated, using partially-specified user parameters and default distributions—the exact parameterization shown to the right of Figure 5.1 is such an instantiation of the more general parameters just given. This issue of defining statistical relationships between circuit characteristics (the "profile") has been discussed in the previous two chapters, and we will remark further on it in Section 5.4.2 and Appendix A.

The second stage is to create and output a circuit-graph with that exact parameterization, and we deal with this below.

5.2.1 The Combinational Generation Algorithm.

Here we give the details of the generation algorithm for combinational circuits.

The inputs to GEN are n, n_{edges}, n_{PI}, n_{PO}, d (delay), k (LUT-size), max_out (maximum allowable fanout of any node), the shape function, the fanout and edge length distributions

and the locality parameter L (not yet defined). The output is a netlist of k-input lookup-tables. Reconvergence is not a generation parameter but we use the reconvergence number of generated circuits in the validation process of Section 6.3.

Since parameter expansion has already taken place, we know the distributions are exact, meaning that

$\sum_{i=0}^{d} \text{shape}[i] = \sum_{i=0}^{\text{max\_out}} \text{fanouts}[i] = n$, and

$\sum_{i=0}^{d} \text{edges}[i] = \sum_{i=0}^{\text{max\_out}} i \cdot \text{fanouts}[i] = n_{\text{edges}}$.

Using the shape distribution, shape[1..d], we are immediately able to define the number of nodes at each combinational delay level. Fanouts[1..max_out] gives us the exact set of fanouts available (but not yet assigned to nodes). Edges[1..d] gives us the set of edges to be assigned between nodes. Our problem is then, as illustrated in Figure 5.2, to determine a one to one assignment of fanout values to nodes, and an assignment of edges between nodes such that the number of out-edges from a node equals its assigned fanout, and the number of edges in to a node is no more than the bound, k, on fanin. We have a number of further constraints: the resulting graph must be acyclic (as the circuit is to be combinational); every node must have at least one fanin from the previous delay level, and no fanins from later delay levels (so that combinational delay of the node as specified by the shape function is correct); all nodes at delay-0 (i.e. the inputs) have no fanins, and all other nodes have at least 2 fanins; and all fanins to a node must come from distinct nodes (no duplicate inputs).

We need the following definitions:

(a) N_i, i=0..d is the set of nodes at delay level i, where $N = \bigcup\{N_i\}$,

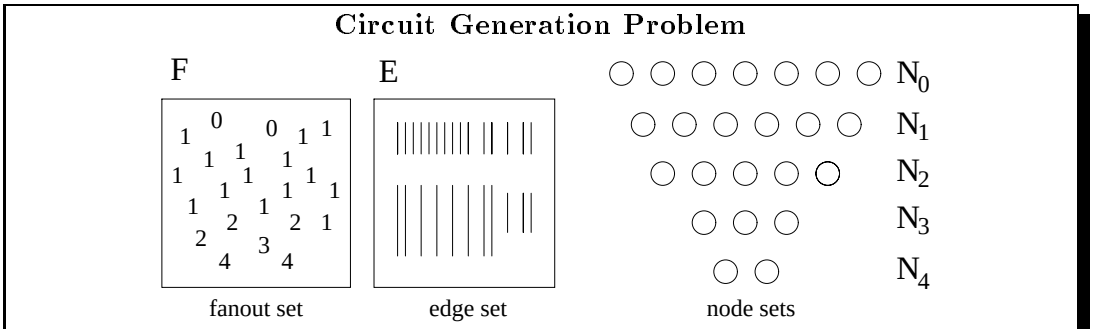
(b) $n_i = |N_i|$,

(c) $F = \{f_j, j = 1..n\}$, is the set of node fanouts, and

(d) $E = \{e_h, h = 1..n_{\text{edges}}\}$, is the set of edge-lengths (abstractly, the set of all edges).

We formally define the generation problem in Figure 5.2.

This assignment problem appears to be computationally difficult and we conjecture it is NP-complete. The existence of a polynomial time algorithm would be relatively uninteresting, however, unless it was both $O(n \log n)$ time or less and still allowed us to have different (i.e. random) outputs on each execution. We require a nearly linear time algorithm in order to generate large circuits. Therefore we solve the problem heuristically, as described in detail in the subsections which follow.



Given: F, E, N_i.

Find: assignments of nodes in N to each $f_j \in F$, and pairs of nodes for each $e_h \in E$ such that:

1. The number of edges leaving any $x \in N$ is exactly its corresponding fanout f_x.
2. All $x \in N_i$ have at least one fanin from N_{i-1} $(i > 0)$. (i.e. the calculated delay(x) equals its assigned value.)
3. Fanin$(x) \leq k$ for all $x \in N$.
4. Fanins of $x \in N$ are distinct (i.e. no two fanouts of gate y are both inputs to x.)

Figure 5.2: The generation/construction problem.

The general line of approach is as follows: First we determine an assignment of edges and out-degree to levels N_i, but not yet to individual nodes within each level. We call the N_i *level-nodes* and the graph at this point the *level-graph*. We then split each level into nodes and assign first fanouts and then edges, previously assigned only to levels, to the individual nodes. A post-processing step designates any additional primary outputs required.

There are 5 major steps in the algorithm for generating a combinational circuit from an exact specification. We provide enough detail here to understand the important aspects of the algorithm. Readers who are interested in the more detailed aspects of the software are referred to the external documentation and the freely available implementation and source-code [40]. Throughout the description of the algorithm, we will follow through the small example of Figure 5.1, from the exact parameterization to the final circuit. For each major step we indicate the module name in the implementation.

The final algorithm shown here is the result of a great deal of experimentation. Earlier versions broke up the problem differently, or did steps in a different order. Some of the major decisions which lead to the better performance of the final algorithm were the boundary

calculations in Step 1 and the decision to divide the allocation of edges to both before and after degree assignment.

Step 1: Compute bounds on in and out degree for each level (pre_degree.c).

When we (later) assign actual edges between levels, we implicitly set the total fanin and fanout for each level. Because we want to do edge assignment quickly, with no backtracking, it is useful to have upper and lower bounds on fanin and fanout for each level.

As a result, the first step of the algorithm is to determine the maximum and minimum fanin (in-degree) and fanout (out-degree) for each delay level: vectors min_in[i], max_in[i], min_out[i] and max_out[i]. While the number of nodes at each level is known, the total fanin is not known exactly because a four input LUT may only have two or three inputs in many cases. For 2-LUTs (as in our example) the fanin bound is deterministic, because we enforce the rule of no single-input nodes.

We require each node at level i to have between two and k fanins, one of which must come from the preceding delay level to establish combinational delay. This gives immediate rough bounds of min_in[i]$=2\cdot n_i$ and max_in[i]$=k\cdot n_i$. Similarly, each non-primary-output node must have at least one fanout, providing an initial lower-bound min_out[i]$=n_i-(n_{\mathrm{PO}}-n_d)$.

Max_out[i] is calculated heuristically using the fanout distribution and the previously calculated vectors for later levels, based on a number of rules: max_out[i] is bounded above by $\sum_{j=i+1}^{d}$ max_in[j] - $\sum_{j=i+1}^{d-1}$ min_out[j] representing the remaining inputs in the LUTs at later levels less the reserved output edges for later levels; max_out[i] is also bounded by $n_i \sum_{j=i+1}^{d} n_j$ to avoid double connections and by the sum of the n_i largest elements in the fanout list F (i.e. the maximum fanout of *any* n_i nodes regardless of location).

The initial bounds are improved iteratively: the bounds on max_out just determined necessitate an updated calculation of max_in and min_in for later levels which in turn affect max_out[i]. We continue until no more tightening of the boundaries is possible, which is no more than d^2 iterations: we iterate d times, and iteration i fixes (at least) the bounds for level i by looking at the d other levels.

The result of this step is the determination of the boundary vectors min_in[i], max_in[i], min_out[i] and max_out[i], $i=0..d$, as pictured in Figure 5.3 (Step 1). Each level-node N_i is labeled with n_i and its fanin boundaries (upper left corner) and fanout boundaries (lower left corner). Sometimes, in particular for small circuits, these bounds can be very tight.

In general, however, the upper and lower bound for fanout will differ by about 10-15%. In the case of fanin, the difference is dependent on the average fanin / number of edges in the circuit: for fanin 2 the bounds will be exact, and the upper and lower bounds will diverge to about 10% as the average-fanin hits $k = 4$.

Step 2: Assign edges between levels (levels.c).

Now that we have some idea of the number of edges to be assigned to and from each level, we will proceed with initial edge assignment. In this step, we will assign most, but not all edges. Recall that we are not assigning edges between nodes, just allocating them between combinational delay levels.

There are three phases to Step 2. As edges are assigned, we calculate two new vectors, assigned_in[i] and assigned_out[i] to represent the "used up" in and out-degree for level i. The *available* in and out-degree to a level is defined as the difference between the assigned and the maximum, and the *required* in and out-degree is defined as the difference between the assigned and the minimum (or 0 when assigned is larger than minimum).

Step 2(a). We first consider the "critical" unit edges, edges which lie on the boundary of the first and last levels of the circuit or which are required to ensure that combinational delay constraints can be met. We assign MAX(min_out[0], min_in[1]) edges between levels 0 and 1, and MAX(min_out[$d-1$], min_in[d]) edges between levels $d-1$ and d. Then we establish the combinational delay for each other level i, $i = 2..d-1$, by assigning n_i edges between levels $i-1$ and i.

Step 2(b). Secondly, we assign the *long* (length > 1) edges. This is a crucial step, because if these are assigned poorly it becomes difficult or impossible to complete the graph construction without violating the shape or edge-length distributions. Long edges are assigned probabilistically. We calculate the number of possible level to level starting and ending point combinations for edges of length l at each level i, MIN(avail_out[i], avail_in[$i+l$]), and sample the resulting discrete probability distribution to assign the edges, updating the distribution after each assignment[1]. It is an important feature of GEN that we *sample* from

[1]Given the discrete probability density function, we can sample by generating the cumulative density function, choosing an integer randomly and uniformly, scaling it to the sum of the cdf (area of the pdf), and indexing into the appropriate value. Because the pdf is created in order to do allocation, rather than a single sample, we want to emulate the idea of sampling without replacement, so once we have sampled a value, we then have to adjust the pdf to lower the probability of taking the same value again. Often we often have to modify the pdf further. For example, choosing a fanout value of 20 and 30 might be equally likely in the pdf, but it might not be possible to have both in the same circuit. Thus, when one is

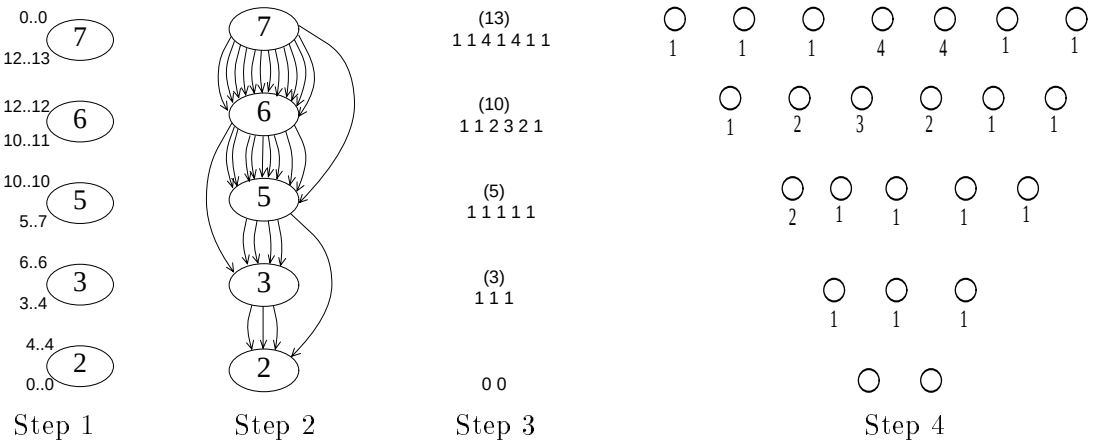


Figure 5.3: Example at the conclusion of Steps 1 to 4.

this distribution rather than just choosing the "optimal" assignment, because we want to produce circuits with different features on each execution with the same parameterization.

Step 2(c). We have only unit edges left. The last part of this step is to assign the remaining *required* edges—those necessary in order to meet the required min_in[i] and min_out[i] for each level i. This part is purely deterministic. Any remaining unit edges are held back for assignment later in Step 3. Typically, these remaining edges are about 10-25% of the original unit edges (or 7-18% of all edges).

The output of Step 2, shown in Figure 5.3 (Step 2), is a modification to each level-node N_i in the level-graph, this being a vector (though shown pictorially in the figure) indicating the number of assigned fanout edges of each length that have been assigned to the level. Step 2 also guarantees that the assignment has met the minimum in and out degree requirements for each level.

Step 3: Partition the total fanout at each level (degree.c).

We have the vectors assigned_in[i], assigned_out[i], max_in[i] and max_out[i]. However, the assigned out-degree is a *total* for the level, not a list of individual node values from the fanout distribution.

In this step we partition the total out-degree (e.g. 10) of level i into n_i (e.g. 4) individual values taken from the fanouts distribution (e.g. {4, 3, 2, 1}, summing to 10).

First calculate *target* fanouts, target[i], $i = 0..d-1$, in the range assigned_out[i] to

chosen, the probability of the second also goes to zero. We implement this sometimes by direct calculation, and sometimes by re-smoothing the distribution to a given sum. This basic method is used, with different objectives, throughout the algorithm.

max_out[i], such that $\sum_{i=0}^{d}$ target[i] = n_{edges}. Again, we sample a probability distribution calculated as in Step 2(b), rather than performing a deterministic allocation. The goal is to assign the target out-degrees which are, on average, proportional to the amount of slack between the minimum and maximum fanout values for each level, but probabilistically rather than in exact proportion so that the resulting circuit is different with each execution of GEN with the same inputs.

We are left with the problem of partitioning each *target*[i] into n_i values taken from the fanout distribution. Even for a single level, this integer partitioning problem is NP-complete [29, page 223] to compute exactly, so we can only manage a heuristic solution. Fortunately, this is made easier because of the remaining unassigned unit-edges—target[i] is flexible within the range min_out[i] to max_out[i], so we typically need only an *approximate* integer partition for each level, and can allocate the remaining unit edges as required to make the result exact.

Before entering the main operation of the degree-allocation step, we examine the low fanout levels, defined as levels which have a total fanout less than $2n_i$. Assigning a high-fanout value to such a level could result in later difficulties as we "run out" of edges for giving individual nodes at least one fanout. To dispose of these levels, assign fanouts of 0, 1, and 2 deterministically, based on the availability of fanout-0 values in the fanout set (some, but not all PO nodes will have fanout 0).

The main operation of this step is probabilistic and iterative. For each level, compute average_out[i]=target[i]/n_i, and the values min_possible_out[i] and max_possible_out[i] indicating the degrees which could feasibly be assigned to any node at level i (using the rules of Step 1 applied to individual nodes). Then iterate through the values in the fanout distribution F from largest to smallest (the largest being usually the more restrictive, hence more difficult to place). Among the levels that can accept the current fanout f_j (based on min_possible_out and max_possible_out) we sample average_out[i] as a probability distribution (with the same goals as just mentioned for targets) to choose the level to which f_j will be assigned. (See the footnote in Step 3 for more detail on probabilistic sampling.) Each time we update the status vectors (assigned_out, available_out, average_out, minimum_fanout, maximum_fanout, min_possible_fanout and max_possible_fanout) for the chosen level.

Because of the probabilistic assignment, some levels will receive more than the target number of edges (based on the sum of their fanouts) and some will receive fewer. However,

the details of the assignment do guarantee that all levels will receive between their minimum and maximum total fanout. We also note that we do not always return the *exact* fanout distribution that is given to us, but the differences are very minor.

On the relatively rare occasion that a fanout cannot be accepted by any level, we decrement the fanout value by 1 and continue. This can lead to a minor modification of the input specification, as discussed further in Section 5.4.1.

At the completion of Step 3, all edges have been assigned to levels, and the level-node for each level i contains a list of edges (and their length) which leave that level, and a list of n_i fanout values f_{ij}, $j=1..n_i$, which sum to the total fanout of the level. Figure 5.3 illustrates this situation: the breakdown of total fanout into an (unordered) set of out-degrees is shown above Step 3, and the edge-length distribution is as in Step 2. (Unfortunately, to get an edge-length distribution which differs from Steps 2 to 3, we would need to use $k > 2$ and a larger n, which would make the main operation of the algorithm more difficult to view.)

Step 4: Split levels into nodes (nodes.c).

For this step, levels are treated independently. We need to split each level-node N_i into n_i individual nodes, and assign each of these a fanout from the list of available fanouts f_{ij} now assigned to level i. This would be trivial, were it not for the necessity to introduce *locality* (clustering and local structure) into the resulting circuit, and so we first discuss how we impose locality in the generation.

Our approach to introducing locality into the generation algorithm is to impose an ordering on the nodes at each level, and use proximity within this ordering between nodes at different levels as a metric of locality when we later choose the edge-connections between nodes. This can also be viewed as trying to generate graphs which will "look good" when displayed as pictures such as Figure 5.1, because minimization of edge lengths in a graph drawing also has the effect of reducing crossings and of displaying any inherent locality in the graph [30]—by creating a circuit with one known good ordering/drawing we have simulated this form of locality in the generation. The ordering we will use is simply the sorted order within the linear list of nodes within each level (this ordering is arbitrary until we have associated distinguishing features such as fanout or edges to the individual nodes). The measure of goodness of an edge is then the distance between the source and destination nodes in their levels node-lists, relative to competitors. As a result, the order in which

fanouts are assigned within the node list becomes important, because placing high-fanout nodes in an unbalanced way into the node list will skew the effects of locality measurement in Step 5.

The locality *index* assigned to each of the n_i nodes in the nodelist for level i is a scaled proportion of the maximum sized level. Thus if the level with the largest number of nodes contains 100 nodes, and the current level 10 nodes, then the latter will have nodes at locality indices 5, 15, 25, ..., 95. Before fanout allocation the order of nodes is arbitrary, so the nodes are now indistinguishable other than for this index.

Our goal in assigning fanouts to nodes in the list is to distribute the high fanout nodes well for maximum locality generation. To do this, we sample a *binary tree distribution* to allocate fanouts, in order from the highest to lowest fanout. To calculate the distribution, label the nodes of a balanced binary tree on n_i nodes with the number of leaves in its subtree. Then perform an inorder traversal of the tree, and place the labels in (probability density function) pdf[i], $i = 1..n_i$. For example, the binary tree pdf of length 15 is [1,2,1,4,1,2,1,8,1,2,1,4,1,2,1]. In the most likely case, then, the highest fanout node would be assigned in the middle, the next two highest fanouts at the quartiles, and so on. Another way to view this distribution is to take an ordered list of n_i nodes, assign a value p to the middle node $n_i/2$, a value $p/2$ to the nodes $n_i/4$ and $3n_i/4$, $p/8$ to the middle nodes in the resulting ranges and so on, then scale the resulting distribution to integers. The point of this operation is to (on average) place the highest fanout node in the middle of the ordering, the next two highest fanout nodes at the quartile points, and so on. Again, probabilistic sampling means that we don't get exactly the same result each time, and just as importantly, that we don't generate artificially symmetric circuits.

This step in the algorithm assigns to each node x_j in level i, a value fanout(x_j) from $\{f_{ij}\}$ and a value index(x_j) to each x_j, $j = 1..n_i$. A further calculation assigns p_j, $0 \leq p_j \leq f_j$, the *long-edge fanout* of node x_j, defined as the number of edges of length greater than one from x_j[2]. This is again probabilistic, sampled uniformly over all long out-edges in the level.

At the conclusion of Step 4, each node x in the circuit has an assigned delay, fanout, long-fanout and index, but no actual edges have been assigned between nodes at different levels in the graph. The fanout values are shown in Figure 5.3 (Step 4). This information, plus the edge-length assignments elsewhere in the figure comprise the input to Step 5 of the

[2]There are not enough long edges to warrant storing a vector of lengths

algorithm.

Step 5: Assign edges to nodes (edges.c).

The major remaining step is to connect the fanout edges on each node to a corresponding input port on a node on a later delay level, as specified by the edge-length. We proceed from level 1 to level d, connecting the edges to each level i.

To connect the in-edges to level i, we first calculate the source list, of unconnected edges preceding level i which are of the correct length to connect to level i. Nodes with multiple fanouts are inserted only once in the list, and nodes are deleted as their fanout is exhausted. The destination list consists of all nodes at level i. Both these lists are maintained in sorted order by node index (defined in Step 4).

Step 5(a). If the size (in *edges*) of the source list is more than twice the number of available *nodes* in the destination list, we pre process the high-fanout nodes (those with fanout more than 1/8 the number of nodes in the destination list) separately. To process a single high-fanout node x, we randomly choose a range of nodes of size between fanout(x) and 3·fanout(x)/2, centered at the closest index node y in the destination list to index(x). Choosing a random set of fanout(x) nodes from this set, we make the physical edge connections, and update all status vectors. This process is repeated for all high-fanout nodes in the source list. The purpose of this step is to avoid a situation where we have a large number of out-edges from the same source node x later in the edge-assignment phase which cannot be assigned without creating double connections from node x to some node y—this would otherwise be common because of the greedy nature of the algorithm.

Step 5(b). Establish combinational delay by connecting each node in the destination list which does not already have a fanin edge from 5(a) to one node from the source list. To choose the fanin for node y, we sample the source list L times, where L is the locality parameter of generation (discussed below), choosing the result x with the closest index to index(y). For this step, even though long-edge candidates exist in the source list, only source-nodes at the preceding combinational delay level are considered.

Step 5(c). Perform a second sweep similar to 5(b) (including locality) to ensure that each node y in the destination list receives a second incoming edge. There is no longer a restriction on the length of the edge, but we cannot choose the same fanin as is already attached to y from step 5(b).

Step 5(d). Now that the minimum requirements are met for each node in the destination list, iteratively choose a random node from the destination list, and choose an input from the source list as per 5(b) and (c). Continue until the source and destination lists are exhausted.

At the conclusion of Step 5, the circuit is complete, except that we may have fewer out-degree zero nodes than the required number of primary outputs. We postprocess the circuit to (randomly) label the required number of additional LUT nodes as primary outputs.

The final result of the generation algorithm (for one random seed) on the progression of Figure 5.3 from the original specification is the original example of Figure 5.1.

5.2.2 The Locality Parameter.

The locality parameter L has not been formally discussed to this point. As mentioned in Step 4, we find that a purely random connection of edges between levels does not model the type of clustering found in real circuits. At the same time, deterministically connecting the edges based on aligning index values yields a circuit which is overly local, and is actually too easy to place and route. We find that a reasonable approach in practice is to define a locality parameter L, and use it to bias the above algorithm towards greater locality; when choosing an input for a given destination node, we sample L times, and choose the source node which is closest in index value to the destination node under consideration. For higher values of L, the probability of directly lining up indices increases; for L=1, the algorithm is as originally described.

Though L can be specified as a user parameter to generation it does not tie directly to the characterization of a circuit. That is, we have no way to measure it for a specific given circuit. Through experimentation, we have found that there is no constant locality parameter which yields the correct results for all circuits (independent of size), but a value which scales logarithmically with the size, n, of the circuit yields good results. Outside of n, L is unrelated to the other input parameters of the circuit.

We find that the locality parameter can significantly affect the properties of the resulting circuit. Though we can empirically do very well at generating circuits simply by varying the relationship between L and n, it would be better to tie locality to characterization, particularly when dealing with generation of "clone" circuits.

Improved Locality Generation.

In order to improve the generation of locality in circuits, we have been pursuing work to reparameterize Step 5 of the GEN algorithm to use the *spread* and *span* metrics defined in Section 3.5 rather than L.

Algorithmically, this does not significantly change Step 5. Using *spread* we assign x coordinates for each node u within the allowable range. Using the average span for the level, we stochastically choose a span for each node u, and attempt to choose the previous level edge connections to u to realize this actual span.

To this point, we have not been able to improve on our generated circuits by taking the new locality information into account. We have several theories on this, and on what further characterization is required, and we will discuss the issue further in 6.3.

5.3 Sequential Circuit Generation.

In this section, we discuss how to generate sequential circuits.

As per the model of Chapter 4, we define a sequential circuit as a hierarchy of combinational subcircuits which are connected together with FF-edges and back-edges. In that characterization, we decomposed a sequential circuit into its combinational components, introducing ghost input and output ports. Here we pass new information about the GI and GO interface into the subcircuit generation, then "glue" the subcircuits together to form a complete sequential circuit. The final circuit will have no ghost inputs or outputs, as they will have all been glued together into back-edges (a ghost output connected to a ghost input at a preceding sequential level) or FF-edges (a ghost output connected to a flip-flop at the immediately next level). As mentioned in Section 4.2.3 the model and algorithm actually generalize to arbitrary forms of hierarchy, given the appropriate parameterization, but here we will talk only about simple sequential circuits with a single level of hierarchy.

Though the hierarchy and locality in a sequential circuit are partly captured by the number of ghost inputs and ghost outputs between subcircuits, it is also very important to know the *shape* of these connections. This is because we want to retain the combinational delay of nodes as defined in the subcircuits, so we can only connect a ghost output to a ghost input if the GO is either has a lower combinational delay or the GI is a flip-flop. Define the vector *GIshape[d]* as the number of ghost inputs at combinational delay d, $d = 0..max\_delay$,

and $GOshape[d]$ similarly for ghost outputs. These will introduce a topological constraint on the connections between different subcircuits in addition to simply specifying the number of connections. In practice, we find that these vectors are important, especially for generating clones, because they often uncover "quirky" aspects of different circuits. Note that the GIshape for one level and the GOshape for the other level in a 2-level circuit will roughly correspond, but will not usually be exact—for MCNC circuits, there is typically some slack between the combinational delay of the endpoints. It is crucial to have compatible GI and GO shape vectors between different levels, or the algorithm is forced either to create an inordinate number of long edges, or to introduce extra flip-flops in order to resolve GI and GO at incompatible delay levels.

To describe the sequential algorithm, we need to address three issues: how to exactly parameterize a sequential circuit and its subcircuits; the modifications required to the combinational algorithm to accommodate new parameters; and the gluing algorithm for creating the final circuit from the subcircuits. These are covered, respectively, in the next three sections.

5.3.1 Sequential Circuit Parameterization

A sequential circuit is parameterized by *levels* (the number of sequential levels), n_{DFF} (flip-flops), n_{back} (back-edges), n_{PI} and n_{PO}, its sequential shape (the number of nodes at each sequential level), and the parameterizations of its combinational subcircuits.

Adding to the parameterization of combinational circuits, we have n_{GI}, n_{GO}, n_{latch} (the number of GO designated for FF-edges), *level* (the sequential level for this subcircuit), and the vectors $GIshape[i]$ and $GOshape[i]$, $i = 0..d$.

In a fully specified parameterization, the combined information in the subcircuit specifications completely determines the circuit, so values like n_{DFF} and n_{back} are redundant. If the subcircuit parameterizations are determined by the default parameterizations (i.e. fsm_circ in Appendix A) then that high-level information is used to generate, for example, compatible ghost I/O shapes before generation begins.

The definitions are best understood with an example. Figures 5.4(a) and 5.4(b) represent combinational subcircuits which will be glued together into the complete sequential circuit shown in Figure 5.4(c). The subcircuit in Figure 5.4(a) has parameterization[3] $\{n=7, level=$

[3]Note that these are partial parameter lists only, as some parameters not relevant to the current discussion

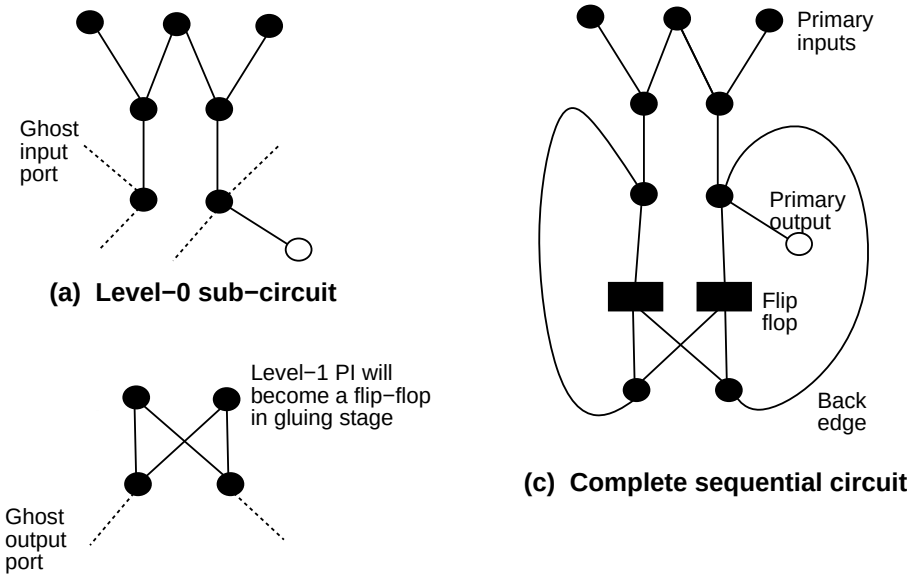


Figure 5.4: Example construction of a 2-level sequential circuit.

$0, n_{PI} = 3, n_{PO} = 1, n_{edges} = 6, n_{GI} = 2, n_{GO} = 2, n_{latch} = 2, shape = (3,2,2), GIshape = (0,0,2), GOshape = (0,0,2)\}$. The circuit in Figure 5.4(b) has $\{n = 4, level = 1, n_{PI} = 2, n_{GI} = 0, n_{GO} = 2, GOshape = (0,2), n_{PO} = 0, n_{latch} = 0\}$. The complete circuit is described by $\{n = 11, n_{PI} = 3, n_{PO} = 1, levels = 2, n_{DFF} = 2, n_{back} = 2\}$ in addition to the specification of its subcircuits. Note that the flip-flops serve as primary inputs in the specification of the subcircuit at level 1, but primary inputs cannot exist at levels greater than zero (by definition) in the final circuit, so these are converted to flip-flops as they are glued to ghost outputs from the previous level. Notice how the GOshape of level one is, when shifted right by one, equal to GIshape of level zero. In practice the shifted GOshape is lexicographically less than or equal to the GIshape when looking at back-edges.

5.3.2 Changes to the Combinational Algorithm.

To generate subcircuits, we use a modification of the original combinational algorithm of Section 5.2. The additional constraints in the model implied by n_{GI}, n_{GO}, n_{latch}, *GIshape*, and *GOshape* necessitate changes throughout the algorithm, as they change the ratio of nodes to edges, introduce nodes with no fanout, and nodes with fanin of one when ghost inputs are present.

of sequential circuits are left out.

Identifying Ghost Outputs (Step 1).

One of our primary applications is to generate circuits which are good inputs for FPGA tools. The typical logic block configuration in an FPGA is a 4-input LUT followed by a flip-flop. The output signal from the LUT can either be registered through the flip-flop, or not. Thus any LUT we generate which has both a registered and unregistered output will require *two* FPGA logic blocks in technology mapping, increasing the size of the circuit to the place and route tool and ruining our ability to compare circuits on the basis of routability. Simple experiments show that about 90% of the LUTs which feed a flip-flop in real circuits have no other outputs so we want to, wherever possible, assign fanout values of 0 to nodes which will have a single ghost output destined for a FF-edge.

To accomplish this goal, we identify the delay location of the n_{latch} ghost outputs which will eventually feed a flip-flop in Step 1. This allows us to take them into account during the degree allocation phase. The result of this calculation is to make a new vector latch_shape[i], $i = 0..d$, available to the degree calculations of Step 1.

We also point out that any LUT which feeds a flip-flop will also feed only one flip-flop, since it (usually) makes no sense to register the same signal twice.

Degree Allocation (Step 1).

Recall that Step 1 of the combinational algorithm calculates bounds on the maximum and minimum fanin and fanout of each combinational delay level. The distribution of GI and GO ports affects this process in several ways.

1. We assume that latch_shape[i] nodes at level i will have a minimum fanout of zero, rather than one (as per the above discussion).

2. We allow (but don't require) shape[i] - GIshape[i] nodes at level i to have minimum fanin one rather than two. Note that we must still allocate at least one "real" fanin for each node, or it would not (by definition) be in this subcircuit.

3. We subtract GIshape[i] nodes from the maximum fanin of level i, to leave room for the incoming back-edges.

In addition to these specific changes to degree allocation, there are a significant number of minor modifications required in the details of the probabilistic sampling. This is mainly

because the loss of 20-50% of the edges in the specification (to GIs and GOs) results in a more restricted and difficult problem.

Fanout Assignment (Step 3).

Step 3 of the algorithm, which assigns actual values from the fanout distribution to delay levels, takes into account latch_shape in the allocation of zero-fanout nodes, as per the above discussion. The number of fanout-0 nodes for any level is bounded by GOshape[i] + POShape[i].

Ghost I/O Assignment (Step 4).

Recall that Step 4 of the combinational algorithm creates the nodes, and assigns their fanout values. Previous changes have tried to "make room" for the ghost I/Os, and here we actually allocate GI and GO ports to individual nodes.

The allocation of ghost inputs is straightforward: we allocate the GIshape[i] ghost inputs randomly and uniformly to the nodes at delay level i. Looking at the data for real circuits, we find that there is no statistical reason to do otherwise.

We designate latch_shape[i] nodes as *latched*. These nodes will eventually be candidates for gluing to a flip-flop. As much as possible, these will be fanout-0 nodes, and will not be assigned additional GOs. If there are remaining fanout-0 nodes after this step, we assign additional GOs. All remaining GOs are kept for a new post-processing step discussed next.

Remaining GO assignment (new Step 6).

Sequential subcircuits usually have fewer available edges than fully combinational circuits, so we use the ghost outputs, in part, to "repair" any extra zero-fanout nodes which may exist (usually some, but a small proportion) on the delay level they are assigned to. The remaining ghost outputs are not assigned uniformly. We want to generate more realistic circuits which tend to have a smaller number of high-fanout nodes to previous levels, rather than many nodes with a single ghost output. To do this, we choose a random subset of the nodes on each delay level requiring ghost outputs, smaller than the number of ghost outputs available, then assign the ghost outputs uniformly to nodes in the subset.

These modifications to the combinational algorithm allow us to generate a combinational circuit with the correct number of ghost inputs and outputs at the required combinational

delay levels so that the gluing process can take multiple circuits and glue them together.

5.3.3 Gluing Subcircuits.

The problem of joining subcircuits together into the final sequential circuit C is essentially one of appropriately matching the ghost ports between the subcircuits into back-edges and FF-edges.

When gluing begins, we have a list of subcircuits C_i, $i = 1..c$ to be connected, sorted by increasing sequential level. Each subcircuit contains a list *GIlist* of ghost inputs, a list *FF_outlist* of ghost outputs which have been labeled as targeting a flip-flop (from n_{latch} in the specification), a list *GOlist* of other ghost outputs intended for back-edges and a list *FF_inlist* of primary inputs in subcircuits at non zero sequential levels which will become flip-flops. Each ghost input and output is attached to a node in the subcircuit, and inherits the combinational delay of that node.

The matching is constrained by combinational delay and sequential levels. We cannot join a node x at sequential level l to a node y at level $l + 1$, unless y is a PI (i.e. intended to become a flip-flop). We also cannot join a node x to *any* node y at a level beyond $l + 1$ without violating the definition of sequential level on the nodes of C. Similarly, we cannot join a ghost output on a node x to a ghost input on a node y if $d(x) \geq d(y)$, without violating the combinational delay of y, and we cannot connect two ghost outputs attached to x with two ghost inputs to y, or we create a duplicate fanin to y.

This problem reduces to a standard bipartite matching problem and there are known exact algorithms to solve it. However, the exact approaches are based on network-flow algorithms which are too slow (i.e. $O(n\sqrt{n})$ time) to allow us to generate large circuits. Furthermore, in order to apply the geometric locality heuristic used in combinational generation to gluing, and later to extend the gluing algorithm to one which does not find *all* connections, but leaves some ghost inputs and outputs disconnected (as would be desired for multi-level hierarchical generation) we would require weighted matching, which uses $O(n^2 \log n)$ time [66]. Since the other parts of GEN operate in either linear or $O(n \log n)$ time, this would not be acceptable.

Thus we approach the gluing problem heuristically with a greedy algorithm. The most important aspect of the operation is to properly order the connections so as to increase the chances of finding a good solution. A solution which fails to connect all possible edges will

result in GEN later having to diverge from its input-specification by creating extra flip-flops or by moving ghost inputs or outputs to different nodes.

Because registered ghost outputs are labeled separately from the other ghost outputs, the problems of gluing back-edges and gluing FF-edges are independent. However, different subcircuits do "compete" for back-edges. We give priority to earlier sequential levels by processing in the following order (justified later):

for $i = 0..c$ /* c is the number of subcircuits */
    connect back-edges from C_j, $j \neq i$, to GIs of C_i.
    connect FF-edges from registered GO nodes in C_i to PIs in C_{i+1}
end for

Locality of connection.

We have previously discussed the locality metric in making combinational connections between nodes in Step 5. For sequential gluing, define the *index* of a node as an integer proportional to the node's location in the node list for a given delay level in any subcircuit (the $0..n_i - 1$ ordering of the n_i nodes in delay level i, scaled to the maximum width over all combinational levels). When edges are connected in Step 5 of the base algorithm, we probabilistically favour connections between nodes which have closer indices, in order to introduce clustering in the circuit. This form of geometric clustering is evident when viewing pictures of circuits generated by heuristic graph-drawing packages such as DOT [30] (e.g. see the many drawings in Chapter 6).

In order to generate realistic circuits it is important to continue this process when connecting nodes to flip-flops and back-edges, or we generate circuits with many crossing edges which are overly difficult to place and route. Thus, we continue to use the node index for sequential gluing.

Gluing back-edges.

The algorithm for gluing back edges to the ghost inputs of one circuit C_i from all other subcircuits is as follows.

First create a destination list of all ghost inputs in C_i and a source list of all ghost outputs in the other subcircuits which are at later sequential levels. Sort both lists by

increasing *index* within decreasing *delay*. The purpose of this order is to use up the highest delay ghost outputs first (because they are more likely to not find a matching ghost input and then require a flip-flop or movement later), and to match them to the highest delay ghost inputs with which they are compatible. Given that, we want to match indices as well as possible.

Now proceed through the source list in order. Define the *match value* of a source node x with a destination node y as ∞ if (x, y) is an invalid edge (by the constraints above), and $d(y) - d(x)$ otherwise. We search the destination list for the first node with lowest match value, which also lines up a compatible index by the sorting. Note that we don't actually have to look at the entire destination list; this can be done in $O(d)$ time, using a few additional pointers indexed into the destination list. Combinational delay d is essentially a constant so the algorithm is fast.

The time required for this gluing phase is dominated by the sorting, so we need $O(n \log n)$ time[4] per subcircuit, of which there are a constant number. Note that "n" in the algorithmic complexity refers to the number of back-edges in C, which is typically about 5-10% of the size of the whole circuit[5].

The reason that the main algorithm processes subcircuits in order of their sequential level is that the earlier levels typically have both many more nodes and greater combinational delay, and also a more complex overall structure. (Later levels often reduce to a register-file with only a couple of logic nodes.)

Gluing Edges to Flip-Flops.

The process for gluing nodes with ghost outputs labeled as latches to primary inputs at the next sequential level is more straightforward. For each adjacent pair of levels, create a source and destination list as before, sort the lists by index (independent of delay), and line up nodes directly (the lists are the same size, by the original specification of the subcircuits). This is an additive factor of $O(n \log n)$ time to the preceding steps, so the entire gluing algorithm remains $O(n \log n)$ time. (In this case, n refers to the number of flip-flops in the circuit which is, in practice, not the entire size of the circuit.)

[4]Due to the fact that the node lists are already sorted, we can reduce this to an $O(n \cdot d)$ algorithm with appropriate data structures. However, given the tight constants which exist for sorting algorithms, we believe the constant for doing this would dominate $\log n$ for all reasonable n, so it is not of practical interest to do so. The same applies to most (but not all) sorts which occur in GEN.

[5]This doesn't change the abstract complexity, but the algorithm runs faster in practice.

Note that the order in which subcircuits are considered is unimportant, as the connections are independent.

Post-processing.

As mentioned earlier, it is not always the case that a perfect matching exists for the back-edges. A post-processing step is necessary to resolve the remaining incompatible ghost inputs and ghost outputs. In this step ghost inputs and outputs are moved to suitable candidates elsewhere in the subcircuits until matches are found. In extreme cases (flagged by warnings from GEN) up to 40% of back-edges can be unresolved before post-processing, but typically only 0-5% of ghost inputs and outputs (which comprise less than 1% of all edges) remain after the main gluing algorithm.

5.4 Implementation Details.

5.4.1 Meeting the Input Specification.

It is not always the case that GEN determines a circuit which meets the input specification. As with any heuristic algorithm, there exist input possibilities for which the heuristics fail. In the case of GEN, we find that we are occasionally (1-2% of the time) unable to complete a valid circuit. In these cases, the tool reports a "failure to determine a circuit with this specification." About 2-3% of the time, GEN will complete a circuit, but will report that it was forced to modify the input specification significantly in order to finish (though this is necessarily minor enough to not warrant failure). We consider these to be minor problems, because the user can run the tool again with a new random seed, and typically will get an acceptable output on the second try.

5.4.2 Parameterization and Default Scripts.

The discussion to this point has involved the generation of a circuit with a completely specified *exact* specification. In practice, the user would choose only a small number of parameters (or possibly just n), and the remaining are chosen from default parameter distributions.

GEN is augmented with a sophisticated C-like language, SYMPLE, for parameter generation. The default distributions are written in this language, and the user can specify

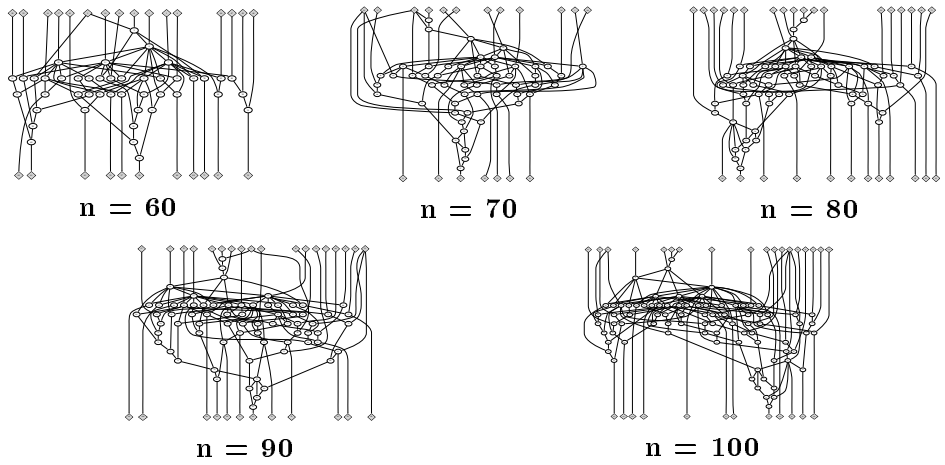


Figure 5.5: A GEN circuit family ({`k=2; n=60..100 by 10`}).

modifications in the control script for a circuit. SYMPLE provides a great deal of control over parameters. The complete default scripts for combinational circuits, defaults.gen, comb.gen, fsm.gen and special.gen are are shown in Appendix A, along with a description of SYMPLE. As an example, observe how n_{IO} is currently defined as a set of piecewise Rent-like equations, each of which has the Rent parameter drawn from a Gaussian distribution (see the IOFrame of comb.gen).

The current default sets and parameters have been determined from experimentation with the MCNC benchmark circuits. It would be possible to perform the same experimentation with an alternate set of benchmarks, and generate a modified default script.

SYMPLE allows parameters to be specified as constants, drawn from statistical distributions or chosen as functions of other parameters. Figure 5.5 shows a series of circuits generated with the varying n but other parameters fixed, to generate a *family* of related circuits. SYMPLE scales related parameters (e.g. depth and shape) yet retains the similarity of other properties. This ability to scale circuits while retaining fundamental similarities introduces an entirely new paradigm for evaluating the scalability of architectures and algorithms.

5.4.3 Input Scripts and Clone Circuits.

The input to GEN takes basically two forms. The user can specify a parameterization which they create themselves, use CIRC to extract a parameterization from an existing circuit, then generate a *clone* circuit with the same properties, or do a mixture of the two by modifying

```
/* CIRC 3.0, compiled Tue Oct  1 14:30:51 EDT 1996.
 */
X = comb_circ {
        name="alu4clone";
        n=1536; kin=4; nPI=14; nPO=8; delay=7;
        nEdges=5400; edges=(0, 4494, 757, 125, 23, 1, 0, 0);
        shape=(14, 692, 518, 198, 80, 21, 11, 2);
        outs=(8,1267,67,41,32,33,14,13,11,3,2,9,9,5,4,0,0,1,1,0,
               1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,1,2,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,3,0,1,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1);
        max_out=249; nZeros=8;
};
output(circuit(X));
```

Figure 5.6: A GEN clone script for the MCNC circuit **alu4**, output by CIRC.

a clone script.

Figure 5.6 shows the second case, in the form of a GEN script output from CIRC given the MCNC circuit **alu4**[6]. The object "comb_circ" referred in the script to is the default frame in the script comb.gen, and the specifications inside the set brackets indicate modifications to parameters in comb_circ which override the defaults.

Figure 5.7, in contrast, shows a user-defined GEN script to create a 1,000 LUT circuit. The user has chosen simply to generate a combinational circuit with 1,000 LUTs, 58 PI and 16 PO with combinational delay 9. The remaining, unspecified, parameters (shape, edges, etc...) are chosen from default distributions which use the specified circuit parameters such as delay and n_{PI} as input parameters themselves.

To visualize the operation of GEN for sequential circuits, and to see the type of variation that can occur in generating a clone, Figures 5.8 and 5.9 show the clone script produced by CIRC for the sequential circuit **bbtas**, the circuit itself, and two clones produced by GEN given the clone script. Note that we use node labels rather than the actual back-edges to

[6]The command line used to generate this clone script is "`circ in=alu4 k=4 gen`." This takes the 4-LUT mapped circuit alu4.blif from the default MCNC directory with $k = 4$, and produces the GEN script pictured.

```
X = comb_circ { name="X"; n=1000; nPI=58; nPO=16; delay=9; };
output(circuit(X));
```

Figure 5.7: A simple user-generated GEN script for a 1000 LUT circuit.

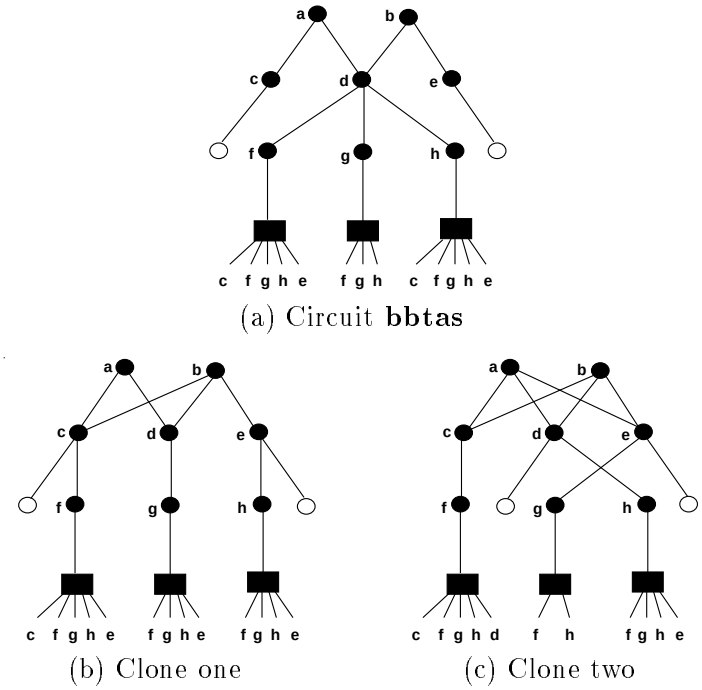
```
/* CIRC 3.1, compiled Wed Aug 28 15:36:17 PDT 1996. */
X = {
    name="bbtasclone";
    L0=(@.comb_circ) {
        name="L0";
        n=8; kin=4; nPI=2; nDFF=0; level=0; delay=2;
        shape=(2,3,3);
        nEdges=7; edges=(0,7,0);
        nGI=13; GIshape=(4,9,0);
        nGO=3; GOshape=(0,0,3);
        nPO=2; POshape=(0,2,0);
        outs=(5,0,2,1);
        max_out=3; nZeros=5, nBot=3;
    };
    L1=(@.comb_circ) {
        name="L1";
        n=3; kin=4; nPI=0; nDFF=3; level=1; delay=0;
        shape=(3);
        nEdges=0; edges=(0);
        nGI=0; GIshape=(0);
        nGO=13; GOshape=(13);
        nPO=0; POshape=(0);
        outs=(3);
        max_out=0; nZeros=3; nBot=3;
    };
    glue=(L0, L1);
};
output(circuit(X));
```

Figure 5.8: Clone script, produced by CIRC for **bbtas**.

improve readability.

One aspect that the parameterization does not necessarily capture is the *symmetry* of the original circuit. We observe that neither clone has the symmetry of the original. Note, however, that recapturing the block structure and symmetry in a flat netlist are open (and very difficult) research problems of their own.

We point out, as well, that the two clones are different, yet both respect the parameter-



(a) Circuit **bbtas**

(b) Clone one

(c) Clone two

Figure 5.9: The MCNC sequential circuit **bbtas** and two clones.

ization of the input script. One of the features of the implementation is that the user can generate multiple different circuits with the same underlying specification.

5.4.4 Time Complexity of the GEN Algorithm.

The theoretical time complexity of the algorithm and its GEN implementation is the larger of $O(d^2)$ from Step 1 and $O(n \log n)$ from each other step. In practice, we assume that $d << n$, so the complexity reduces to $O(n \log n)$. Each step in the algorithm addresses each element a constant number of times in processing for a linear factor, with possible constant number of preprocessing sorts or the creations of a random permutation, each of which takes $O(n \log n)$ time. The algorithm uses a constant amount of space per node, hence $O(n)$ for the algorithm.

In practice GEN is very fast. Generation of a 2,000 LUT circuit takes about 7 seconds on a Sparc-5, using 500K of memory. For perspective, the same circuit requires about 45 minutes and 2M of memory to place and route using even a fast and memory-efficient tool such as VPR. A circuit of 30,000 LUTs (beyond the size of current FPGAs) requires about 30 seconds and 1M to generate, versus a half-day or more to place and route.

We have successfully generated circuits of up to 200,000 LUTs, well beyond the level of current FPGAs. The GEN implementation is currently limited to about that size, due simply to the use of 32 bit integers: we need to be able to calculate n^2 to determine some probability distributions. Larger circuits would require special purpose arithmetic, at least for specific parts of the code, or a hierarchical approach to generation.