Generation of Synthetic Sequential Benchmark Circuits*

Michael Hutton†, Jonathan Rose‡ and Derek Corneil†
University of Toronto

Abstract

Programmable logic architectures increase in capacity before commercial circuits are designed for them, yielding a distinct problem for FPGA vendors: how to test and evaluate the effectiveness of new architectures and software. Benchmark circuits are a precious commodity, and often cannot be found at the correct granularity, or in the desired quantity.

In previous work, we have defined important physical characteristics of combinational circuits. We presented a tool (CIRC) to extract them, and gave an algorithm and tool (GEN) which generates random circuits, parameterized by those characteristics or by a realistic set of defaults. Though a promising first step, only a small portion of real circuits are fully combinational.

In this paper we extend the effort to model sequential circuits. We propose new characteristics and generate circuits which are sequential. This allows for the generation of truly useful benchmark circuits, both at and beyond the sizes of next-generation FPGAs. By comparing the post-layout properties of the generated circuits with already existing circuits, we demonstrate that the synthetic circuits are much more realistic than random graphs with the same number of nodes, edges and I/Os.

1 Introduction

In an ideal world, an FPGA vendor would use hundreds of benchmark circuits in determining the architecture of a next generation device, as well as developing the associated automatic placement and routing software for it. In this way, the architectural design space would be adequately explored and the best software algorithms would be used and well-tested.

However, because the part is new, there are few designs available at the correct granularity and size to perform this kind of exploration. Some circuits will always exist via customer migration from gate-arrays, synthesis from high-level design languages, or through various other means, but these rarely suffice and companies are forced to purchase benchmarks or to expend considerable effort creating them internally.

There exist alternatives to using "real" benchmarks of the desired size. The PREP benchmark set [8] places a number of disconnected copies of the same small circuit into one netlist. Random graphs are another possibility but we have demonstrated [7] that random graphs are too unrealistic.

*Supported by NSERC Canada and a grant from Hewlett Packard. Departments of Computer Science† and Electrical and Computer Engineering‡, University of Toronto, Ontario M5S 3G4. {mdhutton@cs,jayar@eecg,dgc@cs}.toronto.edu.

In previous work we addressed the problem of random generation of *combinational* circuits [7]. We defined properties such as size, delay, physical shape, edge-length distribution, fanout distribution and reconvergence to describe the physical characteristics of a purely combinational circuit after the technology mapping stage. A public-domain tool, CIRC, was developed to extract these parameters. We gave an algorithm to randomly generate a circuit with an exact parameterization, and presented another tool, GEN, which implemented it. By comparing characteristics of the generated circuit that were not specified as parameters to generation (post-placement wire-length and track-count and a quantification of reconvergence) we showed that the generated circuits behaved very comparably to real circuits, whereas random circuits of the same size did not.

In a different approach to the generation problem, Darnauer and Dai [4] gave an algorithm for generating random undirected graphs to meet a given I/O ratio and Rent parameter, primarily aimed at a study of routability, and with applications to creating partitioning benchmarks. They showed the validity of their approach for relatively small combinational circuits, but it is not yet clear how successful it is for evaluating new architectures and place and route software or for larger or sequential circuits.

In this paper we address the problem of generating *sequential* circuits, i.e. circuits that contain flip-flops and directed cycles (broken by a flip-flop) in the logic. We expand on the combinational circuit characteristics by defining additional parameters for sequential circuits. The same approach can be applied to modeling and generating hierarchical circuits. Using the new parameters, we have made significant changes to the basic combinational algorithm to allow for the generation of these circuits, and added new aspects to deal with hierarchy. The tool is capable of quickly generating electrically valid and reasonable sequential benchmark circuits which can be read by commercial FPGA software.

To show that that these benchmarks are realistic, we use the approach illustrated in Figure 1. Given an industrial benchmark circuit, we use CIRC to extract its parameterization, and GEN to generate a *clone* circuit with the precisely specified set of characteristics. We also generate a random graph with the same number of nodes, edges and I/Os, but otherwise unconstrained by our characterization parameters. Then we place and route all three with an academic tool VPR [3], and with Altera Corporation's MAX+PLUS2 software. By comparing the post-placement and routing statistics for the original circuit and its clone, and contrasting this to the results for a random graph of the same size as the original circuit, we are able to show that our method generates circuits which significantly more realistic than random graphs.

Though this paper concentrates mostly on circuits taken from exact specifications, GEN also comes with a sophisticated set of defaults to generate circuits "from scratch," in which the only required parameter is the circuit size. We have developed

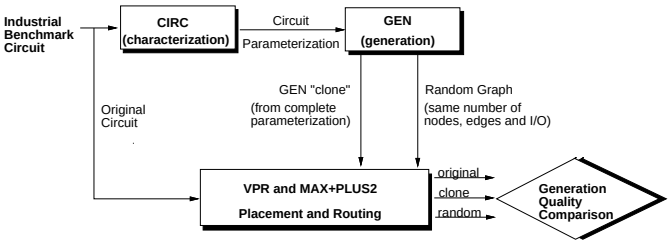


Figure 1: Approach to Circuit Generation

a *specification language* in which parameters can be chosen from various standard and new statistical distributions and provided *default* GEN-*scripts* which provide compatible values for any missing parameters in an input specification. The user can use these defaults, drawn from our experiments on MCNC circuits and personal experience with the tools, or program their own in the specification language.

In Section 2 we briefly review the characteristic definitions and the algorithm from [7] for generating combinational circuits. Section 3 describes the new sequential characteristics needed to model and generate sequential circuits hierarchically. The overall algorithm integrating combinational and sequential generation comprises Section 4. In Section 5 we discuss and validate the quality of the synthetic circuits by comparing their routability with industrial benchmark circuits and random graphs of the same size. We conclude in Section 6 and discuss extensions to the current prototype software.

2 Background: Combinational Circuits.

In this section we review the definitions, algorithm, and terminology of [7], which dealt with combinational circuits only.

2.1 Combinational circuit parameterization. We model a combinational circuit C by a series of scalar and vector parameters. Our base representation of a circuit is as a graph with nodes and 2-point connections (*edges*, as opposed to nets or hyper-edges). Define n_{PI} and n_{PO} as the number of primary inputs and outputs in C, and n_{LOG} as the number of logic nodes. Then n, the *size* of C, is $n_{PI} + n_{LOG}$ (we treat a PI as node type, but a PO as a property of a logic node). For any node x, $fanin(x)$ is the number of edges entering x. Similarly, fanout(x) is the number of edges leaving x and $max\_fanout(C)$ is MAX{fanout(x)} over all x in C. We assume that fanin(x) is always bounded by some constant k (typically 4), but that max_fanout(C) is bounded only by n. Defining $fanouts[i]$, i=0..max_fanout, as the number of nodes in C with fanout i, we have the *fanout distribution* of C. The number of edges n_{edges} in C is the sum, over all x in C, of fanin(x) (equivalently the sum of fanout(x)).

The remaining parameters are related to combinational delay. Each node x has a maximum combinational delay, defined by $d(x) = 0$ if x is a PI, otherwise $d(x) = 1 + \text{MAX}(d(y_i))$ over all inputs y_i to x. The combinational delay of C, $d(C)$ is the maximum $d(x)$ over all x in C. Define n_i as the number of nodes in C with combinational delay i. Then the *shape distribution* of C is $shape(C) = [n_i]$, $i = 0..d(C)$. For an edge $e = (x, y)$, define $length(e) = d(y) - d(x)$. Define e_i as the number of edges in C with length i, inducing the *edge-length distribution*, $edges(C) = [e_i]$, $i = 0..d(C)$. An edge of length-one is a *unit* edge and of any other length a *long* edge. In [7], we found that *shape* (similarly *edges*) does vary from circuit to circuit, but that there is a typical class of distributions which applies to most real circuits but differs from the shapes of random graphs.

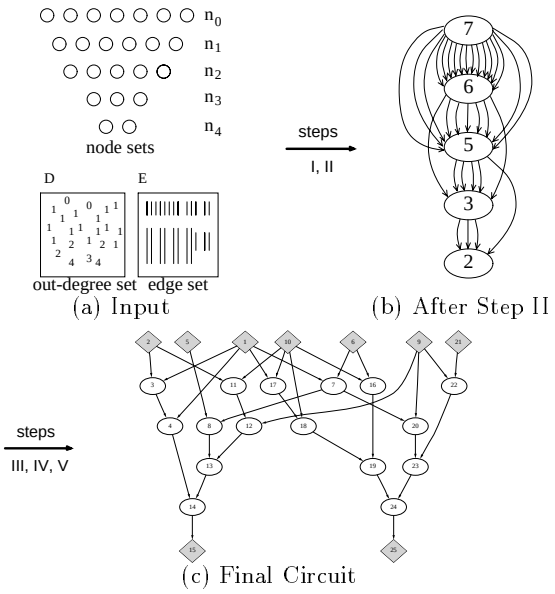


Figure 2: The Combinational Generation Algorithm.

2.2 Generating combinational circuits. Also in [7], we described an algorithm for generating combinational circuits from the list parameters described above: n, n_{PI}, n_{PO}, n_{edges}, k, max_fanout, max_delay and the $fanouts$, *shape*, and edge-length (*edges*) distributions. The algorithm will create a graph (netlist) on n nodes and n_{edges} edges, such that each node x is assigned one fanout value from the set represented by the $fanouts$, that assigned value corresponds to the actual fanout of x in the graph, combinational delay is well-defined for all nodes (i.e. d(y) < d(x) for all fanins y of x, and at least one fanin y_0 has d(y_0) = d(x)-1), fanin is bounded by k for all nodes, and all fanins to x are distinct (i.e. any signal enters a logic node at most once).

The algorithm for generating a combinational circuit is illustrated in Figure 2, and we give a brief overview of it here. The parameterization defines a set of disconnected nodes at each combinational delay level (Figure 2(a)), and sets of unassigned edges and fanouts. We initially consider all nodes on the same level as collapsed to a single *level-node*. Step I computes boundaries on the maximum and minimum in and out-degree of each level. Step II assigns the majority of edges between levels, yielding the intermediate representation shown in the Figure 2(b). Step III partitions the total out-degree of each level into n_i values chosen from $fanouts$. Step IV divides the level-nodes into individual nodes and assigns fanout values each. Step V connects edges (currently between levels) to nodes, and introduces some local clustering to the netlist. The overall algorithm yields a circuit as shown in Figure 2(c). In Section 4 we will overview the modifications necessary for sequential circuits.

The GEN system actually has two phases. The above algorithm describes the second phase, generation of a circuit from an *exact specification*. More typically, the user will specify only a few of the scalar parameters, and the front-end to GEN will create the remaining parameters from the *default scripts*

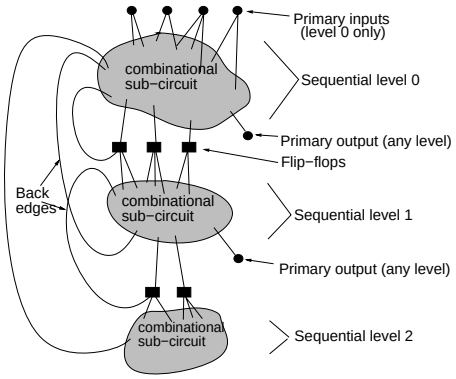


Figure 3: Abstract Model of a 3-Level Sequential Circuit

(mentioned earlier). The parameter selection phase will then complete the parameterization to generate an exact specification, and pass a complete description to the main algorithm.

2.3 Quality of combinational GEN-circuits. We demonstrated the validity of the combinational chararacterization and the generation algorithm using MCNC circuits and the process illustrated in Figure 1, as discussed in the Section 1 (though we used VPR alone for the combinational experiments.) We found that the post-placement wirelength of the GEN clone-circuits differed by 11%, on average, from the source circuit, whereas the corresponding random graph differed by 63% on average. Looking only at the largest 12 circuits, the numbers were 28% and 218%, respectively.

The major weakness of the first GEN was that it did not handle sequential circuits. In addition to being more difficult to generate, sequential circuits would be more difficult to place and route, because the underlying graph is more complex. The extensions to the model to deal with sequential circuits and to generate them comprise the remainder of this paper.

3 A Model For Sequential Circuits.

In order to generate sequential circuits, it is necessary to form a model of what we mean by a sequential circuit. We describe this as a hierarchy of two or more[1] combinational circuits connected by "flip-flop-edges" (*FF-edges*) and "back-edges," as shown in Figure 3. We assume that there are D-type flip-flops between combinational portions of a circuit, so all nodes are of type PI, logic or flip-flop. Recall that PO is a property of a logic node, not a separate node type. For simplicity, all flip-flops in a circuit will share a single global clock, which is not represented in the netlist.

Under this model, the definitions of primary input, primary output, and all fanout measures remain as before. However, combinational delay is modified so that any flip-flop node is at combinational delay 0, independent of its (single) input. The *sequential level*, $level(x)$ of node x is defined as 0 if x is a primary input, $1 + level(y)$ for a flip-flop x with input y, and MIN(level(y_i)) over all inputs y_i to x otherwise. Notice that all primary inputs must thus occur in sequential level 0. Define an edge (x, y) to be a *forward-edge* if $level(x) = level(y)$ and a *back-edge* if $level(x) > level(y)$. All other edges are necessarily *FF-edges* connecting a logic node to a flip-flop at the immediately next sequential level. The definition of edge-length is as before, even if the nodes are at different sequential levels, except that FF-edges are always of length one. The *size* of the circuit is now $n = n_{LOG} + n_{PI} + n_{DFF}$.

We define a sequential circuit as a hierarchy of combinational sub-circuits which are connected together with FF-edges and back-edges, as illustrated in Figure 4. To generate the interface where these sub-circuits are to be joined, we introduce *ghost input* (GI) and *ghost output* (GO) ports in each sub-circuit. These are reserved fanin (fanout) ports attached to logic nodes in the combinational circuit. Note that GI and GO ports correspond more closely to edges than to nodes, since a single node can have up to $k-1$ ghost inputs and up to *max_fanout* ghost outputs. The number of ghost outputs, n_{GO}, is divided into those which will eventually feed a flip-flop (n_{latch}) and the remainder, which will become the source of a back-edge. A final sequential 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[2]). Ghost outputs are assigned (in sub-circuit generation) such that nodes with a ghost output destined for FF-edge connection will have just that one ghost output[3].

Though the hierarchy and locality in a sequential circuit is partly captured by the number of ghost inputs and ghost outputs between sub-circuits, it is also important to describe the *shape* of these connections. This is because combinational delay constrains us to connecting ghost outputs at a lower combinational delay level than the corresponding ghost input (though any ghost output can be connected to 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 sub-circuits in addition to simply 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 would only correspond exactly if all edges in the circuit were unit-edges, which is not usually the case.

The definitions are best understood with an example. Figures 4(a) and 4(b) represent combinational sub-circuits which will be glued together into the complete sequential circuit shown in Figure 4(c). The sub-circuit in Figure 4(a) has parameterization[4] $\{n = 7, level = 0, n_{PI} = 3, n_{PO} = 1, n_{edges} =$

[1] By our definitions, a single level would be a purely combinational circuit.

[2] In fact, the gluing algorithm is more general than this, and gen has no restriction against joining nodes at the same sequential level with compatible delays, as long as they are in different sub-circuits. However, the current discussion is limited to sequential circuits which have only a single combinational sub-circuit at each level, as pictured in Figure 3. We have written gen scripts which contain multiple combinational sub-circuits at each sequential level, in order to generate partitioning benchmarks with known cut-sizes. Unfortunately, there is no *automatic* process for writing this kind of fine-grained hierarchy at this time, so the user would have to specify the sub-circuits and the GI/GO interface completely in their gen-script.

[3] This is for two reasons. Firstly, we don't want to register the same signal through two different flip-flops. Secondly, we don't want to generate circuits which map poorly to FPGA logic blocks, which are typically a 4-input LUT followed by an optional flip-flop where only one of the registered and un-registered signals are externally available.

[4] Note that these are partial parameter lists only, as some parameters not relevant to the current discussion of sequential circuits are left out.

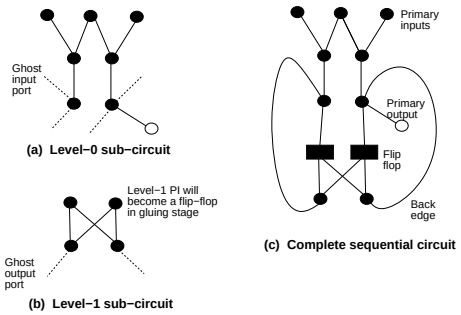


Figure 4: Example construction of a 2-level sequential circuit.

$6, n_{GI} = 2, n_{GO} = 2, n_{latch} = 2, shape = (3, 2, 2), GIshape = (0, 0, 2), GOshape = (0, 0, 2)\}$. The circuit in Figure 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 sub-circuits. Note that the flip-flops serve as primary inputs in the specification of the sub-circuit, 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.

For an exact specification of a sequential circuit, this information is sufficient to generate the complete circuit[5]. An example of a complete exact specification for the MCNC circuit bbtas, output by CIRC as a GEN-script, is shown later in Figure 5, along with example clone-circuits produced from it.

4 The Algorithm for Sequential Generation.

In this section we describe our algorithm for generating sequential circuits. There are two major topics: outlining the modifications to our algorithm to generate base-level sub-circuits with ghost inputs and outputs; and describing the process for gluing sub-circuits together.

4.1 Generating combinational sub-circuits. To generate sub-circuits, we use a modification of the original combinational algorithm [7]. 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. Because the original algorithm is rather long, it is not possible to re-iterate all steps in sufficient detail to convey the changes, so we will restrict the discussion here to a discussion of the most important aspects and refer the interested reader to the software documentation and the public-domain code available from the authors.

Referring to the five steps of the algorithm in Section 2.2, our most important changes involve the identification of registered nodes, ghost inputs and ghost outputs, as follows:

1. The n_{latch} registered nodes must be separated from the other ghost outputs in Step I, because we would like to make these have no other fanouts, if possible, so they must be known before degree allocation.

2. Ghost inputs do not need to be assigned until Step IV, though we do need to take care in earlier steps to allow for ghost inputs in the total combinational fanin of a delay-level. The ghost inputs are assigned randomly and uniformly across the nodes in a level with available fanin.

3. The assignment of non-registered ghost outputs are kept until a new post-processing step VI. Sequential sub-circuits 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.

This process generates 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.

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

When gluing begins, we have a list of sub-circuits C_i, $i = 1..c$ to be connected, sorted by increasing sequential level. Each sub-circuit 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 sub-circuits at non-zero sequential levels which will become flip-flops. Each ghost input and output is attached to a node in the sub-circuit, and inherits the combinational delay of that node.

The matching is constrained by combinational delay and sequential levels. We cannot join a node at sequential level l to a node at level $l+1$, unless that node is a PI (i.e. intended to become a flip-flop). We also cannot join a node to *any* node 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 [9]. 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

[5] However, if a circuit is defined using defaults, sequential user-parameters such as *nDFF* and *nBack* are automatically broken into GI and GO by the first phase of gen.

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 sub-circuits 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 num sub-circuits */
    connect back-edges from other C_j to GIs of C_i.
    connect FF-edges from registered GO
        of C_i to next-level PIs other C_j
end for
```

4.2.1 Locality of connection. We have previously alluded to a "locality metric" in making combinational connections between nodes in Step V. 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 sub-circuit (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 V 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 [5] (e.g. see Figure 2(c)).

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.

4.2.2 Gluing back-edges. The algorithm for gluing back edges to the ghost inputs of one circuit C_i from all other sub-circuits 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 sub-circuits 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 best 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 couple of additional pointers indexed into the destination list, and 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[6] per sub-circuit, 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[7].

The reason that the main algorithm processes sub-circuits 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.)

4.2.3 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 sub-circuits). 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.)

Note that the order in which sub-circuits are considered is unimportant, as the connections are independent.

4.2.4 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 sub-circuits 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 Validating the Quality of GEN-circuits

As mentioned in the introduction, we test the viability of sequential GEN-circuits by generating clones of industrial benchmark circuits, and comparing the post-placement and routing statistics from VPR and MAX+PLUS2 for the original circuit with that of the clone circuit and a equivalently sized (in terms of nodes, edges and I/O) random graph. The circuits referred to here are actual industrial circuits belonging to Altera. The first author was able to perform these experiments while employed there on a summer internship.

Before giving the routing results, we need to describe how we generate the random graphs used for comparison.

5.1 Generating random graphs. We generate a random directed graph on n nodes and n_e edges with n_{PI} primary inputs, n_{PO} primary outputs, with n_{DFF} available flip-flops (for breaking combinational cycles, as we want only synchronous designs) and k_{max}-bounded fanin. The algorithm is as follows.

1. Determine the maximum k such that $2 \cdot k \cdot n$ is less

[6] 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.

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

than n_e. Create a random permutation σ of size $2 \cdot k \cdot n$, to represent $2 \cdot k \cdot n$ nodes, and join nodes σ_{2i} and σ_{2i+1} with an edge, $i = 0..(k \cdot n) - 1$. This creates a graph on $2 \cdot k \cdot n$ nodes with $k \cdot n$ edges, where each node is connected to exactly one other, i.e. a random matching.

2. Now collapse all nodes labeled $\sigma_{ki}..\sigma_{(k+1)i-1}$ into a single node x_i. The result is an n node undirected graph where the degree of each node is exactly k (a k-regular graph[8]) and the distribution of graphs generated is guaranteed to be uniformly distributed over all k-regular graphs of size n.

3. Direct all edges from lower-numbered nodes to higher, to get a directed graph. Randomly label n_{PI} fanin-0 nodes as PI (similarly n_{PO}) fanout-0 nodes as PO). Randomly connect non-labeled fanout-0 and fanout-0 nodes by new edges until they are exhausted, then continue randomly connecting random nodes to random nodes with fanin less than k_{max} until the graph contains n_e edges. When it is necessary to connect a node to a node of a lower number, separate the two by a flip-flop if one remains to allocate, otherwise search for an alternate connection that does not involve a back-edge.

This process generates a graph with the specified number of each node-type and the specified number of edges. A more standard definition random graph (i.e. $G(n, p)$ on n nodes with each edge existing with probability p), would not be an interesting comparison with GEN, because it is much too hard to place and route (e.g. it contains a clique on $\log(n)$ nodes, almost always).

The graphs generated by the above process could be seen as a "first pass" version of GEN which takes fewer parameters into account. In fact, this algorithm alone would be an improvement over most naive approaches to generating random graphs for benchmarks. Comparing real circuits to clones and these random graphs is essentially measuring how far along the scale from "random" to "real" the current GEN approach has traveled.

5.2 Comparing Routing Results. Table 1 shows the comparison between the original, GEN and random circuits after placement and global routing by VPR[9] [3] and implementation on an Altera 10K20-RC240 FPGA [2] by MAX+PLUS2. The benchmarks used are all of the appropriate size (between 60 and 100% logic utilization, with most in the higher end of the range) for exercising this 10K20 part, which has 1152 LCELLS (logic blocks) and 240 user I/O pins.

The first column gives each circuit a name. The second column gives the total wirelength after global routing. Then we give the percentage of extra wiring (beyond that required for the original) required by the corresponding clone circuit and random graph. Similarly, we then have the track-count (channel width) followed by the percentage increase in track-count for the corresponding clone circuit and random graph. The last two columns show the percentage increase in "routing resources" used by the clone circuit and the random circuit when implemented on the 10K20 FPGA. To respect information about the benchmark circuits which is proprietary to Altera the actual resource usage in the device is not displayed—for this study it is only the percentage difference that is of interest.

	VPR wire			VPR tracks			10K20 tracks	
		clone	rand		clone	rand	clone	rand
Circuit	orig	%diff	%diff	orig	%diff	%diff	%diff	%diff
A	5102	21	144	6	16	83	14	132
B	7719	64	215	5	80	160	71	.
C	6344	27	160	6	16	116	30	.
D	6818	20	147	6	16	133	32	.
E	6609	53	266	5	60	160	35	.
F	4293	57	188	5	40	140	41	197
G	4147	2	158	5	0	140	16	208
H	5107	21	137	5	40	120	0	123
I	4692	19	155	5	40	160	23	132
J	6087	34	153	5	60	120	51	165
K	9313	42	202	6	33	133	38	.
L	6546	36	222	6	33	100	55	.
M	7748	86	248	5	100	220	85	.
N	10794	-43	52	10	-40	30	-41	.
O	8070	17	140	7	14	100	25	.
P	5562	88	268	5	80	180	90	.
Q	6460	71	167	5	80	160	.	.
S	6417	29	166	5	40	140	24	.
T	4662	28	170	6	0	83	16	108
U	8828	2	156	6	16	150	53	.
V	4876	81	201	4	75	175	63	174
W	4837	28	143	4	50	150	34	117
mean	6358	35%	175%	5.5	38%	134%	36%	151%

Table 1: Routability comparisons between original benchmark circuits, GEN-clones and random graphs ('.' indicates a no-fit).

For our metric of FPGA resource usage, we count the total number of full-horizontal, half-horizontal and vertical lines used by the design in a 10K20, as reported by MAX+PLUS2. Because we are using an actual device, it is possible that a design does not "fit." Though all original circuits do fit in the 10K20, 1 of the clone circuits and 13 of the random graphs did not, and these are indicated by a '.' in the table.

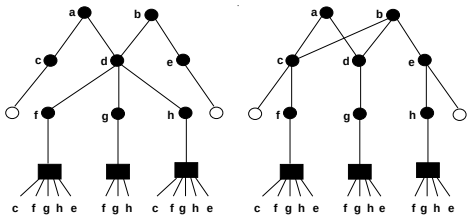
The last row of the table indicates the averages for each column. For the last two columns, the missing data is *not* included in the average.

We find that the clone circuits are, in general, harder to place and route than are the original circuits we took the specifications from, though a given clone is always closer to the original than the corresponding random graph. On average, the clone circuits used 35% more wirelength and 38% more tracks than the original circuit, whereas the random graphs used graphs used 175% more wirelength and 134% more tracks. This is further reflected in the implementation of the clone and random circuits on the commercial FPGA where (when they did fit) the clone circuits used an average of 36% more routing resources and the random graphs used 151% more routing resources. We also find that about half of the random graphs do not fit at all in the part, whereas only one clone failed to fit. In [6, 7] we give the definition of a measure quantifying reconvergence in a circuit. By this measure, GEN circuits differ by about 0.19 on average, while random graphs differ by 0.28 on average.

These results show that the GEN clone circuits are significantly more realistic than the random graphs. However, the GEN circuits are also harder to place and route relative to the

[8] There are details to deal with the double and and self-connections between nodes without sacrificing the uniform distribution, but these are beyond the current discussion.

[9] Vpr uses the model of a symmetric array of logic blocks, similar to a gate-array or a Xilinx 4000 series FPGA, and reports the total wirelength and the maximum channel width (number of tracks) used after global routing.



(a) Circuit bbtas (b) Clone circuit

```
/* CIRC 3.0, compiled Wed Aug 28 15:36:17 PDT 1996. */
X = { name="bbtasclone";
L0=(@.comb_circ) { name="L0"; n=8; kin=4; nPI=2; nPO=2;
 nDFF=0;nEdges=7;level=0;delay=2; nBot=3; shape=(2,3,3);
 nGI=13;GIshape=(4,9,0);nGO=3; GOshape=(0,0,3); nZeros=5;
 POshape=(0,2,0);edges=(0,7,0);outs=(5,0,2,1);max_out=3;};
L1=(@.comb_circ) { name="L1"; n=3; kin=4; nPI=0; nDFF=3;
 level=1; delay=0; nEdges=0; nBot=3; shape=(3); nGI=0;
 GIshape=(0); nGO=13; GOshape=(13); nPO=0; nZeros=3;
 POshape=(0); edges=(0); outs=(3); max_out=0; };
glue=(L0, L1);
};
output(circuit(X));
```

(c) Clone script, produced by CIRC.

Figure 5: The MCNC circuit bbtas, its clone script from CIRC, and a resulting clone produced by GEN.

originals. We believe that a greater amount of local clustering is required, and we are currently exploring methods to provide this.

5.3 An Example. Here we present a small example that helps to understand the overall operation of GEN, and the type of variation that can occur in generating a clone. Figure 5 shows a picture of the MCNC circuit "bbtas," and a clone circuit produced by its GEN-script from CIRC. Note that we use node labels to illustrate back-edges to improve readability.

Two aspects that the parameterization does not capture are *symmetry* and the type of *locality* that would be reflected in the block-diagram of the circuit. We observe that the clone circuit exhibits less symmetry than the original, and in larger circuits we can see identifiable block-structure in the original design which is not passed on to GEN for duplication. Note, however, that re-capturing the block structure and symmetry in a flat netlist are open (and very difficult) research problems of their own. These problems will likely have to be tackled (or simulated with a model in GEN) in order to further increase the quality (routability) of the benchmark circuits generated.

6 Conclusions and Further Work

In this paper we have defined a new model for describing sequential circuits as a hierarchy of combinational sub-circuits. The model includes the parameters of ghost inputs, ghost outputs and their delay-shapes. The model can also be used to describe more general forms of hierarchy than simply that between sequential levels. We have given an algorithm for generating realistic sequential benchmark netlists given the exact parameterization of a circuit in this model. This builds on previous research in which we gave a similar algorithm for the simpler problem of purely combinational circuits.

In addition, we have described a public-domain[10] prototype software system which implements the sequential model with a characterization program (CIRC V3.1) and a generation program (GEN V3.1). Using the software, we have "cloned" a number of industrial benchmark circuits, and showed that GEN-circuits are significantly closer to real circuits (in terms of placement and routing statistics) than carefully generated random graphs.

GEN is also capable of generating circuits "from scratch" using a set of default scripts based on analysis of benchmark circuits, and which can be user-modified. The software executes quickly, and can generate circuits for current FPGA sizes with only a few minutes of CPU time. GEN is able to produce circuits in a number of netlist formats, including Actel ADL, Altera AHDL (TDF) and Xilinx XNF.

CIRC and GEN prototypes have been installed for use at Xilinx, Altera and Actel, several CAD software companies, as well as other industrial and academic sites. In addition, we have contributed benchmarks created by GEN to an informal partitioning competition at the 1996 Design Automation Conference (organized by Franz Brglez), and to other partitioning researchers[1].

We see a number of areas for future exploration. One is to modify the base generation algorithm to automatically impose a partition hierarchy on the circuit as it is being built, possibly similar to Darnauer and Dai's [4] use of the Rent-exponent to introduce hierarchy in their partitioning benchmarks. Though GEN will currently output circuits of up to about 100,000 LUTs (about 20 times the size of a modern FPGA), we believe generating high-quality large benchmarks will require some degree of imposed symmetry and hierarchy within the netlist. A second area for future work would be to generate "system"-level hierarchy, by including datapath and other structured logic which can be synthesized or produced with LPM modules and random logic components from GEN.

Acknowledgements: The authors would like to thank Vaughn Betz for use of VPR. Special thanks to Altera Corporation for providing the first author with a summer internship, during which parts of this research were performed, and for allowing access to their benchmark circuits during that time, and to Hewlett Packard Corporation for financial support.

[10] See http://www.cs.toronto.edu/~mdhutton/gen or http://www.eecg.toronto.edu/~jayar for details.

References

[1] C. Alpert, *Private communication.* UCLA and IBM Austin.

[2] Altera Corporation, *1996 Data Book.*

[3] V. Betz and J. Rose, *Directional Bias and Non-Uniformity in FPGA Global Routing Architectures*, in 14th IEEE/ACM Int'l Conference on Computer-Aided Design, 1996, pp. 652–659.

[4] J. Darnauer and W. Dai, *A Method for Generating Random Circuits and Its Application to Routability Measurement*, in 4th ACM/SIGDA Int'l Symp. on FPGAs, Feb., 1996, pp. 66–72.

[5] E. R. Gasner, E. Koutsofios, S. C. North, and K.-P. Vo, *A Technique for Drawing Directed Graphs*, IEEE. Trans. Soft. Eng., 19 (1993), pp. 214–230.

[6] M. D. Hutton, *Characterization and Generation of Digital Benchmark Circuits.* Ph.D. Thesis in preparation, University of Toronto, 1996.

[7] M. D. Hutton, J. P. Grossman, J. S. Rose, and D. G. Corneil, *Characterization and Parameterized Random Generation of Digital Circuits*, in 33rd ACM/SIGDA Design Automation Conference (DAC), June., 1996, pp. 94–99. (Journal version in preparation.).

[8] Programmable Electronics Performance Corporation, *PLD Benchmark Suite#1, V1.2.* 504 Nino Ave. Los Gatos, CA 95032, 1993.

[9] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.